Universidade Federal de Pernambuco
Centro de Informática
Departamento de Sistemas de Computação

Graduação em Ciência da Computação

Avaliando o Microsoft Solutions Framework for Agile Software Development em relação ao Extreme Programming

Ivan Cordeiro Cardim

TRABALHO DE GRADUAÇÃO

Recife
13 de Fevereiro de 2006

Universidade Federal de Pernambuco
Centro de Informática
Departamento de Sistemas de Computação

Ivan Cordeiro Cardim

Avaliando o Microsoft Solutions Framework for Agile Software Development em relação ao Extreme Programming

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Departamento de Sistemas de Computação da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Prof. Dr. Alexandre Marcos Lins de Vasconcelos*

Recife
13 de Fevereiro de 2006

A minha família

Agradecimentos

"Quidquid latine dictum sit, altum sonatur"
(Tudo falado em latim parece profundo)
—AUTOR DESCONHECIDO

É com um imenso prazer que eu concluo mais esta fase da minha vida. O caminho percorrido até aqui foi árduo, mas intensamente agradável pelo simples fato que eu não precisei percorrê-lo sozinho. Eu tive e tenho a sorte de estar sempre bem acompanhado. Agradecer a todas estas pessoas que me ajudaram a chegar até aqui parece mais difícil do que realizar todo o percurso novamente.

Em primeiro lugar gostaria de agradecer à minha família, que é fortemente responsável por tudo aquilo que sou hoje. Não consigo imaginar maneira melhor de ter sido criado, e me orgulho de cada valor que me foi ensinado por meus pais. Me orgulho também de tudo que pude viver e aprender com meus irmãos até agora, e de tantas outras coisas que ainda vamos viver juntos.

A grandes tutores que eu tive, por todos os ensinamentos dentro e fora da sala de aula: Paulo Borba, Fernando Fonseca, Vander Alves e Alexandre Vasconcelos.

A Raony e Ricardo pelas valiosas revisões e discussões filosóficas a respeito deste trabalho.

E a tudo de bom que eu vivi neste período de quatro anos e meio. Doces amizades, belos amores. A vontade de ser sempre melhor, a vontade de saber sempre de tudo. Aos velhos companheiros, e aos novos que encontrei aqui. E a tanta coisa mais que chega dá preguiça de escrever.

Batatinha quando nasce
Se esparrama pelo chão
Menininha quando dorme
Põe a mão no coração
—AUTOR DESCONHECIDO

Resumo

Este trabalho se propõe a estabelecer uma análise comparativa entre o *Extreme Programming* (XP) e uma das vertentes da versão 4.0 do *Microsoft Solutions Framework* (MSF), intitulada *MSF For Agile Software Development*. São apresentadas breves introduções ao XP e ao MSF 4.0, e a seguir, é feita uma análise comparativa que tenta estabelecer pontos de mapeamento e comparação do MSF 4.0 em relação ao XP. Através desta comparação, este trabalho visa analisar a aderência do MSF a XP, a metodologia ágil mais famosa atualmente.

Palavras-chave: Processos de Software, Processos ágeis, MSF, XP

Abstract

This work intends to make a comparative analisys between Extreme Programming (XP) and one of the versions of the MSF 4.0, named Microsoft Solutions Framework for Agile Software Development. Brief introductions are given on XP and MSF 4.0, and then the comparative analisys is made in order to establish comparison points of MSF 4.0 in relation to XP. Through this analisys, this work intends to determine the compliance level of the MSF to XP, the most famous agile methodology at the moment.

Keywords: Software Processes, Agile Processes, MSF, XP

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

Introdução

Programmer's Drinking Song sung to the tune of "100 Bottles of Beer":
99 little bugs in the code,
99 bugs in the code,
fix one bug, compile it again,
101 little bugs in the code.
101 little bugs in the code....
(Repeat until BUGS = 0)
—AUTOR DESCONHECIDO

A engenharia de software tem suas origens na década de 1960, em uma conferência que visava entender e estudar os principais problemas que envolviam a indústria de software na época [14]. Até então vivia-se o que ficou conhecido como a *crise do software*: projetos excedendo suas previsões de cronograma e orçamento e mesmo assim resultando em software de baixa qualidade, não atendendo a seus requisitos corretamente.

A indústria do software, por ser muito recente, não dispunha ainda de métodos que possibilitassem que o desenvolvimento de software fosse feito como era a construção de um edifício. Os esforços que vieram nessa época giraram em torno da criação de um modelo de desenvolvimento de software mais organizado.

Por volta de 1970, foi criado o modelo cascata de ciclo de vida de software [15], que estabelecia o fluxo de desenvolvimento como uma seqüência das fases de análise de requisitos, projeto, implementação, testes e manutenção. Essa abordagem ainda era um pouco ingênua, pois imaginava que o fluxo do projeto seguia constantemente por essas etapas, quando no mundo real este fluxo não acontece de maneira tão linear.

Em meados de 1980, criou-se um processo unificado de software, que evoluiu para o *Rational Unified Process* (RUP) [5] posteriormente. Foi realizado um estudo que buscava estudar as características de vários projetos de software que haviam dado errado, para tentar identificar as principais causas dos erros. Com base nisso, e em outros processos de software existentes na época, nasceu o RUP, um conjunto de boas práticas de desenvolvimento de software. Uma das principais idéias do RUP é, ao invés de realizar um ciclo como o proposto no cascata, quebrar o desenvolvimento em várias iterações, cada uma das quais podendo conter atividades de análise de requisitos, projeto, implementação e testes, porém em níveis de atividade diferentes dependendo da fase em que o projeto se encontra.

Apesar de todo o esforço para a criação e o uso de frameworks como o RUP, o desenvolvimento de software ainda hoje apresenta diversos problemas e boa parte dos projetos ainda falha

em alcançar seus objetivos. Uma das explicações sugeridas para tal fato é que frameworks como o RUP ainda são muito burocráticos e exigem planejamento demais.

Durante o fim dos anos 90 foram desenvolvidas técnicas de desenvolvimento de software chamadas "leves", que tentavam não se prender a um planejamento muito detalhado do futuro e focavam em desenvolver versões funcionais de software em períodos curtos de tempo. Em 2001, um grupo de 17 pessoas conhecidas na área de "processos leves", se reuniram para definir as bases comuns às suas metodologias. O resultado disso foi o *manifesto ágil* [1], uma definição básica das filosofias que um processo ágil (que desde então passou a ser o novo termo utilizado para "leve") deve seguir.

Inicialmente pregava-se que as metodologias ágeis atendiam apenas projetos com equipes pequenas, pois suas práticas por serem simples demais não seriam escaláveis. Seus criadores e entusiastas no entanto defendem que os métodos ágeis podem se adequar a projetos grandes também, e apresentam resultados de projetos bem sucedidos para comprovar [8]. Com o crescimento e a divulgação destas metodologias, criou-se uma tendência de várias outras metodologias em aproveitar práticas que vêm sendo consolidadas na área ágil. Um indício disto é que metodologias que não são vistas como ágeis, como o RUP, estão abrindo espaço para atender à demanda por agilidade [3].

O *Extreme Programming* (XP) [4] é a mais popular das metodologias ágeis de desenvolvimento de software. O foco principal da metodologia é o baixo custo de mudança; XP se propõe a reduzir o custo de mudança dos seus projetos introduzindo uma série de valores, princípios e práticas que primam por iterações curtas e comunicação freqüente com o cliente.

O *Microsoft Solutions Framework* [13] surgiu em 1994 como um conjunto de boa práticas coletadas pela Microsoft a partir de sua experiência na produção de software e em serviços de consultoria. Desde então, o MSF evoluiu, tornando-se um framework flexível para guiar o desenvolvimento de projetos de software. Recentemente, uma nova versão do MSF (a 4.0) foi lançada; a principal diferença em relação à anterior foi uma concretização maior da mesma. O estabelecimento de papéis bem definidos, a instanciação das boas práticas em fluxos de trabalho e atividades, além da criação de itens de trabalho, consistem em parte dessa concretização.

Dentro do contexto da popularização dos processos ágeis, a versão 4.0 do MSF foi lançada em duas variações: uma abordando processos ágeis e outra processos tradicionais. A variação "ágil" do MSF 4.0 tem o nome de MSF For Agile Software Development, que será foco de estudo nesse trabalho, e que será chamado a partir daqui de MSF Ágil por questões de simplicidade. A vertente "tradicional", o *MSF for CMMI Process Improvement* [11], está fora do escopo deste trabalho e não será abordada aqui. Mais informações sobre o mesmo podem ser encontradas em um estudo semelhante a esse [2], que faz uma comparação entre o MSF tradicional e a PRO.NET [7], uma outra metodologia de desenvolvimento de software.

Boa parte das metodologias atuais existem em forma de *frameworks*, ou seja: um conjunto bastante abrangente de práticas e técnicas que visa atender a uma gama de projetos a mais vasta possível. Um grande problema com isso é conseguir escolher um subconjunto adequado do *framework* para um determinado projeto, de forma que não sejam englobadas práticas que não seriam úteis ao projeto.

Visamos com este trabalho estabelecer um paralelo entre o MSF Ágil e o *Extreme Programming*, mapeando porções do primeiro para o segundo e verificando a aderência do novo MSF à

práticas ágeis.

Além deste capítulo introdutório, este trabalho está organizado nos seguintes capítulos: o Capítulo 2 analisa a nova versão do *Extreme programming*, e as novas características que foram introduzidas por Kent Beck [4]; o capítulo 3 apresenta o MSF 4.0, com foco em sua vertente ágil. O capítulo 4 realiza uma comparação entre as duas metodologias, verificando a aderência do MSF Ágil ao XP e o capítulo 5 tece as conclusões e possibilidades de trabalhos futuros.

CAPÍTULO 2

Extreme Programming

"It ain't what you don't know that gets you in trouble. It's what you know that ain't so."
—WILL ROGERS

No início de 1996, Kent Beck integrava um projeto chamado Chrysler Comprehensive Compensation (C3), que tinha como intuito substituir as várias aplicações de folha de pagamento que a Chrysler possuía por uma só. Com a recente popularização da orientação a objetos que estava ocorrendo na década de 90, era de interesse do projeto que a linguagem utilizada para a criação da aplicação fosse Smalltalk. Beck entrou inicialmente no projeto para realizar aprimoramento de performance, por ter bastante experiência com a linguagem. À medida que trabalhava lá, ele foi percebendo uma série de problemas no processo de desenvolvimento do projeto e propôs melhorias para ele, com base em estudos realizados anteriormente com Ward Cunningham.

Mais tarde Beck acabou se tornando o líder do projeto e convidou Ron Jeffries para trabalhar junto com ele em cima de suas novas idéias e lhe ajudar a aplicá-las ao projeto C3. Ele escreveu um livro chamado *Extreme Programming Explained*, que foi publicado em outubro de 1999, uma compilação e avaliação de todas essas técnicas em forma de uma metodologia. Desde então, diversos livros sobre a metodologia já foram publicados e uma comunidade bem ativa se criou ao redor de XP. Em 2004, a segunda versão do *Extreme Programming Explained* foi lançada, reformulando substancialmente a metodologia existente para acomodar novas características e abandonar algumas que não se mostraram efetivas ou que não tiveram uso. Neste capítulo iremos dar uma visão do que é o novo *Extreme Programming*, fazendo referências a mudanças em relação à versão antiga sempre que necessário.

2.1 Visão Geral

O XP é uma abordagem ágil para o desenvolvimento de software que prima pela adaptabilidade a mudanças em detrimento do tradicional planejamento demasiado, com o intuito de realizar previsões a longo prazo. O seu objetivo principal, como o de várias outras metodologias, é possibilitar desenvolvimento de software com excelência. Mais do que apenas um conjunto de métodos úteis para o desenvolvimento de projetos, XP também comporta toda a filosofia de mudança social que embasa cada uma destas práticas. Um dos principais motivos pelo qual o desenvolvimento de software até hoje produz muito menos do que é capaz é por falhar em ver a pessoa, o indivíduo, como protagonista. A maneira como XP é organizado tenta ser o

mais natural possível para todas as pessoas envolvidas, favorecendo desta forma a eficiência das mesmas.

XP prega que o desenvolvimento de software deve ser baseado em cinco *valores*: *comunicação*, *feedback*, *simplicidade*, *coragem* e *respeito*. Estes valores funcionam como invariantes do projeto e devem nortear todo o seu desenvolvimento. São como os objetivos que devem estar presentes em qualquer projeto para que ele dê certo. Todas as atividades existentes em XP devem ser motivadas por algum valor. Existem atividades que podem ser úteis num projeto e inúteis em outro, dependendo de contexto. Por exemplo, num projeto de código aberto com colaboradores espalhados pelo mundo, é natural que se mantenha uma lista de *bugs*[1] num Wiki[2]. No entanto, isso seria desnecessário e inclusive contraproducente caso o projeto fosse pequeno, com apenas dois integrantes que trabalham no mesmo local. Nesse caso, um simples documento de texto atenderia o propósito de documentar os *bugs*. O importante na escolha de que opção adotar é garantir que ela seja o meio para favorecer a *comunicação* no projeto. Atividades como esta são denominadas de *práticas*; elas devem refletir os valores de XP.

Práticas são atividades concretas e bem definidas. Na primeira versão do *XP Explained*, existia um conjunto fechado de doze práticas, que deveria ser atendido em sua totalidade. Como mencionado logo acima há muitas práticas que são dependentes de contexto. É importante atentar para o fato de que esse contexto inclui também as pessoas envolvidas. Mesmo em cenários parecidos, uma determinada equipe pode se adequar a certas práticas e outra não, devido a particularidades de seus integrantes. Ao criar um conjunto de práticas imutáveis, XP estava indo contra a naturalidade que tentava trazer, tentando controlar demais as equipes. Além do mais, era um pensamento inocente imaginar que seria possível cobrir todos os problemas de desenvolvimento de software com um conjunto fixo de práticas. A metodologia amadureceu e agora as práticas não são mais fixas. Agora, existe um conjunto novo de práticas sugeridas, que contém algumas das idéias das práticas antigas e outros elementos completamente novos. Estas devem ser adotadas ou não dependendo da ocasião, nada impedindo que práticas diferentes sejam utilizadas desde que elas sejam suportadas por valores. Isso acontece de maneira um pouco parecida com os valores: nada impede que um time adote outros valores além dos cinco mencionados. No entanto, estes cinco são os valores que representam a filosofia XP. A tentativa de atender a esses objetivos é o que caracteriza um projeto XP.

Tomemos os valores como sendo objetivos abstratos da metodologia e as práticas como sendo atividades pontuais e bem definidas que descrevem de fato como desenvolver software. Devido ao fato dos valores serem muito abstratos, é possível definir uma vasta gama de práticas que podem estar de acordo com cada um deles. Estas práticas podem ser boas ou ruins, dependendo do contexto; os *princípios* são características que devem ser observadas ao tentar aplicar uma prática para atender a um valor. Eles dão uma idéia mais detalhada do que as práticas devem fazer. Os princípios são a ponte entre as práticas e os valores, e devem ser usados para entender como aplicar uma prática num contexto. Assim como os valores, o conjunto de princípios é fixo na metodologia: nada impede que mais princípios sejam utilizados, mas existe o conjunto básico de princípios que define a metodologia XP. Este conjunto de princípios deve ser usado como guia junto com os valores para escolher que práticas serão utilizadas.

[1] Termo utilizado na área de computação para denotar um defeito de *software*.

[2] Uma página na internet que permite a seus usuários adicionar e editar seu conteúdo coletivamente

Metodologias como o RUP existem em forma de *framework*: um conjunto bem definido e concreto de práticas que guiam o desenvolvimento de software de maneira bastante detalhada. XP pode ser visto dessa maneira mais simplista, como uma metodologia concreta, que possui suas práticas bem definidas e detalhadas. Atualmente, no entanto, é mais comum que ele seja visto por uma ótica mais subjetiva e abrangente, que dá mais importância as filosofias por trás das práticas. No entando, não há uma visão oficial que determine que ótica deve ser adotada. Apesar de existirem criadores para XP, não existe um comitê que tome decisões oficiais sobre o que é e o que não é a metodologia. A evolução de XP se dá através da comunidade que o usa.

