Vinicius Almeida Castro

UNIVERSIDADE FEDERAL DE SERGIPE

CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DO CURSO

PROJETO SUPERVISIONADO

# DESENVOLVIMENTO ÁGIL COM PROGRAMAÇÃO EXTREMA EFICÁCIA E DISCIPLINA EXTREMA NO DESENVOLVIMENTO ORIENTADO A OBJETOS DE SOFTWARE

São Cristóvão – SE

2006

Vinicius Almeida Castro

# Monografia de Projeto Supervisionado

Monografia da disciplina Trabalho de Conclusão do Curso.
Ciência da Computação da Universidade Federal de Sergipe.

Orientador:

Prof. MSc. Giovanny Fernando Lucero Palma

# Agradecimentos

Agradeço a Deus por ter me dado saúde e ao apoio dos meus pais, que se preocupam tanto comigo.

Agradeço pela paciência, pela compreensão e principalmente pelo grande apoio, fundamental para a conclusão deste trabalho, do meu orientador formal, professor Giovanny Fernando Lucero Palma.

Agradeço especialmente e com grande destaque ao amigo e orientador informal, Mauricio de Lemos Collares, com quem tive a oportunidade de aprender muito. Sem a ajuda dele, com certeza este trabalho não teria ido longe devido a meu desconhecimento completo sobre a maior parte dos assuntos abordados, principalmente no início, e também devido à escassez de documentação de qualidade em português, e muitas vezes também em inglês, sobre os assuntos abordados, principalmente os mais técnicos.

Agradeço a Vinícius Manhães Teles, que na minha opinião é o maior nome da XP no Brasil, pelo excelente livro em português, pelo conteúdo vasto sobre XP disponibilizado na Internet, por tudo que ele fez e vem fazendo para ajudar as metodologias ágeis a crescer no Brasil. E como se não bastasse tudo isto, por ter me dado apoio direto, respondendo a emails, colocando-se a disposição para ajudar e fazendo comentários no meu *blog*.

Agradeço a professora Kênia Kodel Cox, pelo grande apoio dado na preparação da apresentação desta monografia. Os ensinamentos e dicas passados por ela durante horas e horas de um único dia, além dos textos gentilmente cedidos, com certeza foram fundamentais para minha grande satisfação depois de encerrada a apresentação. Sempre a admirei, tanto como profissional, quanto como pessoa, por isso deixo meus sinceros parabéns por tudo que ela já fez e fará pelos alunos do departamento de computação da UFS.

Agradeço a amiga Grazziela Nilo. Pela cobrança que me ajudou muito a não perder a disciplina, pelo apoio durante quase toda a produção deste trabalho, pela torcida e pelos bons momentos.

Agradeço a Elisângela Maria Alves de Oliveira Rocha, pela amizade, pelo apoio e pela torcida sincera.

Agradeço a todos que trabalham no CPD da UFS, com destaque para o setor de desenvolvimento de sistemas, onde, como desenvolvedor, tive a oportunidade de aprender muito. Agradeço com destaque a compreensão da coordenadora Estelamaris, que apesar de não concordar muito com metodologias ágeis, me deu todo o apoio para estudar e adotar  práticas ágeis nos projetos em que trabalho individualmente.

## SUMÁRIO

## LISTA DE FIGURAS

# ABSTRACT

Almeida Castro, Vinicius, Agile Development with Extreme Programming. Advisor: Giovanny Fernando Lucero Palma. São Cristóvão : Universidade Federal de Sergipe / Computer Science Department, 2007. Term paper for the “Trabalho de Conclusão do Curso” course.

Every day sees the rise in demand for increasingly sophisticated systems and applications. However, according to statistics, the majority of software projects in the last decades failed due to reasons such as expenses that are greater than the budget, duration larger than what's scheduled, unsatisfactory features, low quality or project cancellation, among others. This work presents a new software development methodology, known as Extreme Programming or XP, which has been very successful lately exactly because it helps minimize flaws in projects and software.

This work studies the agile paradigm and discusses values, principles and practices of Extreme Programming, the most known and used agile methodology nowadays.

***Keywords:*** *Extreme Programming, XP, Agile Development, Agile Manifesto, TDD.*

# RESUMO

Almeida Castro, Vinicius, Desenvolvimento Ágil com Programação Extrema. Orientador: Giovanny Fernando Lucero Palma. São Cristóvão : Universidade Federal de Sergipe / Departamento de Computação, 2007. Monografia para a disciplina Trabalho de Conclusão do Curso.