As próximas seções subseções listam e detalham os valores, princípios e práticas sugeridos por Kent Beck na segunda edição do *Extreme Programming Explained*.

2.2 Valores

Os valores são objetivos principais que devem tentar ser alcançados em qualquer projeto XP, independente de seu contexto. Qualquer prática que for adotada no projeto deve em última instância servir para tentar atender um ou mais destes valores. Uma prática sem um valor para lhe sustentar é somente uma atividade sem significado e não é útil para o projeto. Abaixo, é apresentada uma descrição mais detalhada de cada um dos valores adotados por XP.

Comunicação: boa parte dos problemas no desenvolvimento de software ocorre por falta de comunicação, ou de comunicação não efetiva. Quando um problema ocorre, é muito provável que alguém na equipe já saiba a solução para ele, por já ter passado pelo problema ou que possa prover informações relevantes para a sua solução, por já ter passado por alguma situação parecida. O problema é fazer com que haja a comunicação entre a pessoa com o problema e a pessoa com a solução. XP procura deixar a comunicação fácil e sempre fluente entre todos os integrantes do projeto, inclusive os clientes, que também fazem parte da equipe.

Simplicidade: XP tenta não pecar por excesso através desse valor. Faça algo o mais simples possível de uma maneira que funcione. Inicialmente, deixe de lado toda a complexidade adicional que geralmente se tem ao imaginar soluções que funcionem para casos que não são realidade ainda. Depois melhore sua solução gradativamente, se necessário. Não se preocupe com nada que não vá ser necessário agora. Faça as coisas o máis fácil possível, para favorecer sua equipe.

Feedback: é comunicação, mas é uma parte tão importante dela que é destacada como um valor diferente. As condições de um projeto mudam bastante durante o seu curso, e o que é certo hoje pode não ser amanhã. Requisitos, por exemplo, mudam o tempo todo. Desenvolver *software* é um processo de aprendizado: muitas vezes o cliente aprende o que quer durante o projeto, assim como o programador aprende a fazer aquilo durante o projeto. *Feedback* é a maneira mais segura de aprender se o que foi feito até agora está certo. Também é através de *feedback* constante que se pode acompanhar todas as mudanças que estão ocorrendo, e reagir a elas em tempo hábil e de maneira adequada.

Coragem: : pode ser visto como um valor de suporte aos outros valores. Descartar uma solução em andamento ao perceber que ela não vai apresentar os resultados esperados e começar outra do início requer coragem. Fazer mudanças substanciais na arquitetura requer coragem. Confrontar opiniões de outros colegas e falar verdades, mesmo que desagradáveis requer coragem, mas favorece o andamento e pode ser crucial para o sucesso do projeto.

Respeito: é um pré-requisito para que a metodologia funcione. Respeito significa entender que nenhuma das pessoas envolvidas no desenvolvimento de software é mais importante do que outra, e que as opiniões e as necessidades de todas elas precisam ser respeitadas. Se o projeto não está sendo levado a sério pela equipe, não há como ele dar certo.

Como já foi dito, estes não são os únicos valores possíveis para melhorar o desenvolvimento de software. No entanto, um time que adotasse outros valores não estaria seguindo a metodologia XP. Estes valores é que representam os ideais da metodologia. Um time poderia adotar valores adicionais além destes, mas é necessário que haja cuidado pois não pode haver contradições entre os valores de um time uma vez que todos eles precisam ser mantidos sempre. Por exemplo, se um dos valores de um time é previsibilidade, haveria práticas que focariam num *design* forte no início do projeto. No entanto, essas práticas estariam indo de encontro a práticas de XP como *Design Incremental*, que visam atender a outro valor (*feedback*). Desta forma, fica impossível atender um valor sem entrar em conflito com outro.

Uma característica importante a se observar aqui é a sinergia entre os valores. Seguir exageradamente algum dos valores poderia acabar sendo prejudicial para o projeto. Um valor também influência outro valor, com o intuito de manter um equilíbrio entre os objetivos. A idéia é que a sua junção seja mais que a soma das suas partes.

Suponha o exemplo dado anteriormente, de um time de programadores espalhado pelo mundo que quer manter uma lista de *bugs* de um projeto. A lista poderia ser um documento do word, que ficaria em uma página na internet e poderia ser baixada por todos a qualquer momento. Isso estaria facilitando a comunicação entre todos. No entanto, com várias pessoas precisando modificar o arquivo, as pessoas precisariam estar baixando constantemente novas versões do documento. Essa comunicação precisa ser mais simples para acomodar a essas mudanças constantes e prover *feedback* correto. Mantendo a lista através de um Wiki, é possível que a atualização seja feita de maneira simples, através da web e que todo o documento esteja sempre atualizado, num único lugar.

2.3 Princípios

Os princípios de XP realizam a ligação entre os valores e as práticas. Eles servem para guiar a escolha e criação de práticas adequadas aos princípios. Existem ao todo 14 princípios. Todos eles são listados e detalhados abaixo.

Humanidade: é a preocupação com a pessoa por trás do desenvolvimento de sofrwtare. Software é desenvolvido por pessoas, logo fatores humanos são a principal chave para desenvolver software de qualidade. XP objetiva atender aos objetivos das pessoas e de suas

organizações, de modo que possa beneficiar ambos. Muitas vezes no entanto não é possível atender aos objetivos de todos, então é necessário encontrar o equilíbrio entre eles. Se superestimamos as necessidades das pessoas, elas não trabalharão suficientemente, o que acarretará em baixa produtividade e posteriormente perdas para o negócio. Se por outro lado, você superestima as necessidades da organização, vai haver ansiedade, trabalho além do normal e conflitos. Isso também não é bom para o negócio. A idéia é balancear os dois. As necessidades das pessoas são:

- Segurança Básica - a necessidade de manter o seu emprego
- Realização - o sentido de utilidade no seu próprio trabalho
- Pertinência - a habilidade de se identificar com o grupo
- Crescimento - a oportunidade de expandir suas habilidades e perspectivas
- Intimidade - a habilidade de entender e ser entendido por outros

As práticas de XP atendem tanto às necessidades pessoais quanto as de negócio. Novas práticas que porventura venham a ser inseridas devem levar esse quesito em consideração.

Economia: quando se produz software também se gera valor de negócio. Dois aspectos econômicos são cruciais para XP: valor atual e valor de opções. O primeiro considera que o mesmo valor de dinheiro vale mais hoje, do que amanhã. Por isso, quanto mais cedo se ganha dinheiro e quanto mais tarde se gasta, maior é o lucro que o software gera. Isto é relacionado ao valor de opções: é de extrema valia para o negócio protelar os investimentos em *design* até que eles sejam óbvios. Algumas práticas como *Design Incremental* representam essa abordagem.

Benefício Mútuo: toda atividade deveria beneficiar todas as pessoas e organizações envolvidas. É necessário resolver mais problemas do que se cria. Este é possívelmente o princípio mais importante de XP. Sempre irão existir soluções simples para qualquer problema, onde alguns saem ganhando e outros perdendo. Frequentemente, estas soluções serão atalhos tentadores. No entanto, elas serão sempre uma perda coletiva, pois podem prejudicar relacionamentos e o ambiente de trabalho. Mais uma vez, é importante que as práticas beneficiem todos os integrantes da equipe, agora e no futuro.

Auto-semelhança: quando uma solução dá certo, ela deve tentar ser empregada sempre que possível e adequado. Se tomarmos como exemplo uma prática comum que é escrever testes que falham para depois escrever código que passa nos testes, podemos observar a recorrência deste cenário em vários níveis. Num trimestre, se listam os problemas e serem resolvidos, e depois estórias que os resolvem são escritas. Numa semana, as estórias que devem ser implementadas são listadas, seguidas pelos testes de aceitação que as validam e por fim pelo código que faz os testes funcionarem.

Aprimoramento: melhora constante é fundamental para XP. A perfeição nunca é alcançada; no entanto, deve sempre ser perseguida. As ações de todos envolvidos no projeto devem tentar ser as melhores possíveis, e mesmo quando se consegue bons resultados deve se imaginar como melhorá-los.

Diversidade: equipes onde todos são parecidos podem ser confortáveis, mas não são efetivas. Times devem comportar diferentes conhecimentos, habilidades e personalidades para poder atender a uma sorte mais abrangente de problemas. É claro que essas diferenças também acarretam em maior possibilidade de conflitos, mas estes devem ser resolvidos. Ter diferentes opiniões e propor diferentes soluções é bastante útil no desenvolvimento de software, uma vez que se consiga tirar proveito disso, gerenciando os conflitos e escolhendo as melhores alternativas.

Reflexão: um time eficiente não faz apenas o seu trabalho. Eles se perguntam como estão trabalhando e porque estão trabalhando do jeito que estão. Eles precisam analisar as razões por trás do sucesso (ou falha). Eles precisam tornar seus erros explícitos e tentar aprender a partir deles. É necessário parar um pouco durante os ciclos trimestrais e semanais para refletir sobre o andamento do projeto, e que melhorias seriam possíveis.

Fluxo: o desenvolvimento do software deve seguir um fluxo constante, no qual todas as atividades são realizadas simultaneamente. O resultado final, ou seja, o programa, deve estar sempre disponível e sendo atualizado. As práticas assumem um fluxo contínuo de atividades, e não uma sequência de fases que só gera resultado ao término da última. O fluxo contínuo possibilita *feedback* frequente, o que serve para garantir que o sistema está evoluindo na direção certa, evitando problemas relacionados a integração apenas no fim do projeto.

Oportunidade: problemas devem ser vistos como uma oportunidade para o aprimoramento. Sempre haverá problemas; para obter a excelência, é mister fazer mais do que apenas corrigi-los. É necessário aprender a transformá-los em oportunidades de aprendizagem e melhoria. Por exemplo, para suprir a deficiência de planejamento a longo prazo, é possível fazer uso de ciclos trimestrais e rever o planejamento sempre que um problema ocorrer. Com o tempo e a prática, o planejamento se aproxima da realidade.

Redundância: problemas críticos e difíceis devem ser resolvidos de várias maneiras. Desta forma, se uma solução falha, as outras vão impedir um desastre. O custo da redundância é menor do que o custo da falha. Os defeitos no software devem ser atacados de diversas maneiras (pair programming, testes automáticos, sit together, etc.) Os mesmos defeitos serão encontrados de maneiras diferentes. No entanto, a qualidade decorrente disso não tem preço.

Falha: muitas vezes o erro é caminho para o acerto. Se não há uma maneira conhecida de resolver um problema, é recomendado que se tentem de várias maneiras diferentes. É possível que nenhuma delas funcione, mas mesmo nesse caso haverá aprendizado. Uma falha tem sua valia, desde que haja aprendizado a partir dela. Não deve haver medo da falha. É melhor tentar algo e falhar ao invés de esperar durante muito tempo, tentando fazer as coisas certas desde o início.

Qualidade: a qualidade sempre precisa ser máxima. Aceitar um baixo nível de qualidade não estimula a economia ou um desenvolvimento mais rápido. Pelo contrário, a melhora

da qualidade traz necessariamente um acréscimo a outras características de um sistema, como produtividade e eficiência. Além do mais, qualidade não é apenas um fator de economia. Membros da equipe devem ficar orgulhosos do seu trabalho porque este melhora a auto-estima do time. É necessário não confundir qualidade com perfeccionismo, entretanto. Se uma ação é protelada devido a uma busca por qualidade excessiva, a qualidade não está sendo promovida de fato. É preferível tentar e falhar, e então refinar as soluções imperfeitas encontradas.

Passos Pequenos: mudanças de grande porte, preparadas durante um longo período para acontecer de uma vez só são perigosas. É muito mais seguro proceder iterativamente em passos pequenos: o menor passo possível que esteja garantidamente na direção certa. Passos pequenos não significam pouca velocidade. Uma vez que os passos pequenos realizam menos mudanças por vez, é possível dar vários passos no tempo que se daria apenas um passo grande. Passos pequenos também oferecem mais segurança: um passo pequeno errado impacta menos e é mais facilmente corrigido do que um passo grande, que ao falhar pode comprometer bastante o andamento do projeto.

Responsabilidade Aceita: responsabilidade não pode ser atribuída, ela deve ser aceita. É possível dizer a seus programadores que façam uma coisa ou outra, mas isso não é o ideal, por vários motivos: você pode estar pedindo menos do que se poderia fazer, ou pode também pedir mais do que se pode fazer (que é o que normalmente acontece). Determinado desenvolvedor pode ser mais apropriado do que outro para realizar uma tarefa, e é importante que ele tenha abertura para poder dizer isso e chamar essa responsabilidade para si.

2.4 Práticas

As práticas têm a função de dar forma aos valores. Elas são a parte mais rasa da metodologia, atividades que o time de desenvolvimento estará executando no dia a dia. Apesar de não serem detalhadas no nível de um fluxo de trabalho que deve ser seguido passo a passo, são o que mais se assemelha a isso na metodologia. Inicialmente, XP propunha 12 práticas. Após algum tempo em uso, percebeu-se que algumas delas não eram tão funcionais quanto se esperava, e o conjunto de práticas foi modificado. Novas práticas foram criadas, e agora as práticas são divididas em práticas primárias e práticas corolárias. Nas próximas subseções elas são explanadas com mais detalhes.

A Figura 2.1 mostra as práticas iniciais de XP e as práticas que existem agora. As linhas ilustram o mapeamento entre as práticas novas e as práticas antigas.

Não existe uma ordem exata na qual as práticas devem ser aplicadas (exceto no caso das práticas corolárias, como veremos abaixo). Quantas e quais práticas são utilizadas dependem da ocasião e da necessidade do time. A adoção pode e deve ser feita incrementalmente. As práticas foram desenvolvidas para influenciar positivamente umas as outras. À medida que mais praticas vão sendo adotadas, o resultado de práticas que já estavam sendo utilizadas tende a melhorar também.

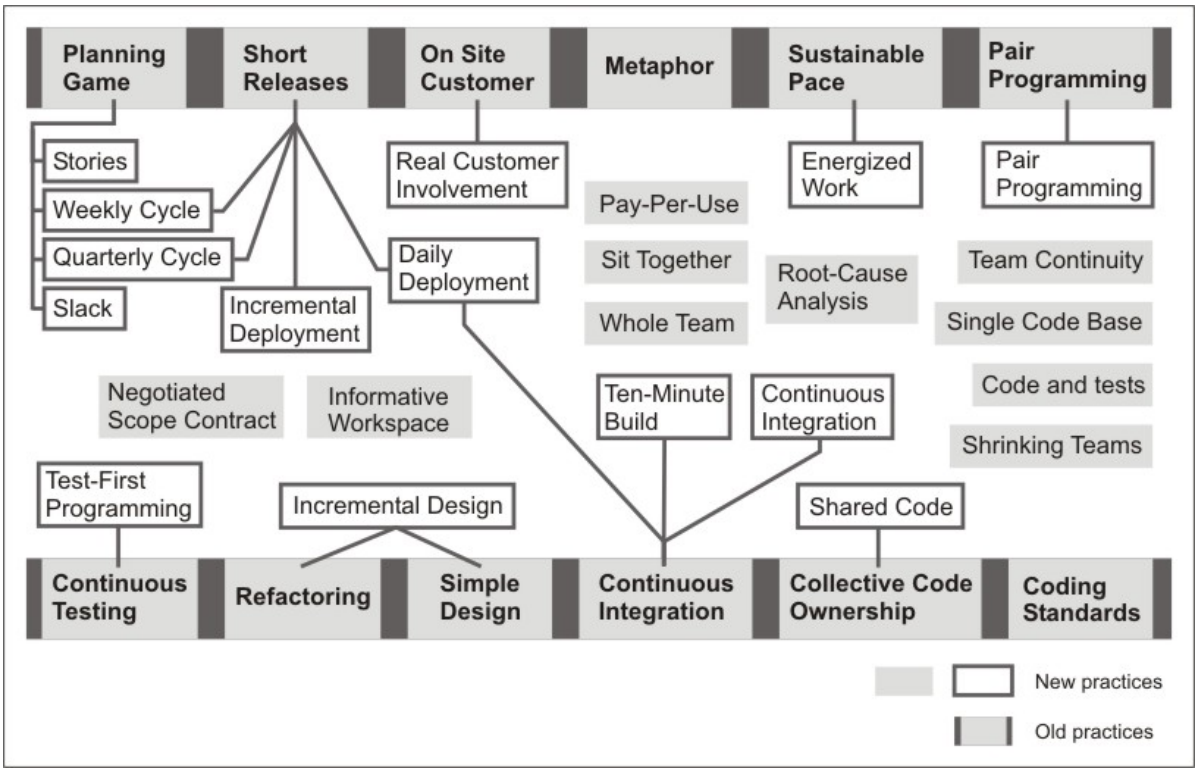

Figura 2.1 Mapeamento entre práticas antigas e novas

2.4.1 Práticas Primárias

As práticas primárias têm a característica de poder ser adotadas em qualquer ordem, sem nenhum pré-requisito. São elas:

Sit Together: deve existir um espaço específico para acomodar toda a equipe de desenvolvimento. A comunicação é extremamente favorecida se for possível conversar com qualquer um ou até todos os integrantes do projeto a qualquer momento. A idéia não é ficar num lugar apertado e desconfortável: o ambiente deve ser grande para acomodar todo o time e dar a cada um seu próprio espaço. Organize-o para facilitar as interações entre os integrantes do time. É importante que os espaços individuais de cada um sejam próximos uns aos outros. Se não há um espaço onde todos possam sentar juntos o tempo todo, procure uma sala de reuniões ou algum espaço disponível onde os integrantes possam ao menos passar algum tempo juntos, para resolver o que eles precisam. Isso inclui tanto realizar reuniões quanto programar em conjunto, discutir arquitetura, etc.

Whole Team: inclua no time pessoas que possuem características e habilidades necessárias para o sucesso do projeto. Mais do que isso, inclua pessoas cujas habilidades se completem. Além disso, todos devem sentir que fazem parte de uma mesma equipe, e estar dispostos a interagir e colaborar bastante uns com os outros.

Informative Workspace: faça com que sua área de trabalho reflita o andamento do seu projeto. Decore-a com informações sobre o projeto, pôsteres e murais listando as estórias que já foram concluídas, as que estão sendo feitas atualmente e as que faltam. Um visitante deve conseguir ter uma idéia do andamento do projeto só de olhar para o ambiente de trabalho.

Energized Work: os desenvolvedores precisam estar sempre descansados, para que possam focar no seu trabalho e ser produtivos. Para isso, é importante limitar a possibilidade de horas extras, para que cada um tenha tempo de viver sua vida pessoal em casa, e não no trabalho.

Pair Programming: todo o código do sistema deve ser escrito por dois programadores simultaneamente. Isso faz com que o foco e a produtividade seja maior e menos defeitos ocorram. Realizando rodízio de pares, é possível também melhorar a comunicação e o relacionamento entre os integrantes do time, bem como o conhecimento dos integrantes sobre diversas áreas do sistema.

Stories: as funcionalidades do sistema são descritas utilizando pequenas descrições de funcionalidades tangíveis para o usuário. As estórias são utilizadas para guiar o desenvolvimento do sistema, e boa parte do planejamento do projeito é feito em cima delas. O usuário deve escrever as estórias, uma vez que ele é que sabe exatamente o que quer do sistema. Vale a pena mencionar que as estórias servem para representar tanto requisitos funcionais quanto os chamados requisitos não funcionais, sem fazer distinção entre eles.

Weekly Cycle: o desenvolvimento de software é realizado em ciclos pequenos, de uma semana. No início de cada semana seu planejamento é realizado numa reunião com o cliente onde se decide que estórias serão implementadas. O cliente deve se reunir com a equipe para que todos entendam exatamente o significado de cada estória e possa estimar a dificuldade e o tempo levado para implementar cada uma. A equipe deve informar ao cliente o tempo disponível que ele tem para a implementação de estórias naquela semana. Por exemplo, se há dois pares de programadores no time trabalhando 8 horas por dia, ele pode escolher 16 horas de estórias para serem implementadas naquela semana. Baseado nesse tempo e nas estimativas, o cliente irá escolher que estórias ele quer que sejam implementadas primeiro, optando por aquelas que trarão valor de negócio mais rápido. À medida que o tempo passa, as estimativas dos programadores vão se tornando mais precisas, e o planejamento vai sendo atendido com mais conformidade. O ciclo semanal é que dita o ritmo do andamento do projeto, e dependendo da complexidade das estórias e do próprio projeto, pode ser estendido para levar mais de uma semana.

Quarterly Cycle: o desenvolvimento deve possuir ciclos maiores do que os semanais. Com ciclos trimestrais, um planejamento um pouco mais abrangente é realizado. Nestas ocasiões acontece um balanço sobre o time, o projeto e o progresso obtido até então.

Slack: não se deve fazer promessas que não podem ser cumpridas. Em qualquer plano, defina tarefas que podem ser descartadas caso o cronograma não consiga ser acompanhado.

Dessa forma é possível manter uma margem de segurança que pode ser utilizada para se proteger de problemas inesperados.

Ten-Minute Build: o sistema deve poder ser compilado, e executado, assim como todos os seus testes em menos de dez minutos. Essa restrição precisa ser atendida para que o sistema seja executado com muita frequência. Quanto mais tempo os testes levarem para rodar e o sistema levar para ser gerado, menos vezes ele será executado no total, e é de suma importância que o sistema seja executado várias vezes para que possa haver *feedback* sobre ele.

Continuous Integration: as mudanças no software devem ser integradas e testadas várias vezes durante o dia para evitar problemas com integração no fim. Isso deve ser feito no máximo a cada duas horas. Quanto mais tempo se espera para realizar a integração maior a possibilidade de ocorrer problemas. Realizando integração frequentemente os problemas que podem ocorrer serão sempre pequenos e podem ser mantidos sob controle.

Test-First Programming: antes de escrever e atualizar código se escrevem testes para verificar o mesmo. Esta prática serve para evitar os problemas abaixo:

- Foco no escopo: é fácil se deixar levar e programar rapidamente, escrevendo tudo que você pensa direto no código. Ao escrever os testes antes, o programador já pensa no que pode dar errado de antemão. Isso faz com que ele preste mais atenção no problema e escreva o código com mais cuidado.
- Acoplamento e Coesão: se há dificuldade em escrever um teste que foque numa parte específica do programa, então existe algum problema de design. Num código fracamente acoplado e altamente coeso, os testes devem ser fáceis de escrever. Escrever os testes antes favorece estas características.
- Confiança: escrever código de confiança, e documentar esse código com testes automatizados é um comportamento que adquire a confiança dos companheiros.
- Ritmo: após adquirir costume com a prática, os programadores desenvolvem um ritmo saudável para o desenvolvimento: testar, codificar, refatorar, testar, e assim sucessivamente.

Incremental Design: ao invés de realizar todo o trabalho de design no início do projeto, realizar o design incremental. Produzir código assim que possível. Desta forma, XP consegue *feedback* rápido e pode implementar melhorias cedo. XP não deixa de realizar design, mas faz só o necessário. O design do sistema é trabalhado diariamente, baseado nas estórias que serão implementadas naquele período e no que já existe até então. Posteriormente o design será refeito, mas só quando necessário. A idéia por trás disso é que como os requisitos mudam bastante, o sistema pode acomodar novos requisitos a qualquer momento. Além disso, não se perde tempo tendo baseado o design do sistema em requisitos que podem ser descartados antes de serem implementados.

2.4.2 Práticas Corolárias

As práticas corolárias são um pouco mais complexas que as primárias. Elas têm o pré-requisito de só poder aplicadas após todas as práticas primárias que serão necessárias já estiverem em uso. Isso porque elas podem gerar alguns efeitos colaterais indesejados caso as práticas primárias não estejam sob controle. Por exemplo, gerar versões diariamente antes de reduzir substancialmente o número de defeitos (através de práticas primárias como programação em pares, integração contínua, etc.) vai trazer mais problemas do que utilidade. Não é o foco deste trabalho explicar XP em todos os seus detalhes. Como as práticas primárias são o que definem se uma equipe está ou não seguindo a metodologia, as práticas corolárias serão apenas mencionadas. São elas: Real Customer Involvement, Incremental Deployment, Team Continuity, Shrinking Teams, Root-Cause Analysis, Shared Code, Code and Tests, Single Code Base, Daily Deployment, Negociated Scope Contract and Pay-Per-Use.

2.5 Papéis

Os papéis em XP não são rígidos nem fixos. O objetivo é fazer com que todos possam contribuir com o máximo que tiverem para oferecer para o sucesso do time. No início, papéis fixos podem ajudar no aprendizado de novos hábitos, com os papéis técnicos tomando as decisões técnicas e os papéis de negócio tomando as decisões de negócio. Uma vez que os relacionamentos entre os integrantes do time estiverem sedimentados, e que haja respeito entre eles, os papéis fixos começam a interferir no objetivo de que cada um faça o melhor que puder para ajudar o projeto. Programadores podem escrever estórias se eles estiverem na melhor posição para escrever a estória. Gerentes de projeto podem sugerir melhoras na arquitetura se estiverem na melhor posição para isso.

Dizer que os papéis podem contribuir para um time XP não significa que existe apenas um mapeamento de uma pessoa pra um papel. Um programador pode ter um pouco de arquiteto. Um usuário pode se tornar um gerente de projeto. Os papéis podem ter diferentes subdivisões de acordo com a equipe, se isso for adequado para o projeto. O objetivo principal não é preencher papéis abstratos com pessoas, e sim fazer com que as pessoas contribuam com aquilo que sabem melhor para o projeto.

2.5.1 Programador

O programador é responsável pela tarefa mais importante da metodologia, ele programa. No entanto, existe um diferencial importantíssimo na programação realizada em XP: ela é fortemente guiada via *feedback*. O programador não programa somente, ele precisa programar e estar recebendo sempre recebendo *feedback* do usuário do sistema, para realizar correções no que for necessário, abandonar algo que não era como o cliente esperava, ou simplesmente seguir em frente e desenvolver outras estórias.

No início de cada ciclo semanal (normalmente uma semana) o programador precisa se reunir com o gerente e os clientes, para entender o que precisa ser feito no sistema. Nessa reunião, onde o cliente irá explicar as funcionalidades desejadas para o sistema através de

estórias, o programador deve perguntar tudo que não entender, e ter certeza que sua visão sobre as estórias é a mesma do cliente e do gerente. É necessário não só entender o que o cliente quer que seja feito, mas é importante definir critérios para determinar se a implementação de uma estória foi bem sucedida. Estes critérios são chamados de *Testes de Aceitação*, e devem ser feitos nesta mesma reunião onde as estórias são criadas. Esse conjunto de atividades é conhecido como o *Jogo do Planejamento*.

Depois dessa fase inicial, o programador pode implementar as estórias, utilizando *Test Driven Development*: primeiro ele pensa nos problemas possíveis que aquelas estórias podem ter, e escreve testes unitários para representar estes problemas. Só então ele começa a escrever as estórias em si. A idéia principal é que ele escreva testes para capturar todas as situações de problema possíveis que uma estória pode ter, e na hora de escrever a estória, faça ela o mais simples possível, desde que ela consiga passar por todos os testes. O programador também realiza o projeto da arquitetura do sistema, mas somente do que existe até agora, modificando o que for necessário para acomodar a nova estória. Esta arquitetura vai sendo alterada gradativamente, à medida que mais estórias vão sendo implementadas. Uma vez que as estórias são implementadas, os testes de aplicação são feitos para verificar a corretude das mesmas. Depois, elas precisam ser disponibilizadas para que o cliente as teste e possa dar *feedback* de como elas estão. Se ele estiver satisfeito, as estórias ficam como estão. Se não, novas estórias são escritas para corrigir as antigas, e elas entram na pilha de estórias a ser implementadas novamente.

2.5.2 Cliente

O cliente em XP é um integrante da equipe, e deve estar presente sempre que possível tanto parar tirar dúvidas sobre o que deve ser feito quanto para validar o que já foi feito até então. Existe uma importante subdivisão desse papel de cliente que deve ser levada em consideração, e está descrita logo abaixo.

***Goal Donnor*:** é o usuário do sistema. Ele é o possivelmente o maior conhecedor do domínio da aplicação e das funcionalidades que o sistema deverá implementar. Ele determina o que é útil para a aplicação do ponto de vista do usuário, escolhendo novas funcionalidades, validando e ajudando a corrigir antigas. O objetivo principal deste papel é melhorar o software do ponto de vista do usuário.

***Gold Owner*:** é quem banca o sistema. Ele é responsável por validar decisões tomadas no projeto que tenham impacto financeiro. Determina o que tem valor de negócio e o que não tem. Pode vetar decisões de projeto que ele ache que não irão trazer retorno satisfatório. O objetivo principal deste papel é maximizar o retorno sobre o investimento feito.

Esta diferença precisa ser entendida, porque apesar de ambos serem clientes, os seus interesses vão divergir com bastante freqüência. Por exemplo, para o *Goal Donnor* sempre vão existir melhorias a serem feitas no projeto. No entanto, para o *Gold Owner*, existe um período onde o investimento no projeto pode estar sendo bem recompensado, mas a probabilidade de que à medida que o tempo passe esse retorno diminua é grande. O *Gold Owner* deve ser capaz de reconhecer este momento numa situação como esta e parar a execução do projeto. A Figura 2.2 ilustra essa situação.

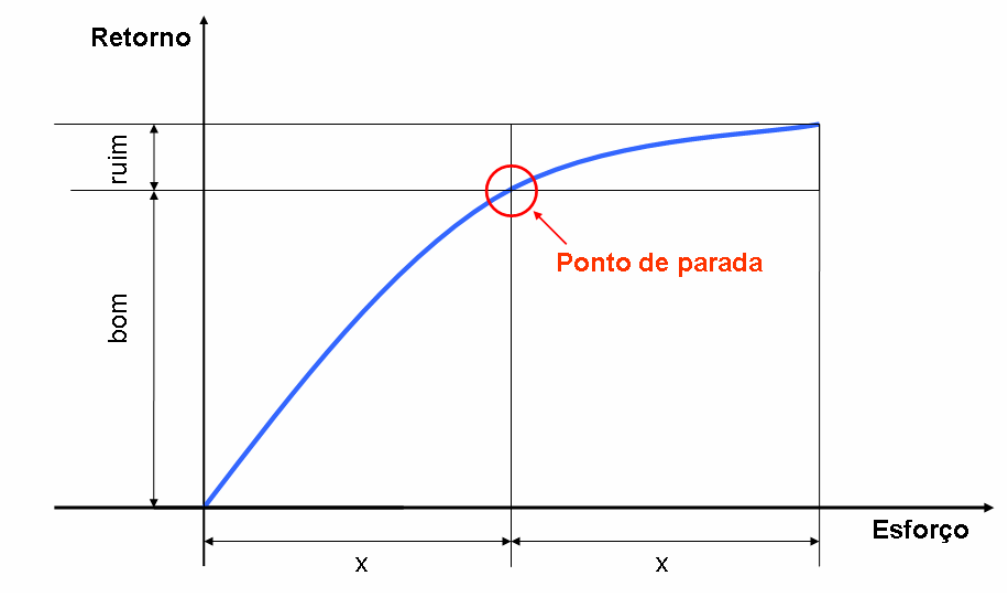

Figura 2.2 Gráfico que ilustra diminuição do retorno sobre esforço

Como mencionado acima, o cliente tem participação ativa no projeto. Inicialmente, o *Goal Donnor* deve apresentar a sua idéia de projeto para o *Gold Owner*, que deve entender exatamente o que é, e uma vez que julgue que o investimento vale a pena, dar o aval para o início do projeto. Depois que o projeto se inicia, o papel do *Gold Owner* é acompanhar o retorno que está sendo gerado pelo mesmo, e validar decisões do *Goal Donnor* relativas a escopo e funcionalidades. O papel do *Gold Owner* não é modificar funcionalidades sugeridas pelo *Goal Donnor*, uma vez que ele não necessariamente possui um conhecimento profundo sobre o domínio da aplicação. No entanto ele deve participar na escolha de que funcionalidades devem ou não devem ser implementadas de fato, e no caso das que devem, determinar quando isso deve acontecer, de modo que as que tragam mais valor de negócio mais rápido sejam implementadas primeiro.