A cada dia aumenta a demanda por sistemas e aplicações cada vez mais sofisticados. Porém, segundo as estatísticas, nas últimas décadas a grande maioria dos projetos de software falhou por motivos como gastos maiores que o orçamento, consumo de tempo maior que o cronograma, funcionalidades insatisfatórias, baixa qualidade ou cancelamento total do projeto, entre outros. Esse trabalho apresenta uma nova metodologia de desenvolvimento, conhecida por Programação Extrema ou XP (*Extreme Programming*), que vem fazendo muito sucesso nos últimos anos justamente por ajudar a minimizar as falhas nos projetos e software.

Este trabalho estuda o paradigma ágil e apresenta os valores, princípios e práticas da XP, que é a metodologia ágil mais conhecida e utilizada na atualidade.

***Palavras Chave***: *Programação Extrema, XP, Extremme Programming, Desenvolvimento Ágil, Manifesto Ágil, TDD.*

DCOMP / UFS
Graduação em Ciência da Computação

Orientador:
Prof. MSc. Giovanny Fernando Lucero Palma

# 1. Introdução

Este trabalho tem dois objetivos principais. O primeiro é mostrar em detalhes todo o embasamento teórico por trás da Programação Extrema ou XP (*Extreme Programming*), e assim ajudar a disseminar informações de qualidade sobre este assunto. Este objetivo é muito importante devido à adoção mínima da XP no Brasil em relação aos países desenvolvidos, e também devido ao número muito reduzido de livros em português sobre XP, principalmente livros atualizados que descrevem os valores, princípios e práticas mais atuais. O segundo objetivo é esclarecer porque a XP não é apenas mais uma maneira de fazer software, e sim uma ótima forma de romper com o paradigma tradicional de desenvolvimento que é ineficiente há décadas, e adotar um paradigma totalmente inovador que leva ao desenvolvimento de sistemas com alta qualidade de forma mais econômica e produtiva.

No mundo atual, a cada dia aumenta a demanda por um grande numero de sistemas e aplicações cada vez mais sofisticados, porém desenvolver software é uma atividade difícil e arriscada.

Segundo as estatísticas, entre os maiores riscos do desenvolvimento de software estão: gastos que superam o orçamento, consumo de tempo que supera o cronograma, funcionalidades que não resolvem os problemas dos usuários, baixa qualidade dos sistemas desenvolvidos e cancelamento do projeto por inviabilidade.

Segundo o conceito tradicional, uma metodologia de desenvolvimento de software define conjuntos de atividades e resultados associados que auxiliam, entre outras coisas, a reduzir o alto risco associado a esta atividade. Neste trabalho, as diversas metodologias de desenvolvimentos existentes na literatura serão agrupadas em duas categorias, de acordo com o modelo básico de desenvolvimento adotado. Estes modelos – ágil e tradicional – estão descritos comparativamente no capítulo 2.

O capítulo 3 é o principal deste trabalho. É nele que estão descritos em detalhes os valores, princípios e práticas da XP, que é a metodologia de desenvolvimento ágil mais conhecida e utilizada. Esta metodologia vem fazendo muito sucesso em diversos países, por ajudar a criar sistemas de melhor qualidade, que são produzidos em menos tempo, de forma mais econômica e com menos estresse, tanto para a equipe de desenvolvimento quanto para os clientes, que o habitual. A XP é organizada em torno de um conjunto de valores, princípios e práticas que atuam de forma harmônica e coesa. Além disso, é concisa, totalmente focada no desenvolvimento de software, pode ser aplicada em equipes de qualquer tamanho e se adapta bem a requisitos vagos e que mudam rapidamente.

A XP procura sempre assegurar que o cliente receba o máximo de valor por cada dia de trabalho da equipe de desenvolvimento. Sua adoção por uma equipe freqüentemente impacta fortemente na diminuição dos custos e no aumento da qualidade dos softwares desenvolvidos.

As considerações finais e conclusões são apresentadas no capítulo 4.

O apêndice A descreve o desenvolvimento dirigido por testes. Por ser uma prática muito importante para a XP e por ter resultados práticos facilmente perceptíveis, foi muito utilizada no estudo de caso desenvolvido em paralelo a este trabalho. Além disso, é uma prática muito sofisticada e efetiva. Por estes motivos, recebeu um destaque maior também do ponto de vista da fundamentação teórica.