O *Goal Donnor* por outro lado, deve se reunir com o Gerente e os programadores, a fim de tornar sua visão da aplicação e suas necessidades com relação à mesma claras para toda a equipe. É importante que ele esteja presente no local de desenvolvimento se possível, porque dúvidas a respeito do projeto podem surgir o tempo todo, e ele é a pessoa mais apropriada - e muitas vezes o único com conhecimento - para resolvê-las. No jogo de planejamento, descreve as estórias e os testes de aceitação. Depois, quando as estórias são implementadas, ele deve utilizar o programa desenvolvido (se possível até na situação real para o qual este está sendo feito) a fim de verificar se o mesmo tem algum problema que não foi imaginado até então. Se houver algum problema, ele deve escrever uma nova estória na próxima sessão de planejamento reportando o problema, para que ele possa ser corrigido.

2.5.3 Gerente

O gerente tem a função de ligar todos os componentes do time, e fazer o processo andar de forma suave. O gerente não faz o projeto: ele faz o projeto andar. Ele deve ser o facilitador geral, tirando empecilhos da frente das pessoas para que elas possam realizar seu trabalho corretamente. O gerente coordena as atividades dos outros papéis, fazendo o que for necessário para manter o ritmo de trabalho sempre bom.

No jogo do planejamento, o gerente interage com o cliente e os desenvolvedores, para o esclarescimento geral das estórias e seus testes de aceitação. Ele pode inclusive ajudar na escrita destes elementos se for preciso. Ele pode auxiliar o *Goal Donnor* na definição de um escopo para aquela semana, e discutir estes objetivos com o *Gold Owner* para validar esta definição. O gerente deve organizar a infra-estrutura necessária para o acontecimento do projeto, bem como providenciar quaisquer outras necessidades (físicas ou não) para o time. Ele deve garantir que a comunicação aconteça no projeto, mas sem interferir. O gerente não deve ser o meio de comunicação entre clientes e programadores, por exemplo. Ele deve indicar quem deve falar com quem, e deixar a comunicação a cargo deles. Ele também deve discutir com o tracker e o coach o andamento do projeto e adequação à metodologia, procurando soluções para eventuais problemas. O gerente é a figura procurada quando se tem um problema: ele deve procurar resolver conflitos entre integrantes da equipe, questões de horário, de infra-estrutura, de insatisfação dos integrantes, entre outras coisas.

No entanto, é importante ter em mente que o gerente não deve tentar forçar as coisas acontecerem: ele deve facilitá-las. Um gerente nunca deve, por exemplo forçar um desenvolvedor a realizar estimativas que ele acha que não consegue cumprir. Esta decisão cabe apenas ao desenvolvedor. As obrigações do gerente também incluem saber seu papel, e saber deixar os outros desempenharem os seus. É óbvio que ele também deve ser capaz de reconhecer quando alguém não está correspondendo às expectativas: neste caso, ele deve procurar saber o porquê disso. Se for um problema da pessoa, ele deve conversar com ela a fim de repreender e tentar melhorar seu comportamento. Pode ser que seja um problema das expectativas, que podem estar muito irreais. Cabe ao gerente refletir sobre porque se espera aquilo, e adequar a expectativa à metas mais reais. O objetivo principal do gerente é encontrar pontos falhos no projeto, e providenciar para que eles sejam corrigidos, para que o projeto esteja sempre produzindo o melhor possível.

Ele deve também saber recompensar o bom desempenho da equipe quando apropriado. É difícil que não haja nenhum problema em qualquer ponto do andamento do projeto. Mesmo assim, há pode ser que a equipe esteja trabalhando muito bem dentro de suas condições, e os méritos precisam ser atribuídos.

2.5.4 Tracker

O *tracker* deve ser capaz de mensurar e comunicar o andamento do projeto através de dados. Ele determina e faz uso de métricas para acompanhar diversas características do projeto e da metodologia, verificando por exemplo a cobertura dos testes, a velocidade da equipe, número de bugs, entre outros. Ele deve organizar estes dados de uma forma que seja fácil para que o gerente compreenda, gerando estatísticas, gráficos, e relatórios se for o caso. Não existe um conjunto de métricas pré-determinado; o *tracker* deve coletar as métricas que puder e achar

necessárias, dependendo do contexto do projeto.

O trabalho do tracker se espalha durante todo o andamento do projeto. Dependendo das métricas adotadas, ele pode coletá-las com freqüências diferentes. Ele pode por exemplo, todo dia verificar quantos testes unitários existem, e quantos estão passando no programa. Pode também acompanhar o crescimento de testes unitários ao fim de cada iteração. Pode verificar quantas estórias em médias são completadas por iteração, qual a média de estórias que estão voltando do sistema implantado para ser implementadas novamente pois não estão fazendo o que deveriam, e assim por diante. O tracker provê ao gerente mais *background* sobre o desempenho da equipe, e em conjuto com ele e com o *Coach* pode tomar decisões mais embasadas e refinar previsões para que estas fiquem mais realistas.

2.5.5 Coach

O *coach* é um dos papeis mais determinantes para o sucesso de um projeto XP. Sua função é verificar a aderência da equipe à metodologia, e garantir que todos estejam seguindo-a durante todo o andamento do projeto. Ele deve interagir com todos os integrantes do time durante todo o andamento do projeto, para torná-los cientes de suas obrigações e impedir omissão por parte de qualquer um. O coach precisa saber se impor para comunicar a qualquer integrante (independente de seu papel) caso seu comportamento não esteja de acordo com o que deveria. Para isso ele deve ser fluente na metodologia, tanto para saber em que situações há problemas quanto para sugerir comportamentos alternativos, uma vez que nuances da metodologia variam de projeto para projeto. Seu principal objetivo é fazer com que o time absorva e acredite na metodologia.

2.6 Considerações Finais

Este capítulo apresentou uma visão geral sobre a XP. A metodologia prega um conjunto de cinco valores, quatorze princípios e vinte e quatro práticas. Os valores são a filosofia por trás da metodologia, e representam objetivos gerais que devem estar presentes em qualquer projeto. As práticas são atividades que falam mais detalhadamente o que deve ser feito na metodologia. Uma prática sem um valor que a justifique não é útil para o projeto. No entanto, os valores são bastante abstratos, e é possível encontrar várias práticas que estejam de acordo com cada um deles. Por isso existem os princípios, que são características que devem ser observadas ao tentar aplicar uma prática para atender a um valor. Os princípios são a ponte entre os valores e as práticas. Além disto apresentamos os papéis responsáveis por empregar estes valores, princípios e práticas, caracterizando assim um projeto XP. XP é uma metodologia ágil, ou seja, que preza por rapidez na resposta à mudanças e forte interação com o cliente.

CAPÍTULO 3

MSF for Agile Software Development

O *Microsoft Solutions Framework* surgiu em 1994 como um conjunto de boa práticas compiladas pela Microsoft a partir de sua experiência na produção de software e em serviços de consultoria. Desde então, o MSF evoluiu, tornando-se um *framework* flexível para nortear o desenvolvimento de projetos de software. Essa evolução também teve o intuito de acompanhar tendências bem sucedidas no meio da engenharia de software, como a capacidade de responder à necessidade de mudanças rapidamente. O MSF 4.0 foi publicado em duas linhas diferentes: uma mais tradicional, o *MSF for CMMI Process Improvement*, e uma ágil, o *MSF for Agile Software Development*. Essa última será nosso objeto de estudo neste capítulo, que descreverá suas principais características.

3.1 Visão Geral

Por mais semelhanças que possa ter em relação a outros, cada projeto de software é único devido a uma série de fatores que incluem desde o tipo de aplicação a ser construída, necessidades específicas de cada cliente e até restrições de cronograma e recursos. Já existe, desde o MSF 3.0, a consciência por parte de seus criadores de que é impossível definir um único processo de desenvolvimento de software que seja adaptável a qualquer situação. Foi devido a isso que o MSF 4.0 foi lançado em duas versões diferentes, uma que aborda processos ágeis e outra que aborda processos tradicionais. No entanto, mesmo assim, ainda existe muito em comum entre as duas versões. À medida que as duas forem amadurecendo, a tendência é que a diferença entre elas se torne mais notável.

O MSF 3.0 era composto por quatro elementos básicos: Princípios Fundamentais, Modelo de Processo, Modelo de Equipe e Disciplinas. A análise do MSF 4.0 mostra que este continuou bastante semelhante ao seu predecessor em certos aspectos. Estes incluem os princípios fundamentais, a macroestrutura do Modelo de Processo e a maioria das boas práticas. As maiores modificações ocorreram com a concretização do modelo de equipe e das boas práticas do framework, através do estabelecimento de fluxos de trabalho (*workstreams*). A seguir, os elementos básicos do MSF Ágil são apresentados em mais detalhes.

3.1.1 Mindsets

O MSF Ágil é muito mais do que um conjunto de atividades a serem seguidas. Ele busca a criação de uma cultura que encorage projetos bem sucedidos. Um *mindset* é uma coleção de

valores que determina como os indivíduos irão interpretar e reagir às situações. *Mindsets* devem estar na mente de cada membro da equipe desde o início do projeto até o seu final quando da tomada de decisões, priorização de trabalho, representação do seu grupo de papéis e interação dos membros da equipe com outros participantes. Existem oito *mindsets* no MSF, que estão detalhados abaixo.

Qualidade é definida pelo cliente: clientes satisfeitos são a prioridade primordial de uma boa equipe. Esta satisfação inclui clientes internos e externos. Focar no cliente durante o desenvolvimento significa um compromisso da equipe em relação a compreensão e resolução dos problemas de negócio do mesmo. Uma vez que o problema de negógio foi compreendido, o envolvimento do cliente precisa ser maximizado até um grau que garanta o alinhamento das expectativas deste com o projeto. Técnicas que suportam o ajuste e gerenciamento das expectativas do cliente incluem: relatar as pendências do projeto, produzir *builds*[1] de tamanho pequeno e entregar produtos que atendam a padrões de qualidade.

Orgulho do trabalho individual: sentir orgulho em contribuir para uma solução é importante na criação de um produto de qualidade. Motivação e senso de resposabilidade são produtos diretos deste orgulho. Criar um orgulho focado nas habilidades individuais é uma responsabilidade tanto do indivíduo quanto da organização. Técnicas que ajudam a manter este senso de propriedade em relação ao projeto incluem: estimativas bottom-up, escolha de codinomes que identifiquem o projeto claramente, identificação clara do time, entre outros.

Equipe de pares: o *mindset* "equipe de pares"estabelece valor igual para cada grupo de papéis do modelo de equipe. É necessário que haja comunicação irrestrita entre os papéis, transparência e um conjunto único e visível de pendências. O resultado é um aumento na prestação de contas da equipe e uma comunicação efetiva.

Entrega Freqüente de Versões: nada consegue ser mais efetivo para estebelecer credibilidade do que uma entrega freqüente do produto. Ainda que ter um produto cada dia mais próximo de ser entregue em definitivo seja importante, responder às necessidades dos clientes com pequenas versões de qualidade mostrará progresso. Através da entrega de versões freqüentes, o processo e a infra-estrutura são provadas e melhoradas. Riscos, defeitos e requisitos não-detectados são percebidos mais cedo. Assim, *feedback* pode ser dado quando necessário. Alguns pontos chave para a entrega freqüente de versões são: manter o tamanho dos builds pequeno, trabalhar nas versões em uma maneira "just-in-time"e deixar opções em aberto através da eliminação de decisões prematuras. Entrega freqüente de versões é um *mindset* que estimula a disciplina em uma equipe de desenvolvimento de software.

Desejo de aprender: uma vez que todo projeto de desenvolvimento, ambiente e time é único, cada projeto e iteração cria uma oportunidade de aprendizado. Entretanto, não pode

[1] Um *build* é uma compilação de todos os arquivos, bibliotecas e componentes em um novo conjunto de arquivos executáveis.

haver aprendizado sem um *feedback* honesto e reflexão. A não ser que haja um ambiente que dê suporte e estimule a coragem e a segurança pessoal, o *feedback* será limitado e não contribuirá para o engrandecimento. Postos estes fatores, indivíduos e equipes podem focar em melhorias pessoais, coletando e compartilhando o conhecimento e as lições aprendidas. Adicionalmente, existirão oportunidades para implementar práticas comprovadas por outros e reservar tempo do cronograma para a aprendizagem. Técnicas para gerar o *feedback* necessário para o princípo do desejo de aprender incluem revisão por companheiros e análises retrospectivas.

Tornar-se específico cedo: muitos projetos perdem tempo procrastinando em busca de soluções genéricas ao invés de lidar com problemas solucionáveis. Este *mindset* ressalta a necessidade de dar um passo de cada vez, aprendendo do específico ao invés do abstrato. Definir o projeto em termos cotidianos é a chave para construção efetiva de código que funcione, satisfazer os testes e criar um produto pronto para ser entregue. Outra área que pode aprender do específico é a implantação. Técnicas que suportem este *mindset* incluem personas[2], cenários, projeto de implantação e casos de testes.

Qualidade de Serviço: o *mindset* qualidade de serviço observa a solução e desenvolve planos baseados em cada aspecto da experiência do cliente. A idéia é que qualidades de serviço como performance e segurança não devem ser consideradas tardiamente no projeto, mas através dele como um todo. Quando ignorados, estas qualidades de serviço são geradoras de insatisfação dos clientes e são usualmente fruto de considerações implícitas sobre como a solução vai se comportar. Com este *mindset*, a solução é contemplada como um todo e considerações implícitas se transformam em requisitos de qualidade de serviço. Técnicas que suportam este *mindset* incluem usar o conhecimento de um especialista quando se necessita e descobrir riscos o mais cedo possível.

Cidadania: o *mindset* cidadania foca no controle de recursos do projeto, corporativos e computacionais. A cidadania se manifesta de várias formas, desde a condução do projeto de uma maneira eficiente até a otimização do uso de web services. Técnicas que suportam o *mindset* cidadania incluem: realocação de defeitos com toda a informação necessária para que outra pessoa possa começar bem e prover boas estimativas da quantidade de trabalho que uma tarefa de desenvolvimento ou de teste irá tomar.

3.1.2 Princípios

Parceria com clientes Validação pelo cliente é comumente a diferença entre valor de negócio real e fictício. Entender a proposta de valor da sua solução e comunicá-la efetivamente é um fator chave de sucesso.

Encorajar comunicação aberta Para maximizar a efetividade individual dos integrantes, a informação precisa estar prontamente disponível ser ativamente compartilhada no time.

[2]Uma persona é a descrição das capacidade necessidades, desejos, hábitos de trabalho, tarefas, e *background* de um conjunto particular de usuários. Uma persona é a coleção de dados que descreve características importantes sobre a interação de um grupo particular de usuários com o sistema, condensados em um papel fictício.

Trabalho em direção a uma visão compartilhada Visão compartilhada garante que todos os membros do time enxergam os objetivos que eles pretendem alcançar com o projeto sob uma mesma ótica. A colaboração é aprimorada, porque as decisões do projeto não são tomadas para atender as vontades de um indivíduo e sim para atender a um objetivo que é visto e aceito por todos.

Qualidade é trabalho de todos, todo dia Qualidade requer tanto prevenção de *bugs* quanto verificação de soluções. Análise de código e revisões em pares são utilizadas para realizar estas duas tarefas. Todos os papéis são responsável por prevenção e verificação de bugs.

Manter-se ágil, adaptar-se a mudança Quanto mais uma organização procura maximizar o impacto no negócio de um investimento em tecnologia, mais ela adentra novos territórios. Este novo território é inerentemente incerto e sujeito a mudança, uma vez que exploração e experimentação resultam em novas necessidades e métodos. Exigir certeza num ambiente em processo de mudança e um objetivo não realista, e leva a resultados falhos.

Faça da implantação um hábito O time deve se comprometer a estar criando um produto de excelência, mesmo enquanto realiza mudanças nele. Cada mudança deve ser feita sob a convicção de que oproduto deve estar pronto para ser implantado a qualquer hora. Projete o sistema para ser implantado, e pratique sua implantação num ambiente de desenvolvimento. Desenvolvedores devem estar constantemente executando o produto, e o cliente deve estar sempre testando ou liberando as novas versões.

Fluxo de valor Planejamento, execução e medição do progresso e velocidade devem ser baseados na entrega de valor de negócio sempre crescente para o cliente, e de um retorno de investimento também crescente. Atividades que não agregam valor de negócio devem ser minimizadas, pois são desperdício. Iterações devem ser usadas para manter uma cadência de produtos atualizados que o cliente possa estar sempre avaliando.

3.1.3 Modelo de Equipe

O *Modelo de Equipe* do MSF [12] tem por objetivo definir os papéis e responsabilidades das pessoas que estarão envolvidas em um projeto. Uma característica do referido modelo é que o mesmo prega a formação de uma *equipe de pares*, isto é, não com uma organização hierárquica *top-down* tradicional, mas um modelo para uma equipe colaborativa com responsabilidades compartilhadas. Subjacente ao modelo está o conceito de que qualquer projeto de software, para ser considerado bem sucedido, deve satisfazer a alguns objetivos chave de qualidade. Assim, o Modelo de Equipe do MSF define cada um de seus papéis baseado na necessidade de adequação a um objetivo de qualidade (Tabela 3.1). Para atingir tais objetivos, cada grupo de papéis, que são denominados pela metodologia de grupos de defesa de interesses[3], deve atuar em um conjunto de *áreas funcionais* com *responsabilidades* associadas.

Além dos grupos de defesa, o Modelo de Equipe também prescreve uma maneira de se adequar à escala necessária para um projeto específico. Integrantes podem representar mais de

[3]E serão chamados aqui somente de grupos de defesa, porque questões de simplicidade.

Objetivo	*Grupo de Defesa Responsável*
Entregar a solução dentro das restrições do projeto	Gerência de Programa
Defender interesses do sistema numa visão mais ampla	Arquitetura
Construir a solução obedecendo às especificações	Desenvolvimento
Defender a qualidade da solução pela perspectiva do cliente	Teste
Implantar a solução sem maiores impactos, na infra-estrutura apropriada	Gerência de *Release*
Maximizar a performance do usuário	Experiência do Usuário
Defender os interesses do negócio dos clientes	Gerência de Produto

Tabela 3.1 Grupos de papéis do MSF e objetivos chave dequalidade

um grupo (acumulando papéis), ou um papel pode ser representado por mais de uma pessoa para atender a projetos de tamanhos pequenos e grandes, respectivamente. Existem no MSF Ágil sete grupos de defesa. Eles estão ilustrados na Figura 3.1, que mostra a relação entre cada grupo e os papéis associados.

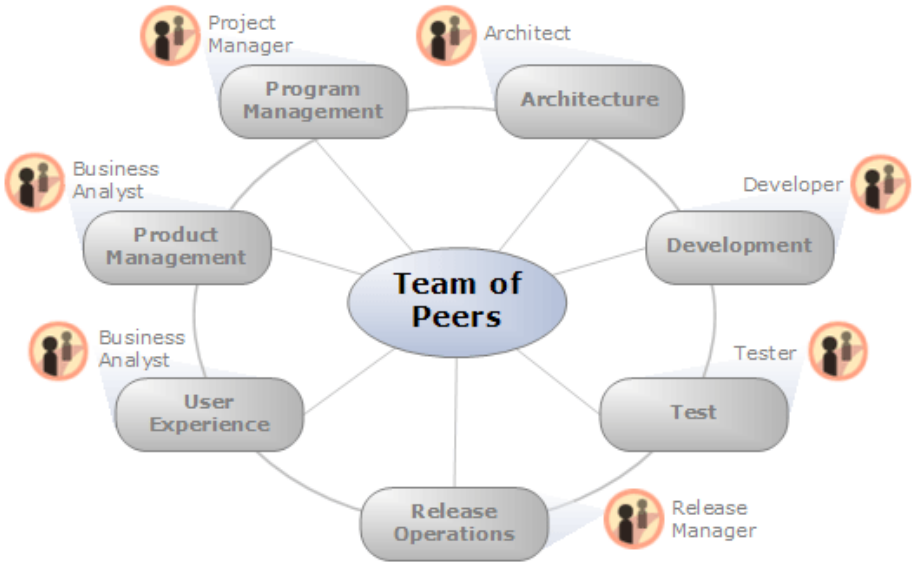

Figura 3.1 Grupos e papéis no MSF Ágil

As próximas subseções apresentam uma idéia geral das responsabilidades de cada papel do Modelo de Equipe.

3.1.3.1 Gerência de programa

O foco da gerência de programa é atender ao objetivo de concluir e entregar o projeto dentro dos prazos e custos estimados, gerando valor de negócio para o cliente. Este grupo também

deve garantir que a solução certa é entregue ao tempo certo, e que todas as expectativas de todos os stakeholdes sejam entendidas, gerenciadas e atendidas durante o projeto.

Gerente de projeto: O principal objetivo do gerente de projeto é garantir a entrega de valor de negócio dentro do cronograma e orçamento acertados. O gerente de projeto é encarregado dos trabalhos de planejamento e definição de cronograma, incluindo o desenvolvimento do projeto e planos de iteração; monitoramento e relato do estado do projeto; e a identificação e mitigação dos riscos. Ele também deve consultar os analistas de negócio a fim de planejar cenários e requisitos de qualidade de serviço para cada iteração, os arquitetos e desenvolvedores para estimar o trabalho, e os analistas de teste a fim de planejar os testes. Além disso, o gerente deve facilitar a comunicação interna da equipe.

3.1.3.2 Arquitetura

A arquitetura defende os interesses do sistema em uma visão mais ampla. Isso inclui os servicos técnicos e não-técnicos com os quais a solução irá interoperar, a infra-estrutura onde ela será implantada, seu lugar no negócio ou na família de produtos e seu plano de versões futuras. O grupo da arquitetura precisa garantir que a solução implantada irá atender a todas as qualidades de serviço[4] e objetivos de negócio, e ainda ser viável a longo prazo.

Arquiteto: O arquiteto é responsável por garantir o sucesso do projeto através do projeto dos aspectos fundamentais da aplicação. Isto inclui definir a estrutura organizacional da aplicação e a estrutura física de sua implantação. Nestas tarefas, o objetivo do arquiteto é reduzir a complexidade através da divisão do sistema em partições claras e simples. A arquitetura resultante é extremamente importante porque não apenas dita como o sistema será construído, como também estabelece se a aplicação conterá as muitas características que são essenciais para determinar um projeto bem sucedido. Estas incluem: usabilidade; se o projeto é confiável e apresenta bom grau de manutenção; se ele está em conformidade com padrões de segurança e performance; e se ele pode evoluir facilmente quando confrontado com mudança nos requisitos.

3.1.3.3 Desenvolvimento

O Desenvolvimento defende a solução técnica. Além de ser responsável pela criação primária de soluções, o desenvolvimento também é encarregado de decisões técnicas importantes, um design limpo, boas estimativas, código de qualidade e de fácil manutenção e testes unitários.

Desenvolvedor: Este papel tem como principal objetivo a implementatação da aplicação conforme sua especificação e dentro do tempo estimado. Do desenvolvedor também se espera ajuda na especificação de características do projeto físico, estimação de tempo e esforço para a finalização de cada característica, construção ou supervisão da implementação das características, preparação do produto para a implantação e atuação como consultor de tecnologia para o restante da equipe.

[4]Termo adotado pelo MSF para denotar uma restrição do sistema tal como performance, carga, *stress*, segurança ou plataforma. Estes requisitos não descrevem funcionalidade em si, e sim restrições sobre funcionalidades. É o elemento equivalente a os conhecidos requisitos não-funcionais do sistema.

3.1.3.4 Teste

Este grupo prevê, procura e reporta qualquer problema que pode diminuir a qualidade da solução aos olhos dos clientes e usuários.

Analista de testes: O principal objetivo do analista de testes é a descoberta e relato de problemas que afetem o produto e que possam vir a prejudicar o calor de negócio deste. O analista precisa entender o contexto do projeto, e ajudar os outros a tomar decisões informadas, baseadas neste contexto. Um objetivo chave do analista de testes é a descoberta e relato de defeitos significativos no produto através da execução de testes sobre o mesmo. Uma vez que um defeito foi encontrado, é trabalho do analista de testes comunicar o seu impacto de uma maneira precisa e descrever soluções que diminuam o mesmo. O analista de testes produz a descrição dos defeitos e os passos para a reprodução destes. Juntamente com o time, o analista de testes determina os padrões de qualidade para o produto. O proposito das atividades de testes é provar que as funcionalidades conhecidas funcionam corretamente e descobrir novas ocorrências em relação ao produto.

3.1.3.5 Operações de *release*

Esse grupo garante a disponibilidade a tempo e a compatibilidade da infra-estrutura para a solução.

Gerente de *release*: Garante que o produto está pronto para entrar na fase de implantação e gerencia a mesma. Para isto, cria planos e cronogramas e certifica os candidatos a versão final.

3.1.3.6 Experiência do usuário

Este grupo de defesa deve entender o contexto dos usuários como um todo, incluindo toda e qualquer sutileza de suas necessidades, e além disso garantir que toda a equipe está consciente da usabilidade de acordo com sua visão.

Analista de negócio: O principal objetivo deste papel é definir uma oportunidade de negócio e a aplicação que servirá a esta. O analista de negócio trabalha com os clientes e outros participantes a fim de entender suas necessidades e objetivos, e traduz estas em definições de personas, cenários e requisitos de qualidade de serviço que a equipe de desenvolvimento usará para construir a aplicação. O analista de negócio provê "expertise"no domínio da aplicação à equipe. Além disso, o analista de negócio age como representante das áreas de *experiência do usuário* e *gerência do produto*, o que significa que ele sempre deve zelar pelos interesses dos usuários e dos patrocinadores dos projeto.

3.1.3.7 Gerência de produto

A gerência de produto tem o dever de entender, comunicar e garantir o sucesso do ponto de vista econômico do cliente requisitando a solução.

O único papel integrante deste grupo é o Analista de Negócio, que foi descrito na subseção anterior.

3.1.4 Escalando para projetos pequenos

Em projetos de pequeno porte, ou projetos de baixa complexidade, uma pessoa pode atuar em vários papéis. Quando se combina papéis, é importante preservar o equilíbrio provido pelos diferentes grupos de defesa de interesses. A Figura 3.2 apresenta uma relação entre os grupos de defesa, indicando quais podem ser acumulados.

	Architecture	Product Management	Program Management	Development	Test	User Experience	Release Management
Architecture		**N**	**P**	**P**	**U**	**U**	**U**
Product Management			**N**	**N**	**P**	**P**	**U**
Program Management				**N**	**U**	**U**	**P**
Development					**N**	**N**	**N**
Test						**P**	**P**
User Experience							**U**
Release Management							

P Possible **U** Unlikely **N** Not Recommended

Figura 3.2 Acúmulo de papeis no MSF Ágil

3.1.5 Fluxos e items de trabalho

Uma importante modificação no MSF Ágil (e no MSF 4.0 de maneira geral), foi a concretização da metodologia através do estabelecimento de fluxos de trabalho. Estes são grupos de atividades associadas a um ou mais papéis do Modelo de Equipe, cujo objetivo é guiar a execução de tarefas específicas durante a execução do projeto.

Por restrições de espaço, não será promovida uma análise detalhada de todos os fluxos de trabalho do MSF Ágil (um total de 14, cada um com várias atividades e sub-atividades). Estes podem ser encontrados em sua documentação [10]. Será analisado, entretanto, um de seus fluxos, para que possa ser melhor compreendida a estrutura geral dos mesmos. Não obstante, a Tabela 3.2 apresenta a relação que os fluxos de trabalho guardam com os papéis do MSF.

Cada fluxo de trabalho é constituído por: uma *visão geral*, que determina o principal objetivo a ser atingido pelo fluxo; *critérios de entrada*, que explicitam dependências em relação a outras atividades, assim como determinam quando o fluxo realizar-se-á; uma seqüência de *atividades* a serem executadas; *critérios de saída*, que deixam claras as pós-condições a serem satisfeitas após o término do fluxo; um conjunto de *papéis envolvidos* que deixa claro os membros da equipe que estarão envolvidos durante o fluxo, e em que nível.

A Figura 3.3 ilustra um fluxo de trabalho. Apesar deste fluxo de trabalho ser responsabilidade de apenas um papel (o analista de negócio), um fluxo de trabalho pode ser responsabilidade de mais de um membro da equipe. Além disso, há diferentes níveis nos quais estes

Fluxo de Trabalho	***Principal Responsável***
Capturar Visão do Projeto	Analista de Negócio
Criar Cenário	Analista de Negócio
Criar Requisito de Qualidade de Serviço	Analista de Negócio
Planejar Iteração	Gerente de Projeto
Guiar Iteração	Gerente de Projeto
Guiar Projeto	Gerente de Projeto
Criar Arquitetura da Solução	Arquiteto
Implementar Tarefa de Desenvolvimento	Desenvolvedor
Corrigir Defeito	Desenvolvedor
Fazer *Build* de Produto	Desenvolvedor
Testar Cenário	Analista de Testes
Testar Requisito de Qualidade de Serviço	Analista de Testes
Finalizar Defeito	Analista de Testes
Fazer *Release* de Produto	Gerente de *Release*

Tabela 3.2 Relação entre fluxos de trabalho e as papéis do MSF responsáveis

membros da equipe podem estar envolvidos no fluxo de trabalho. Os graus de participação incluem os membros que são *responsáveis* pelo fluxo, sendo o papel ativo durante toda a sua execução, e os que são *consultados* para a realização de um fluxo.

Não obstante a seqüencia de atividades provida pela descrição do fluxo de trabalho, estas não constituem elementos atômicos, isto é, no MSF Ágil cada atividade pode ser decomposta em passos concretos para a realização da mesma. Tomando como exemplo a atividade *Escrever descrição de cenário*, notamos uma estrutura bastante semelhante a que encontramos em um fluxo de trabalho. De fato, a análise da Figura 3.4 revela tal semelhança, mostrando que uma atividade possui critérios de entrada e de saída, papéis envolvidos e uma decomposição em elementos, agora atômicos, chamados de sub-atividades.

Além dos fluxos, o MSF Ágil introduziu os *itens de trabalho* (*work items*), os quais correspondem a artefatos que sofrem alterações durante o processo de desenvolvimento como resultado da evolução natural do projeto. Os sete itens de trabalho cujos modelos (*templates*) são fornecidos pelo MSF Ágil são: tarefa, solicitação de mudança, risco, revisão, requisito, defeito e ocorrência.