O apêndice B descreve a refatoração que, apesar de não ser explicitamente citada como uma das práticas atuais, é fundamental para muitas práticas da XP, especialmente para o desenvolvimento guiado por testes. Por este motivo e por ser teoricamente muito rica e interessante, também recebeu um pequeno destaque dentro do trabalho.

O apêndice C descreve um estudo de caso que objetiva mostrar em funcionamento, alguns pontos da teoria abordada no trabalho. Este estudo de caso abrange parte do desenvolvimento de um sistema de gerenciamento de estoque e requisições de materiais, denominado SAM.

# 2. Desenvolvimento Ágil

Este capítulo descreve como surgiu, quais as vantagens e em que se baseia o desenvolvimento ágil de software. Este conteúdo é muito importante, porque esclarece muitas características comuns a todos os processos ágeis, incluindo, é claro, a XP, que apesar de ter sua identidade garantida por um conjunto rico de características próprias, em essência, é mais uma metodologia ágil.

Este capítulo também descreve o que está errado no modelo tradicional de desenvolvimento e porque é vantajoso abandoná-lo e adotar uma metodologia ágil.

Outros exemplos de metodologias ágeis são: SCRUM [SCHWABER, 2001], Adaptive Software Process [HIGHSMITH, 1999], Feature Driven Development (FDD) [PALMER, 2002], Crystal [COCKBURN, 2004] e Agile Modeling [AMBLER, 2002].

## 2.1. Desenvolvimento Tradicional

O termo **desenvolvimento tradicional**, neste trabalho, é utilizado para identificar as metodologias que de alguma forma adotam o desenvolvimento em cascata. No desenvolvimento em cascata, o software é construído seguindo uma seqüência de fases, sendo que cada fase, com exceção da primeira, depende da conclusão da fase anterior para ser iniciada (figura 2.1).

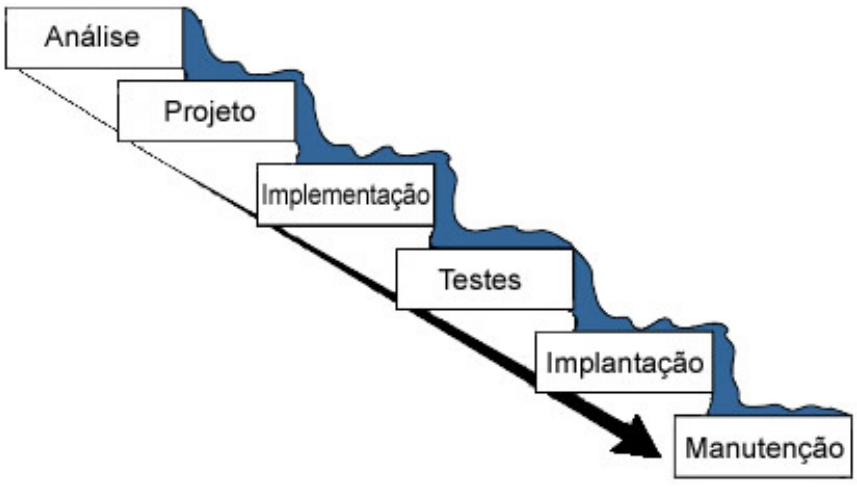


**Figura 2.1 - Desenvolvimento em cascata.**

O desenvolvimento em cascata, há várias décadas é amplamente empregado nos processos de desenvolvimento de software, e ainda hoje é o mais empregado. Por esse motivo, neste trabalho ele é chamado de **desenvolvimento tradicional**.

Mesmo alguns processos de desenvolvimento de software que não são em cascata, como o RUP (*Rational Unified Process*), comumente são utilizados seguindo uma seqüência de fases dentro de cada iteração. Ou seja, apesar do processo de desenvolvimento ser iterativo e incremental, ainda existe uma forte linearidade e especialização das atividades, caracterizada por um conjunto bem definido de fases executadas seqüencialmente dentro de cada iteração (figura 2.2). Neste trabalho, assim como em [TELES, 2004], o termo **desenvolvimento tradicional** engloba também estes casos.

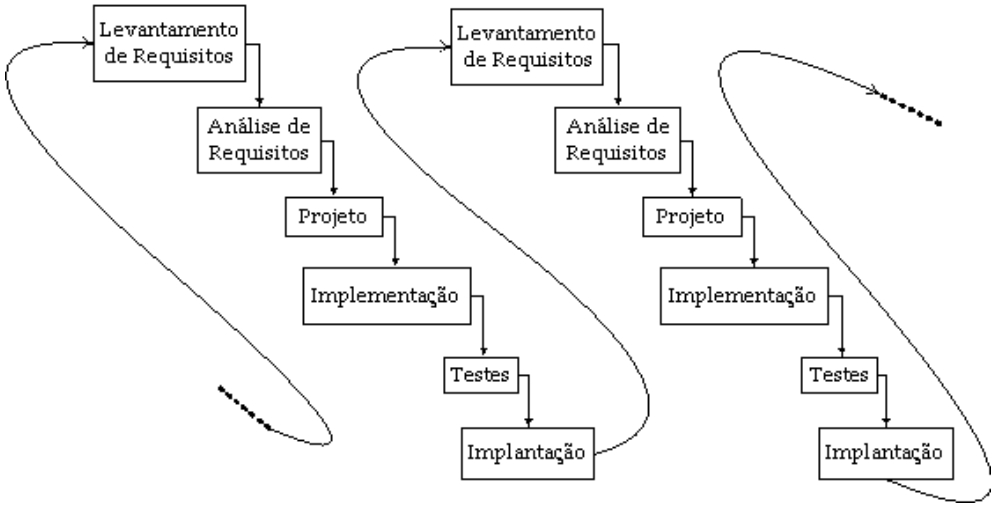


**Figura 2.2 - Desenvolvimento iterativo em cascata.**

## 2.2. Premissas do Desenvolvimento Tradicional

Existem algumas premissas básicas, derivadas no modelo de produção industrial proposto por Frederick Winslow Taylor no início do século XX, que são muito comuns no desenvolvimento tradicional e o influenciam profundamente. Segue abaixo estas premissas e seus significados dentro do contexto do desenvolvimento tradicional de software.

### 2.2.1 Linearidade

A linearidade fica evidente pela própria adoção de um modelo em cascata que, como já foi explicado, é linear por definição.

### 2.2.2 Determinismo

Um processo de desenvolvimento previamente conhecido por gerar determinados resultados com base nas especificações, gerará resultados semelhantes sempre que suas regras forem rigidamente seguidas. O desenvolvimento tradicional, assim como a indústria, persegue o determinismo por acreditar que esta é uma forma de reduzir erros e perdas de tempo [TELES, 2004].

### 2.2.3 Especialização

O processo de desenvolvimento pode ser desmembrado em atividades especializadas que serão executadas de forma independente e depois terão seus resultados integrados para formar o produto final. A especialização torna as tarefas mais simples e conseqüentemente facilita o determinismo. No desenvolvimento de software tradicional, a especialização aparece claramente na rígida divisão de papeis entre membros de uma equipe, como analistas, projetistas, programadores, testadores e implantadores. Cada um deles realiza tarefas bem diferentes e especializadas e terão seus resultados integrados para compor o software finalizado [TELES, 2004].

### 2.2.4 Foco na execução

Quando existe especialização suficiente para que as tarefas sejam simples e determinísticas, não existe a necessidade, por parte de quem executa o trabalho, de pensar sobre o que se está fazendo, basta fazer. Ou seja, basta que cada pessoa envolvida no processo de desenvolvimento execute corretamente a tarefa que lhe cabe, conforme rigidamente estipulado numa metodologia que presumivelmente torna o processo determinístico, para que a especificação seja transformada corretamente em software [DEMARCO, 1987, pg. 114; TELES, 2004]. Isto gera uma grande valorização dos processos e uma conseqüente desvalorização das pessoas, no desenvolvimento de um software [TELES, 2004].

### 2.2.5 Crescimento exponencial do custo de alteração

No modelo de desenvolvimento tradicional, em cascata, o custo de uma alteração cresce exponencialmente, à medida que o processo de desenvolvimento avança (figura 2.3). Esta premissa básica da engenharia de software tradicional foi formulada por Boehm em

*Software Engineering Economics* [BOEHM, 1981]. A aceitação desta premissa tem como conseqüência natural uma busca por processos determinísticos, já que estes prometem menos alterações e maior previsibilidade.

Este princípio também reflete e justifica de certa forma, a busca pela aproximação conceitual entre os processos de desenvolvimento de software e os processos industriais típicos, ou seja, justifica a busca por metodologia lineares, determinísticas, especializadas e focadas na execução.