Tarefa: é uma indicação da necessidade de realização de trabalho por parte de algum membro da equipe (escrever um caso de teste, desenvolver código a partir de um cenário, etc).

Solicitações de mudança: faz-se necessária quando da modificação de qualquer item que integre um *baseline* do projeto e deve ser analisada pelo comitê de controle de mudança (*change control board*) a fim de ser rejeitada ou aprovada.

Risco: documenta uma condição ou evento provável que pode ter efeitos negativos para o projeto. É parte essencial do processo de gerenciamento do projeto identificar e gerenciar os riscos inerentes ao projeto.

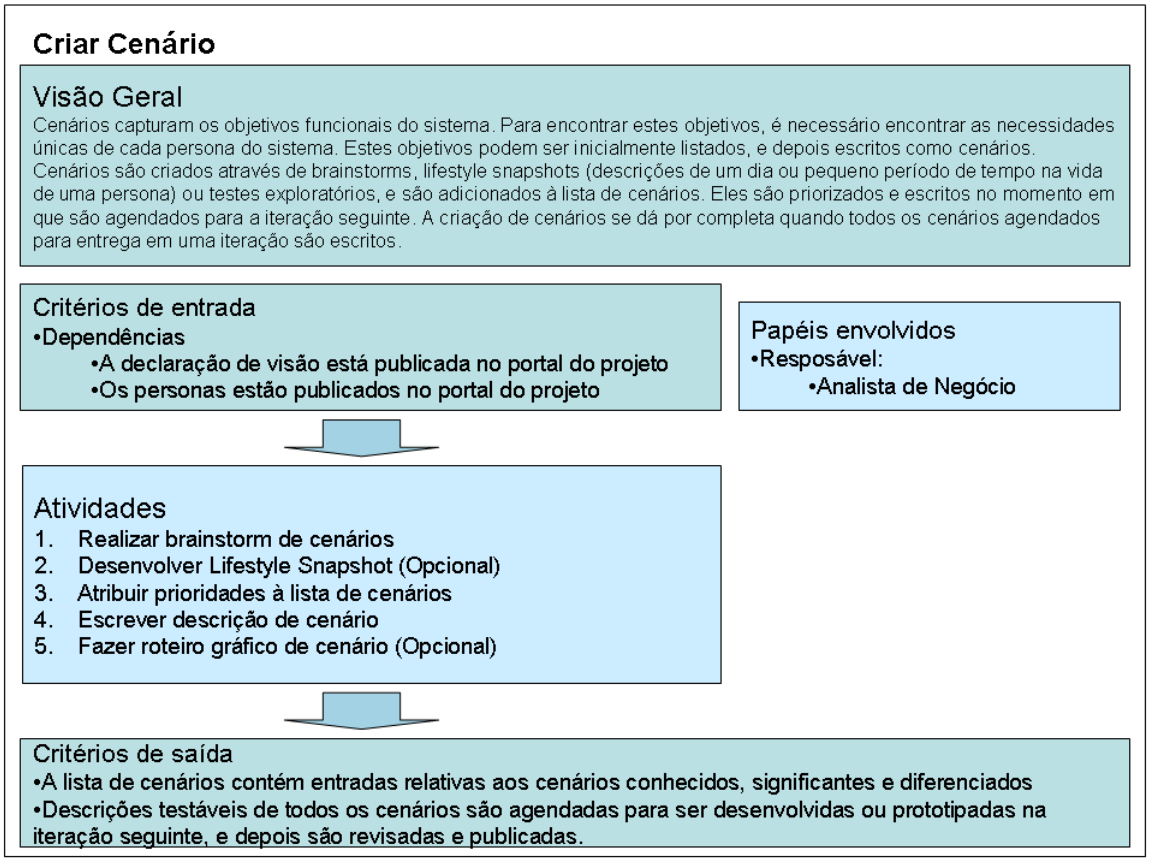

Figura 3.3 Fluxo de trabalho *Criar cenário*

Revisão: documenta os resultados de uma revisão de código ou de modelo. Deve trazer dados a respeito de como o código adequa-se a padrões de corretude de nomes, relevância, extensibilidade, complexidade algorítmica e segurança.

Requisito: captura o que o produto precisa ter para satisfazer as necessidades do cliente. Vários são os tipos de requisitos: cenários, qualidade de serviço, funcionais, operacionais e de interface. Este talvez seja o item de trabalho cuja presença é a mais pervasiva durante o ciclo de desenvolvimento, já que é capturado nas fases iniciais e guia as atividades de análise e projeto, codificação e testes.

Defeito: comunica que um problema potencial existe no produto. Sua descrição deve conter em que contexto o defeito foi encontrado, de maneira que este seja facilmente reproduzido, facilitando sua resolução.

Ocorrência: documenta um evento ou situação que tem a possibilidade de bloquear o andamento do projeto. Diferem dos riscos por serem identificadas de forma espontânea em reuniões de equipe. Uma vez identificadas, são geradas tarefas que, quando completadas com sucesso, implicam a finalização da ocorrência.

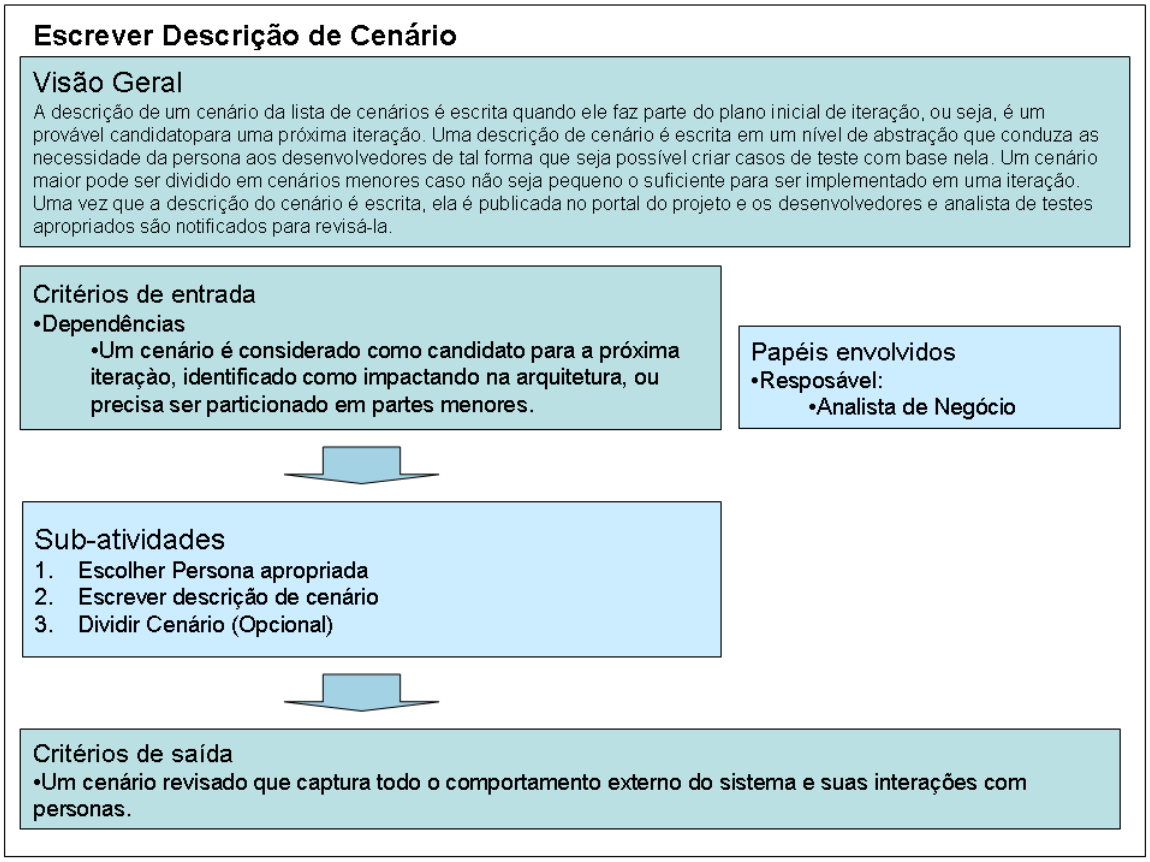

Figura 3.4 Atividade *Escrever descrição de cenário*

3.2 Considerações Finais

Este capítulo apresentou uma visão geral do *Microsoft Solutions Framework for Agile Software Development*. O MSF Ágil consiste em um conjunto de *mindsets*, princípios, um modelo de equipe e uma série de fluxos de trabalho. Os *mindsets* estão para o MSF assim como os valores estão para XP: são objetivos abstratos, que devem tentar ser seguidos independente do projeto, e representam a filosofia da metodologia.

Também como XP, o MSF Ágil apresenta uma série de princípios, que são uma concretização dos *mindsets*. Os fluxos de trabalho consistem em um conjunto detalhado de atividades e sub-atividades que devem ser realizadas pela equipe do projeto. Eles são o que mais se assemelham com as práticas de XP, mas há uma diferença conceitual grande entre eles: os fluxos de trabalho são bastante concretos, e as atividades de cada um deles são descritas em bastante detalhes, enquanto as práticas de XP são ainda um pouco abstratas e sujeitas à interpretação de quem as emprega. Por fim o modelo de equipe define os papéis participantes da equipe. O MSF Ágil possui diversas características ágeis, como iterações curtas e resposta rápida à mudanças.

CAPÍTULO 4

MSF versus XP

Este capítulo visa estabelecer uma análise comparativa entre o *Extreme Programming* e o MSF Ágil. Como a busca por agilidade é algo recente na metodologia MSF, a presente análise foi concebida de forma a tentar alinhar os elementos que compõem o MSF Ágil aos elementos de XP. Assim, os elementos do MSF Ágil são sempre apresentados nas comparações, procurando-se, em seguida, identificar a sua aderência às práticas, princípios e valores do *Extreme Programming*

A presente análise abrange o modelo de equipe e os fluxos de trabalho do MSF Ágil.

4.1 Modelo de Equipe vs. Papéis XP

Esta seção apresenta uma proposta de mapeamento entre os papéis do MSF em papéis de XP, explicando o que eles têm em comum e quais as suas diferenças.

4.1.1 Analista de negócio, Persona, *Gold Owner* e *Goal Donnor*

O analista de negócios do MSF possui uma contraparte bastante similar em XP: o *Gold Owner*. Ambos têm o dever de entender e garantir o sucesso do projeto, definindo qual deve ser o seu escopo (através da visão do projeto, como será apresentado na seção de fluxos de trabalho). Há no entanto, uma diferença entre estes papéis: no MSF o analista de negócio é responsável pela escrita de cenários e de requisitos de qualidade de serviço (ambos são equivalentes as estórias de XP). XP considera que o *Gold Owner* não é a pessoa mais indicada para fazer isso, por não necessariamente ser um usuário do projeto. O papel responsável por escrever as estórias é desempenhado pelo *Goal Donnor*. O MSF perde um pouco em flexibilidade por não admitir a possibilidade de separação entre o cliente que banca o projeto e o usuário, mas os objetivos do *Goal Donnor* são parcialmente satisfeitos via um elemento da metodologia chamado persona (é importante perceber que persona não é um papel). De qualquer forma, é razoável considerar que há uma compatibilidade total entre XP e MSF, uma vez que as duas alcançam os mesmos objetivos neste caso, apenas de maneiras diferentes.

4.1.2 Gerente de projeto MSF, Gerente XP e *Tracker*

O gerente de projeto MSF deve garantir a entrega de valor de negócio dentro do cronograma e orçamento acertados. As tarefas do gerente MSF consistem em: planejar iteração, guiar iteração e guiar projeto. No planejamento da iteração, ele deve trabalhar em conjunto com todos os outros papéis com exceção do gerente de *release*, o que também acontece com o gerente XP

no jogo do planejamento. No entanto, suas atribuições são diferentes: no MSF, o gerente estima a duração de tarefas (o que no XP deve ser feito pelo programador) e define cronogramas (o que é decidido entre os clientes e programadores no XP). Esta diferença contraria o princípio de *Responsabilidade Aceita*, e representa uma divergência de filosofia entre as duas metodologias.

O gerente MSF deve trabalhar em conjunto com o desenvolvedor para estimar a dificuldade e duração dos cenários e requisitos de qualidade de serviço. Em XP, é originalmente tarefa do programador sozinho fazer isso; no entanto, o gerente está presente no momento em que ele faz estas estimativas, e nada impede que ele ajude o programador se necessário. A diferença neste caso seria somente de quem lidera a atividade de estimar e agendar os cenários/estórias. O mesmo se dá com a quebra de um cenário/qualidade de serviço em tarefas atômicas: no MSF, o gerente lidera esta atividade, consultando o programador. Em XP, esta obrigação é do programador.

O gerente de projeto também deve medir o progresso da equipe, e verificar as métricas de testes. Em XP, existe um papel chamado *Tracker*, que tem a obrigação de aferir o progresso da equipe e tornar esse progresso compreensível para a equipe (como por exemplo, via *Informative Workspace)*. Como o *tracker* mede o andamento do projeto varia de caso para caso, e suas métricas vão além de apenas verificar cobertura dos testes. Pode-se dizer então que a conformidade do papel de Gerente do projeto com o papel de *Tracker* é média.

Além disso, o gerente de projeto do MSF também tem a obrigação de identificar riscos no projeto. Em XP não existe nenhuma prática que explicite esta necessidade. A metodologia foi concebida de forma a acomodar mudanças a qualquer momento, mas não há uma preocupação com tentar prever riscos específicos. Os riscos abordados são mais genéricos, como "mudança de requisitos". No entanto, o papel de gerente de XP tem flexibilidade para fazer isso, se achar necessário.

Existe um papel de XP que não é coberto declaradamente por nenhum papel do MSF: o *Coach*. Este papel tem a função de garantir que todos os integrantes do projeto sigam a metodologia corretamente. Apesar deste papel não estar presente no MSF, ele pode ser simulado pelo gerente da seguinte forma: no fluxo de trabalho *guiar projeto*, o gerente pode identificar um risco que consiste em garantir que todos os integrantes do time estejam seguindo a metodologia. Em seguida, basta mitigar este risco corretamente no fluxo de trabalho *Guiar Iteração*. Desta forma, é possível identificar um mapeamento entre todos os papéis gerenciais de XP (Gerente, *Tracker* e *Coach*) e o papel de gerente de produto do MSF.

4.1.3 Arquiteto MSF e Programador XP

O arquiteto no MSF tem a obrigação de realizar mudanças na arquitetura do projeto, mas somente no que é estritamente necessário. Uma vez que uma iteração vai começar, e a arquitetura precisa ser alterada para acomodar novas funcionalidades ou corrigir algum defeito, o arquiteto realiza estas alterações. O papel de arquiteto não existe em XP, mas o programador XP detém todas as obrigações que o arquiteto do MSF possui, idênticas (o programador possui outros papéis no entanto). É possível concluir que há conformidade entre estes dois papéis.

4.1.4 Desenvolvedor MSF, analista de testes MSF e programador XP

Estes são os papéis que apresentam a conformidade mais óbvia: sua lista de tarefas é praticamente idêntica, apresentando apenas algumas variações que não chegam a atrapalhar os objetivos principais de cada um. Como mencionado na sub-seção 4.1.2, em XP o programador é quem comanda as estimativas de dificuldade e duração das estórias, e no MSF, o gerente é quem realiza estas atividades, com a ajuda do desenvolvedor. Exceto por esta diferença, todas as atividades do desenvolvedor e analista de testes MSF estão incluídas dentro das atividades do programador. As únicas atividades que o programador XP possui que não estão nestes dois papéis estão compreendidas no papel de arquiteto.