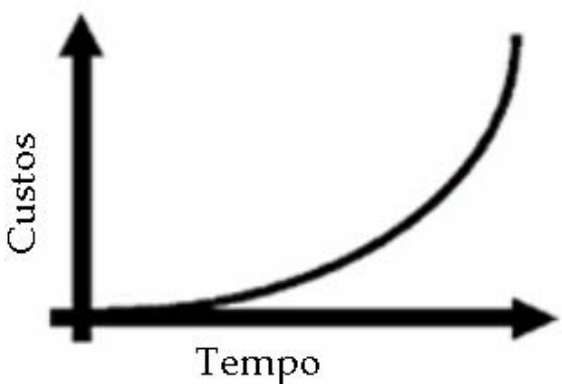


**Figura 2.3 - Crescimento dos custos de alteração**

Segundo Brooks [BROOKS, 1995, p. 264; TELES, 2004], isto não seria nenhum problema se o processo de desenvolvimento tradicional, em cascata, não estivesse errado nas seguintes afirmações:

- O modelo em cascata assume que o projeto irá passar por cada etapa apenas uma vez. Ou seja, assume que a arquitetura é excelente, que o *design* é correto e que a implementação pode ser corrigida durante os testes. Nenhuma destas afirmações funciona bem na prática, seja no modelo em cascata tradicional, seja nas metodologias iterativas e incrementais que empregam “cascatas menores” dentro de cada iteração (figura 2.2) [TELES, 2004].
- O modelo em cascata puro, assume que o sistema todo é construído de uma única vez. Isso implica que para poder testar todo o sistema é preciso que todas as fases anteriores, desde o levantamento dos requisitos até a implementação estejam concluídas e integradas num sistema único. Na prática, entre outros problemas

relacionados, os erros ocorrem em todas as etapas do desenvolvimento e deixar os testes para depois da implementação é algo altamente ineficiente [TELES, 2004].

## 2.3. Falhas do desenvolvimento tradicional

A grande falha do desenvolvimento tradicional é que ele baseia-se em premissas que não se aplicam ao trabalho de desenvolvimento de software.

Segundo Drucker [DRUCKER, 1999, p.111], existem duas categorias de trabalhadores: o **trabalhador manual** e o **trabalhador do conhecimento**. Um trabalhador manual executa atividades que dependem basicamente de habilidades manuais e que não se baseiam no uso intensivo do conhecimento. São atividades relativamente fáceis de automatizar por serem altamente determinísticas e repetitivas. O trabalhador do conhecimento, em contrapartida, é aquele que produz com base no uso intensivo da criatividade e do conhecimento [TELES, 2004].

Apesar de ser um trabalho executado basicamente por trabalhadores do conhecimento, o desenvolvimento tradicional utiliza premissas que só são válidas para o trabalhador manual.

Segundo Vinicius Teles [TELES, 2004, p. 38]: “Inúmeros estudiosos, entre eles Drucker, DeMarco, Lister, Toffler e DeMarsi têm demonstrado que a produtividade do trabalho do conhecimento deriva de fatores completamente diferentes daqueles usados para o desenvolvimento tradicional. De fato, eles mostram que as premissas da produtividade do trabalho do conhecimento são praticamente opostas às do trabalho manual e o grande erro cometido na maioria dos projetos de software é desconsiderar este fato e adotar as mesmas práticas do ambiente industrial.”.

Um exemplo da grande diferença em relação ao trabalhador manual e do conhecimento está na escolha da melhor política para tratar a ocorrência de erros. Para atividades que envolvem trabalhadores manuais, erros podem e devem sempre ser evitados, e os processos rígidos e determinísticos ajudam a alcançar este objetivo. Já para atividades que envolvem trabalhadores do conhecimento, erros devem ser encarados como coisas naturais, saudáveis e até inevitáveis. Neste caso, tentar sistematizar ou impor metodologias rígidas ao processo de desenvolvimento, visando o determinismo, no máximo torna as pessoas envolvidas defensivas e pouco criativas [TELES, 2004].

## 2.4. Resultados do Desenvolvimento Tradicional

Desde 1994, o *Standish Group International* publica a cada dois anos um estudo intitulado de *Chaos Research* [STANDISH, 2001] que consolida as informações de uma grande pesquisa sobre sucessos e fracassos dos projetos de software (figura 2.4). Neste estudo, os resultados dos projetos são enquadrados em uma das seguintes categorias:

- Bem sucedidos – O projeto é finalizado no prazo, dentro do orçamento e contendo todas as funcionalidades especificadas.
- Comprometidos – O projeto é finalizado e um software operacional é entregue, porém o orçamento e o prazo ultrapassam os limites estipulados, e, além disso, o software entregue possui menos funcionalidades do que o especificado.
- Fracassados – O projeto é cancelado em algum momento durante o desenvolvimento.