4.1.5 Gerente de *release*

Este papel deve criar cronogramas para o lançamento de *releases*, validar estes *releases* e criar os *release notes*[1]. O MSF, assim como XP prega que a implantação deve ser frequente, e que cada versão construída do projeto deve ser um candidato a *release*. No entanto, em certos momentos pode-se definir datas chave para o lançamento de versões. Em XP estas datas podem ser escolhidas pelo gerente em conjunto com o *Goal Donnor* e o *Gold Owner*. A validação deve ser realizada pelo *Goal Donnor*, e os *release notes* podem ser simulados como uma estória para ser feita pelos programadores.

4.1.6 Visão geral da comparação de papéis

A Tabela 4.1 resume a relação e a conformidade entre os papéis das duas metodologias, indicando um nível de coincidência entre as atividades dos papéis. Um nível de coincidência *alto* significa que o mapeamento entre os papéis é quase direto. Um nível *médio* signifca que apenas parte das responsabilidades de uma metodologia é realizado pelo papel da outra, mas que estas responsabilidades são divididas entre no máximo dois papéis. Um nível *baixo* indica que apenas uma pequena parte das tarefas é comum aos papéis, e pode indica um espalhamento maior de papéis de uma metodologia nos papéis da outra, como é o caso do papel Gerente de *release* do MSF: suas tarefas estão espalhadas em três outros papéis, com um nível de coincidência de responsabilidades baixo.

4.2 Fluxos de atividades

Cada fluxo de atividades do MSF será mostrado a seguir; breves análises serão feitas a respeito de cada um, e em seguida será feita uma comparação com o XP. Por restrições de espaço este trabalho limita-se a oferecer uma visão comparativa dos fluxos de trabalho e das atividades, sem adentrar em discussões a respeito das sub-atividades. Uma vez que XP não possui fluxos de trabalho, nem algum elemento da metodologia que se assemelhe a eles em seu nível de detalhes, o critério para verificar a aderência destas práticas ao XP será a conformidade de

[1] Breve sumário da informação contida em cada *release*. Descreve requisitos de ambiente, detalhes de instalação, novas funcionalidades, defeitos corrigidos, defeitos conhecidos, etc.

Papel no XP	***Papel no MSF Ágil***	***Coincidência***
Gold Owner	Analista de Negócio	Alto
	Gerente de *Release*	Baixo
Goal Donnor	Analista de Negócio	Baixo
	Persona	Alto
	Gerente de *Release*	Baixo
Gerente	Gerente de Projeto	Alto
	Gerente de *Release*	Baixo
Tracker	Gerente de Projeto	Médio
Coach	Gerente de Projeto	Baixo
Programador	Arquiteto	Alto
	Desenvolvedor	Alto
	Analista de Testes	Alto

Tabela 4.1 Relação entre papéis de XP e do MSF Ágil, mostrando nível de coincidência de responsabilidades entre eles

seus objetivos com os valores, princípios e práticas principalmente. Dependendo no nível desta conformidade, a aderência do fluxo de trabalho será classificada como *forte*, *média* ou *fraca*. Ao final do capítulo, apresentaremos um sumário desta análise.

4.2.1 Capturar visão do projeto

1. Escrever declaração de visão
2. Definir personas
3. Refinar personas

Papel de defesa responsável: Analista de Negócio

Iniciar um projeto requer o estabelecimento claro da visão do mesmo. Capturar esta visão corretamente é de suma importância para que o foco no projeto esteja sempre adequado. A visão deve representar a essência do projeto. A partir desta essência, é necessário entender como o uso do pro jeto é feito, e qual o objetivo de seus usuários. A forma de uso e os objetivos dos usuários são capturados como personas.

É possível então encontrar dois objetivos principais neste fluxo: capturar a essência do projeto, e encontrar seus usuários. Ambos objetivos estão presentes na metodologia XP. A primeira atividade, escrever declaração de visão pode ser considerada equivalente a uma atividade em XP chamada *Escrever cartão de visão do projeto*. Ambas tem o mesmo fim, e são executadas por papéis equivalentes: o analista de negócios no MSF, e o *Gold Owner* no XP.

O segundo objetivo consiste em identificar as personas do projeto. Neste caso, existem algumas diferenças. O conceito de *persona*, como introduzido por Cooper [6], define um personagem fictício que representa um grupo particular de pessoas. Em XP, o papel de *Goal Donnor* possui estas características, entre outras. O *Goal Donnor* personifica todos os usuários

do sistema, e responde por eles. Apesar de não haver em XP o esforço direcionado para identificar todas as relações entre usuário e projeto em forma de atividades concretas como há no MSF, existe a flexibilidade para que exista mais de um *Goal Donnor* no projeto, o que pode ser feito caso seja necessário. Portanto, há a compatibilidade total entre as duas metodologias neste quesito.

4.2.2 Criar cenário

1. Fazer brainstorm de cenário
2. Desenvolver *lifestyle snapshot* (*Opcional*)
3. Priorizar lista de cenários
4. Escrever descrição de cenário
5. Fazer roteiro gráfico de cenário (*Opcional*)

Papel de defesa responsável: Analista de Negócio

Os cenários capturam os objetivos funcionais do sistema. Para encontrar estes objetivos, é preciso examinar as necessidades de cada persona integrante do sistema. As atividades deste fluxo de trabalho acontecem todas no jogo de planejamento XP. O *brainstorm*, o *lifestyle snaphshot* (que pode ser encarado como uma estória de XP, uma vez que é uma descrição de uso do sistema por uma persona), assim como as outras atividades. A diferença se dá em quem é responsável por estas atividades. Em XP, estas atividades são realizadas pelo o *Goal Donnor*, que representa os usuários do programa. Fazendo um paralelo com o MSF, é como se os personas escrevessem as estórias, e não o analista de negócios. Mesmo com essa diferença é possível considerar que há alguma conformidade entre XP e MSF nesta atividade, porque apesar de não sendo o cliente que escreve os cenários no MSF, os personas e suas necessidades são capturados por duas atividades integrantes do fluxo, a saber *Fazer brainstorm de cenário* e *Desenvolver lifestyle snapshot*. O desenvolvimento do *lifestyle snapshot*, como já foi explicado no capítulo anterior, consiste em uma descrição de um dia ou de um curto período de tempo na vida de uma persona, e ajuda a demonstrar como o sistema ou produto serve para facilitar a sua vida.

A atividade 3, *Priorizar lista de cenários* é feita pelo *Gold Owner* no XP, um papel equivalente ao analista de negócio.

4.2.3 Criar requisito de qualidade de serviço

1. Fazer *brainstorm* de requisitos de qualidade de serviço
2. Desenvolver *lifestyle snapshot* (*Opcional*)
3. Priorizar lista de requisitos de qualidade de serviço
4. Escrever requisito de qualidade de serviço

5. Identificar objetivos de segurança

Papel de defesa responsável: Analista de Negócio

Os requisitos de qualidade de serviço representam condições especiais de cenários que precisam ser atendidas. Uma vez que XP trata ambos cenários e requisitos de qualidade de serviço da mesma maneira (fazendo uso de estórias para representar os dois), todas as considerações feitas para o fluxo de trabalho *Criar cenário* na sub-seção 4.2.2 valem para este fluxo (a semelhança entre os dois fluxos é facilmente perceptível também no MSF, bastando comparar as atividades integrantes dos dois).

4.2.4 Planejar iteração

1. Determinar duração da iteração
2. Estimar cenário
3. Estimar requisito de qualidade de serviço
4. Agendar cenário
5. Agendar requisito de qualidade de serviço
6. Alocar tempo para correção de defeitos
7. Dividir cenários em tarefas
8. Dividir qualidade de serviço em tarefas

Papel de defesa responsável: Gerente de projeto

Papéis consultados: Analista de Negócio, Arquiteto, Desenvolvedor e Analista de Testes

Neste fluxo, o principal objetivo é determinar que cenários e requisitos de qualidade de serviço serão realizados nesta iteração. A primeira atividade, de determinar a duração da iteração, é feita com mais freqüência no início do projeto, quando a equipe ainda está otimizando as iterações. Depois de um tempo, quando uma duração ideal é alcançada (uma semana por exemplo), espera-se que ela não sofra mais grandes modificações. É possível no entanto fazer pequenos ajustes na duração da iteração caso haja pequenas mudanças relativas ao projeto, como o acréscimo ou remoção de participantes, mudança da data final de entrega do projeto (caso já haja uma data pré-definida). Este modo de lidar com as iterações é equivalente ao que se faz em XP, com a prática *Weekly Cycle*, suportada pelo princípio *Passos Pequenos*.

Nas atividades 2 e 3, o gerente deve trabalhar em conjunto com o desenvolvedor para estimar a dificuldade e o tempo para se realizar as tarefas candidatas à iteração corrente. A diferença em relação a XP é somente em quem faz estas atividades. Em XP, o responsável principal por estimar estas dificuldades é o programador, uma vez que ele é quem detém conhecimento para tal. Outra diferença é que em XP para realizar esta estimativa, o programador

também deve quebrar as estórias em tarefas menores, como um passo-a-passo do ponto de vista de implementação. Estas tarefas são mais tangíveis ao programador, e consequentemente mais fáceis de estimar, fazendo com que a estimativa da estória como um todo seja mais precisa. No MSF, os cenários só são quebrados em tarefas nas últimas atividades do fluxo (7 e 8), depois que os cenários já foram escolhidos para fazer parte desta iteração. O particionamento no MSF tem o intuito de facilitar a escrita de testes, e refinar a estimativa de tempo anterior.

Nas atividades 4, 5 e 6, o gerente trabalha junto com o analista de negócio para realizar uma estimativa superficial sobre a duração dos cenários e para decidir quanto tempo deve ser alocado para a resolução de defeitos. Em XP, nenhuma destas atividades acontece desta forma. Só os programadores estimam o tempo de duração de uma tarefa, uma vez que eles é quem vão implementá-la. E não existe tempo alocado especificamente para a correção de defeitos. Defeitos encontrados são registrados em forma de estórias, que entram no jogo do planejamento, e devem ser tratadas como qualquer outra estória. Com base nestas estimativas é que o cliente escolhe que estórias serão feitas, e em que ordem.

Podemos perceber discrepâncias entre este fluxo de trabalho e os objetivos de XP. Apesar deste fluxo realizar boa parte das atividades que fazem parte do *jogo do planejamento*, os papéis responsavéis por elas são diferentes. Em XP há uma preocupação em fazer com que o programador e apenas ele crie as suas estimativas, em detrimento de aceitar estimativas propostas (ou impostas) por outra pessoa. Isto é embasado pelo princípio *Responsabilidade Aceita*, apresentado na seção 2.3. A não conformidade com um princípio é grave, e indica uma diferença importante entre as duas metodologias.

4.2.5 Guiar projeto

1. Revisar objetivos
2. Estimar progresso
3. Avaliar limites de métricas de testes
4. Fazer triagem dos defeitos
5. Identificar risco

Papel de defesa responsável: Gerente de projeto

O objetivo deste fluxo de trabalho é monitorar constantemente o progresso do projeto, realizando mudanças necessárias para mantê-lo no caminho certo. Suas três primeiras atividades são tarefa do *tracker* em XP: determinar métricas que possam ser usadas para acompanhar o andamento do projeto, coletar estas métricas e deixa a equipe a par delas. A diferença em relação ao conteúdo das atividades é que no MSF, há apenas o direcionamento para coletar métricas relacionadas a testes, onde em XP o *tracker* pode coletar qualquer tipo de métrica que julgar necessária. Desta forma, a abrangência da função de acompanhamento do andamento do projeto é maior em XP do que no MSF.

A atividade 4 ilustra mais uma diferença entre as duas metodologias. Enquanto no MSF a triagem dos defeitos é realizada pelo gerente de projeto, em XP ela é realizada pelo cliente

(*Goal Donnor*, ou até mesmo o *Gold Owner* caso seja um defeito que cause impacto no valor de negócio do projeto), já que ele é quem melhor pode determinar o que é essencial para o produto. Os defeitos encontrados são escritos em forma de estórias, e voltam para a pilha de estórias que devem ser implementadas. No próximo jogo do planejamento, estas estórias serão escolhidadas para serem feitas, se forem importantes naquele momento.

A atividade de identificar risco é algo que existe no MSF, mas não aparece de forma explícita em XP. XP enfrenta os riscos de maneira geral, se mantendo ágil para poder realizar mudanças se necessário, mas não define práticas que se preocupem com a prevenção de riscos específicos.

4.2.6 Guiar iteração

1. Monitorar iteração
2. Mitigar risco
3. Realizar retrospectiva

Papel de defesa responsável: Gerente de projeto

Este fluxo de atividade é bastante simples: o seu objetivo principal é monitorar o andamento do projeto, e remover os empecilhos encontrados para que a equipe possa trabalhar normalmente. Ao fim da iteração, dar e receber *feedback* para a equipe com o intuito de saber o que deu certo e o que não deu. Este fluxo de atividade é integralmente condizente com parte das atividades do gerente de projeto XP. O gerente deve ser um facilitador, e deixar a equipe trabalhar.

4.2.7 Criar solução de arquitetura

1. Particionar o sistema
2. Determinar interfaces
3. Desenvolver modelo de ameaça
4. Desenvolver modelo de performance
5. Criar protótipo de arquitetura
6. Criar infra-estrutura de arquitetura

Papel de defesa responsável: Arquiteto

O objetivo deste fluxo é criar soluções de arquitetura o mais simples possível para resolver um problema. Este fluxo está completamente de acordo com o valor *Simplicidade* e com a prática *Incremental Design*, encontrados em XP. A arquitetura só deve ser alterada no momento de acomodar novas funcionalidades, e não se deve criar nada além do estritamente necessário

para a solução funcionar. E, protótipos podem ser criados para facilitar o entendimento de um problema, caso não haja uma solução conhecida para tal. Em XP, o papel que realiza as atividades correspondentes a estas é o programador.

4.2.8 Implementar tarefa de desenvolvimento

1. Estimar duração de tarefa de desenvolvimento
2. Criar ou atualizar teste unitário
3. Escrever código para tarefa de desenvolvimento
4. Realizar análise de código
5. Rodar testes unitários
6. Refatorar código
7. Revisar código
8. Integrar mudanças de código

Papel de defesa responsável: Desenvolvedor

A primeira atividade deste fluxo, estimar duração de tarefa de desenvolvimento não acontece neste momento na implementação de uma tarefa em XP. As tarefas são estimadas durante o jogo do planejamento, de forma que quando o programador vai começar a implementá-las já tem uma estimativa de dificuldade e tempo. A atividade 2 é idêntica a prática de XP *Test-First Programming*, que visa a implementação dos testes antes das estórias.

Até a atividade 5, este fluxo de trabalho descreve o comportamento do desenvolvedor MSF da mesma forma que se espera o comportamento de um programador XP. Uma diferença importante acontece na atividade 6, *Refatorar código*. No MSF, após implementar uma tarefa de desenvolvimento e verificar que ela está funcionando, o desenvolvedor deve procurar áreas onde a arquitetura do programa apresenta complexidade demasiada para realizar *refactoring*[2] no código, melhorando a arquitetura. Isto é uma divergência entre as duas metodologias: em XP, quando uma funcionalidade está funcionando, não se perde tempo realizando mudanças nela. Se os testes passam, o programador pode seguir em frente e realizar outras tarefas. Não se deve fazer melhorias na arquitetura só por fazer. O momento de modificar a arquitetura deve ser o mais tarde possível, somente quando vai haver necessidade imediata. Se para a inclusão de uma nova funcionalidade é necessário realizar mudancás no código, então neste caso o programador pode realizar estar mudanças. Portanto, esta atividade (*Refatorar código*) *neste* ponto do fluxo de trabalho contraria a prática *Incremental design*.

As atividades 7 e 8, estão de acordo com o comportamento esperado de um programador XP, e inclusive atendem ao principio *Fluxo*, e a prática *Integração Contínua*.

[2]Mudanças de estrutura que não alteram o comportamento de um código-fonte. *Refactorings* podem deixar o código mais limpo e fácil de ser entendido e mantido.

4.2.9 Corrigir defeito

1. Reproduzir defeito
2. Criar ou atualizar teste unitário
3. Localizar causa do defeito
4. Reassinalar defeito
5. Adotar estratégia de correção de defeito
6. Codificar a correção do defeito
7. Rodar testes unitários
8. Refatorar código
9. Revisar código
10. Integrar mudanças de código

Papel de defesa responsável: Desenvolvedor

XP trata defeitos da mesma maneira que trata novas funcionalidades: através de estórias. Não existe um papel ou momento específico em XP que seja destinado à correção de tarefas. Podemos observar uma certa semelhança entre este fluxo e o fluxo *Implementar tarefa de desenvolvimento* (sub-seção 4.2.8): sua estrutura lógica é bastante parecida, e eles possuem 5 atividades em comum (criar e rodar testes unitários, refatorar, revisar e integrar mudanças no código). Podemos generalizar as observações feitas para aquela atividade, para este caso, com a principal divergência em relação ao XP sendo a realização de refatoramento após o defeito já ter sido corrigido.

4.2.10 Fazer *build* de produto

1. Executar o *build*
2. Verificar o *build*
3. Corrigir o *build*
4. Aceitar *build*

Papel de defesa responsável: Desenvolvedor

Este fluxo de trabalho é bastante simples, e não apresenta nenhuma discrepância em relação a XP. Muito pelo contrário, é importante que os *builds* sejam verificados para que estejam sempre simples, e para que problemas possam ser corrigidos assim que apareçam. No MSF este fluxo deve ser executado assim que um novo conjunto de mudanças tiver pronto para ser integrado ao sistema pelo desenvolvedor, assim como acontece em XP, com o programador. Este fluxo de trabalho condiz com diversos elementos de XP, entre eles *Ten-Minute Build*, *Fluxo* e *Integração Contínua*.

4.2.11 Testar cenário

1. Definir abordagem de teste
2. Escrever testes de validação
3. Selecionar e executar caso de teste
4. Abrir um defeito (*Condicional*)
5. Conduzir testes exploratórios

Papel de defesa responsável: Analista de testes

A primeira diferença importante entre o MSF e XP neste fluxo de trabalho é o responsável por testar o sistema. Em XP, a maioria dos testes é realizada pelo cliente, se possível inclusive com o sistema novo rodando já no ambiente onde se espera que ele seja implantado. Desta forma, os testes são realizados da maneira mais natural possível, e os erros relevantes para o bom funcionamento do produto serão descobertos naturalmente. Entretanto, é possível escrever estórias para realizar testes mais específicos, como testes de carga, *stress*, rede, etc. se estes forem necessidades declaradas pelo cliente. Estes testes, seriam então realizados pelos programadores. Em linhas gerais, a principal diferença em relação a XP é o papel que executa os testes: no MSF o analista de testes, e em XP o cliente e possivelmente o programador.

4.2.12 Testar requisito de qualidade de serviço

1. Definir abordagem de teste
2. Escrever testes de performance
3. Escrever testes de segurança
4. Escrever testes de *stress*
5. Escrever testes de carga
6. Selecionar e executar caso de teste
7. Abrir um defeito (*Condicional*)
8. Conduzir testes exploratórios

Papel de defesa responsável: Analista de testes

Uma vez que XP trata os cenários do MSF e seus requisitos de qualidade de serviço da mesma formá (estórias), tudo o que foi dito sobre o fluxo de trabalho *Testar um cenário* (subseção 4.2.11) se aplica para este fluxo, com a diferença que em XP, estes testes especificamente sempre são realizados pelo programador (mas são definidos pelo cliente).

4.2.13 Finalizar defeito

1. Verificar correção
2. Fechar defeito

Papel de defesa responsável: Analista de testes

Estas atividades não aparecem em XP da forma como são feitas no MSF. No MSF, um defeito é corrigido e separado em momentos diferentes. Em XP, como um defeito é representado através de estórias, no momento em que a estória é concluída e passa pelos testes de aceitação, o defeito já foi resolvido. Ele vai ser verificado mais uma vez quando o sistema for implantado e o usuário foi utilizá-lo, o que deve acontecer com certa freqüência.

4.2.14 Fazer *release* de produto

1. Executar plano de *release*
2. Validar *release*
3. Criar *release notes*
4. Implantar produto

Papel de defesa responsável: Gerente de *release*

Papéis consultados: Desenvolvedor

Os detalhes deste fluxo já foram descritos na Seção 4.1.5. A diferença que este fluxo tem em relação a XP é o papel responsável por executá-lo. Enquanto no MSF há um papel específico para isso, em XP os clientes é que definem as datas-chave para lançamento de *releases*, uma vez que estas datas geralmente são marcos para o negócio deles. Em relação à atividade 3, *Criar release notes*, apesar de não haver em XP uma atividade relevante ela pode ser simulada através de uma estória, sendo feita pelo programador. A implantação do produto (atividade 4) acontece em XP, ficando a cargo dos programadores, como foi dito no princípio XP do *Fluxo* (seção 2.3): o o programa deve estar sempre disponível, atualizado e pronto para ser implantado. No entanto podemos considerar que há uma certa conformidade entre os dois, uma vez que o programador MSF é um dos papéis que deve ser consultado para a realização deste fluxo de atividade.

4.3 Considerações Finais

Este capítulo apresentou as principais diferenças entre o MSF e o *Extreme Programming*. Nesta seção será feito um sumário de todas estas diferenças, para que se possa ter uma visão maior da comparação entre as duas metodologias. É importante observar que há diferentes formas de divergência entre as duas metodologias: a primeira e mais óbvia forma é a diferença de práticas, que consiste em uma metodologia ter práticas que a outra não possui, ou práticas semelhantes

que apresentam estruturas diferentes. Pode haver também diferenças de ordem temporal, ou seja, as duas metodologias podem possuir as mesmas práticas, mas com a ordem de algumas atividades diferente, o que pode representar uma grande diferença nas práticas. Por fim, é importante levar em consideração os papéis responsáveis pela execução dos fluxos de trabalho.

Foram encontradas muitas diferenças entre as duas metodologias. No entanto, as duas metodologias são flexíveis o suficiente para que quase todas estas diferenças de práticas possam ser superadas, simulando práticas de uma na outra. Estas diferenças de práticas são simples, e não são muito importantes. No entanto, há importantes diferenças temporais e de papel que representam divergências entre os valores, e conseqüentemente entre as filosofias das duas metodologias. Estas diferenças são bastante relevantes. Entre estas, existe uma que se destaca mais que todas as outras, e que é responsável por boa parte das outras diferenças. O MSF Ágil falha em ver o cliente como parte da equipe, e acaba delegando tarefas para papéis que não são os mais indicados para realizá-los. Isto não significa que o cliente é completamente esquecido no processo: o MSF Ágil tem a sua maneira de capturar e atender às suas necessidades, através de personas. Mas um dos principais moldes do *Extreme Programming* é a sua integração com o cliente, com *Comunicação* e *Feedback* constante, que são valores que ficam comprometidos quando o cliente passa a ser só um espectador do processo, ao invés de um participante. O cliente como parte integrante da equipe é a melhor maneira de ter certeza se o que está sendo feito está certo. O MSF Ágil tenta substituir o cliente por documentação que tenta capturar as suas necessidades, mas esta documentação nunca pode ser detalhada o suficiente para compreender todas as necessidades de interaçãom com o cliente, e muitas vezes é falha. Apesar de todas as diferenças em relação a XP, o MSF Ágil conseguiu criar um processo com bons indicadores de agilidade, que são comprovados por todas as semelhanças que também há entre as duas metodologias. Sua única falha restringe-se à ótica adotada em relação ao cliente, que vai totalmente contra o que é pregado no manifesto ágil: “Customer collaboration over contract negotiation”. O MSF Ágil ainda se prende à negociação de contrato ao invés de colaboração com o cliente.

Outra diferença importante entre as metodologias é a falta no MSF Ágil de uma pessoa que mantenha a equipe sempre aderente à metodologia, como o *Coach* faz em XP. Não adianta definir uma metodologia tão rica em detalhes se ela não for seguida corretamente. O MSF assume que todos os integrantes saberão o seu papel e suas responsabilidades na equipe, e não atribui a ninguém o dever de verificar e garantir que estes papéis estão sendo seguidos. Esta é uma premissa muito arriscada de se assumir, e dificilmente será satisfeita.

A Tabela 4.2 ilustra a relação entre os fluxos de trabalho do MSF e os elementos de XP que satisfazem os objetivos ou dão suporte de alguma forma às suas atividades, indicando em seguida o grau de satisfação desta relação.

Fluxo de Trabalho	***Elemento de XP***	***Papel no XP***	***Satisfação***
Capturar Visão do Projeto	Cartão de Visão do Projeto Papel de *Goal Donnor*	Gold Owner *Goal Donnor*	Forte
Criar Cenário	Jogo do Planejamento	Programador Gerente *Goal Donnor* *Gold Owner*	Média
Criar Requisito de Qualidade de Serviço	Jogo do Planejamento	Programador Gerente *Goal Donnor* *Gold Owner*	Média
Planejar Iteração	Jogo do Planejamento	Gerente Programador *Goal Donnor*	Fraca
Guiar Projeto	Métricas	*Tracker*	Média
Guiar Iteração	Papel de gerente	Gerente	Forte
Criar Arquitetura da Solução	*Incremental Design*	Programador	Forte
Implementar Tarefa de Desenvolvimento	Papel do Programador *Test-First Programming*	Programador	Média
Corrigir Defeito	Papel do Programador *Test-First Programming*	Programador	Média
Fazer *Build* de Produto	Papel do Programador *10-minute build* *Continuous Integration*	Programador	Forte
Testar Cenário	Papel do *Goal Donnor* Papel do Programador	*Goal Donnor* Programador	Média
Testar Requisito de Qualidade de Serviço	Papel do Programador	Programador	Forte
Finalizar Defeito	*Stories*	Programador	Fraca
Fazer *Release* de Produto	*Continuous Integration*	Programador Gerente *Goal Donnor* *Gold Owner*	Fraca

Tabela 4.2 Relação de conformidade entre fluxos do MSF e elementos de XP

CAPÍTULO 5

Conclusões e trabalhos futuros

"Quem ri por último, não entendeu a piada."
—HELEN GIANGREGORIO

Este trabalho realizou uma análise comparativa entre o *Microsoft Solutions Framework for Agile Software Development* e o *Extreme Programming*. Para tal, foram feitas breves explicações sobre cada uma das metodologias, e em seguida a comparação se deu de duas formas: o modelo de equipe, e os fluxos de trabalho. Os papéis do modelo de equipe do MSF foram comparados com os papéis de XP; em seguida os fluxos de trabalho do MSF foram analisados, e comparados com práticas ou elementos de XP que realizavam as atividades que compunham cada fluxo.

Apesar de já existir desde 1994, esta versão foi a primeira na qual o MSF procurou dar suporte a desenvolvimento com agilidade. Uma boa maneira de verificar o nível de agilidade introduzido na metodologia foi realizar um estudo comparativo com a mais conhecida e utilizada das metodologias ágeis, XP. Por conta disso, o objetivo principal ao realizar a comparação entre estas metodologias foi verificar que nível de semelhança o MSF Ágil apresenta em relação a XP.

Como foi mostrado no Capítulo 4, o MSF apresenta várias diferenças em relação a XP. Muitas delas no entanto, podem ser supridas fazendo uso da flexibilidade das duas metodologias. Há no entanto uma divergência de conceitos no que tange à definição de papéis: o MSF captura as necessidades do cliente através de documentação, e XP prega que o cliente deve fazer parte da equipe, para gerar *feedback* rápido e correto. Mas isto não compromete a agilidade da metodologia, que ainda está em processo de desenvolvimento, e pode inclusive sofrer mudanças neste quesito. Uma sugestão de mudança para o MSF seria a incorporação das personas no modelo de equipe, fazendo com que elas assumissem os fluxos de trabalho relativos à escrita e validação de cenários, requisitos de qualidade de serviços e defeitos.

A principal dificuldade encontrada na comparação foram as freqüentes mudanças que as duas metodologias estão sofrendo atualmente. *Extreme Programming* não possui um órgão regulamentador, que sirva como referência absoluta, então tudo que integra a metodologia é bastante sujeito à interpretação, havendo inclusive muitas divergências entre participantes da comunidade ágil. O MSF Ágil por outro lado ainda se encontra numa versão *beta*, o que implica em pouca literatura disponível sobre o assunto, e muitas vezes documentação incompleta ou inconsistente. Há também a maneira como as duas metodologias são apresentadas: XP de maneira sempre abstrata, exigindo interpretação, e o MSF de maneira concreta, com detalhes até o nível de sub-atividades, que implicou em um grande esforço para diminuir o *gap* de

abstração entre as duas metodologias para realizar a comparação dos fluxos de trabalho.

Uma sugestão de trabalho futuro consiste em realizar experimentos práticos com o intuito de validar os resultados obtidos neste trabalho. Embora a comparação apresentada aqui tente abranger um grande nível de detalhes das duas metodologias, este estudo é essencialmente teórico, e seus resultados não podem ser considerados definitivos. Existem várias nuances que acontecem em projetos reais que não têm como ser capturadas num simples estudo das definições das metodologias. Para validar este estudo, seria importante a realização de projetos similares utilizando as duas metodologias, onde houvesse um acompanhamento destes projetos e uma coleta de dados relacionados ao desenvolvimento. Desta forma seria possível confirmar e obter maiores conclusões sobre os resultados apresentados neste trabalho.

Referências Bibliográficas

[1] Manifesto for agile software development. http://agilemanifesto.org.

[2] Mauro Araújo. *Avaliando a Metodologia Pro.NET em relação ao MSF 4*, August 2005.

[3] Liz Barnett. Big changes coming for rational unified process. Disponível em: http://www.forrester.com/Research/Document/Excerpt/0,7211,37180,00.html. Data de acesso: 10/10/2005.

[4] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2004.

[5] G. Booch, I. Jarcobson, and J. Rumbaugh. *The Unified Software Development Process*. Addison-wesley, 1999.

[6] Alan Cooper. *The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How to Restore the Sanity*. Sams, 1999.

[7] CTXML. Site contendo toda a metodologia PRO.NET. Disponível em : http://www.cin.ufpe.br/∼ctxmlrec. Data de acesso: 10/10/2005.

[8] Amr Elssamadisy. Xp on a large project a developer's view. Disponível em: http://www.agilealliance.org/articles/elssamadisyamrxponala/file/. Data de acesso: 30/01/2006.

[9] Standish Group. *2001 update to the CHAOS report*, 2001. Disponível em: http://www.standishgroup.com/sample_research/. Data de acesso: 18/08/2005.

[10] Microsoft. *MSF For Agile Software Development*. Disponível em: http://msdn.microsoft.com/vstudio/teamsystem/msf/msfagile/default.aspx. Data de acesso: 07/02/2006.

[11] Microsoft. *MSF For CMMI Process Improvement*. Disponível em: http://lab.msdn.microsoft.com/teamsystem/workshop/msfcmmi/default.aspx. Data de acesso: 07/02/2006.

[12] Microsoft. *MSF Team Model v. 3.1*, June 2002. Disponível em: http://www.microsoft.com/technet/itsolutions/msf/default.mspx. Data de acesso: 07/02/2006.

[13] Microsoft. *Microsoft Solutions Framework version 3.0 Overview*, July 2003. Disponível em :http://www.microsoft.com/technet/itsolutions/msf/default.mspx. Data de acesso: 07/02/2006.

[14] P. Naur and Randell B., editors. *Proceedings of the NATO Conference on Software Engineering*, Garmish, Germany, October 1968. NATO Science Committee.

[15] W. W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, August 1970.

[16] William Wake. *Extreme Programming Explored*. Addison-Wesley, 2002.

Este volume foi tipografado em LaTeX na classe UFPEThesis (www.cin.ufpe.br/~paguso/ufpethesis).