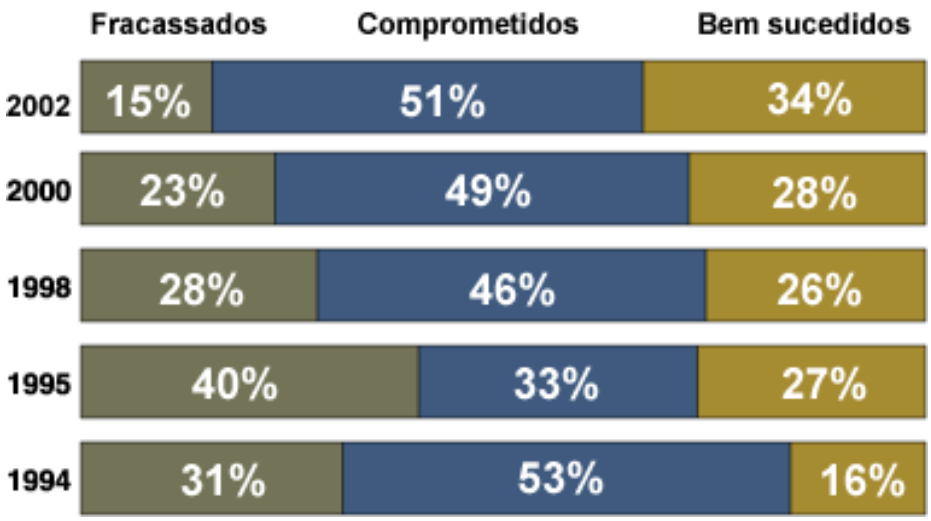


**Figura 2.4 – Resultados dos estudos *Chaos Research***

Este estudo é bastante abrangente, pois engloba uma grande quantidade de projetos com as mais diversas metodologias de desenvolvimento. Porém, apesar da diversidade de metodologias, a grande maioria delas é baseada no desenvolvimento tradicional. A figura 2.4 mostra que apesar de ter ocorrido um aumento substancial da porcentagem de projetos

bem sucedidos e diminuição de fracassados, os últimos resultados ainda são muito ruins, pois os projetos fracassados e comprometidos equivalem a 66% do total.

Como já foi dito neste trabalho, todas as metodologias tentam, entre outras coisas, reduzir o alto risco associado ao desenvolvimento de software. Porém, de acordo com os resultados alarmantes conseguidos nos últimos anos (figura 2.4), está claro que o desenvolvimento tradicional não tem conseguido atingir este objetivo. Estes resultados respaldam a afirmação de Brooks (seção 2.2.5) de que o desenvolvimento tradicional é equivocado.

## 2.5. Desenvolvimento Ágil

O desenvolvimento ágil é um fruto da constatação feita, de forma independente, por diversos profissionais renomados na área de engenharia de software, de que, apesar de terem aprendido segundo a cartilha tradicional, só conseguiam minimizar os riscos associados ao desenvolvimento de software, pensando e agindo de forma muito diferente do que tradicionalmente está nos livros. Estes profissionais, a maioria veteranos consultores para projetos de softwares, decidiram reunir-se no início de 2001 durante um *workshop* realizado em Snowbird, Utah, EUA.

Embora cada envolvido tivesse suas próprias práticas e teorias preferidas, todos concordavam que, em suas experiências prévias, os projetos de sucesso tinham em comum um pequeno conjunto de princípios. Com base nisso eles criaram o Manifesto para o Desenvolvimento Ágil de Software, freqüentemente chamado apenas de Manifesto Ágil. O termo Desenvolvimento Ágil identifica metodologias de desenvolvimento que adotam os princípios do manifesto ágil. Estes princípios são os seguintes:

- **Indivíduos e interação entre eles** mais que processos e ferramentas
- **Software em funcionamento** mais que documentação abrangente
- **Colaboração com o cliente** mais que negociação de contratos
- **Responder a mudanças** mais que seguir um plano

O manifesto reconhece o valor dos itens à direita, mas diz que devemos valorizar bem mais os itens à esquerda.

No desenvolvimento ágil, os projetos adotam o processo iterativo e em espiral (figura 2.5). No modelo espiral a dimensão radial representa o custo acumulado atualizado e a dimensão angular representa o progresso através da espiral.

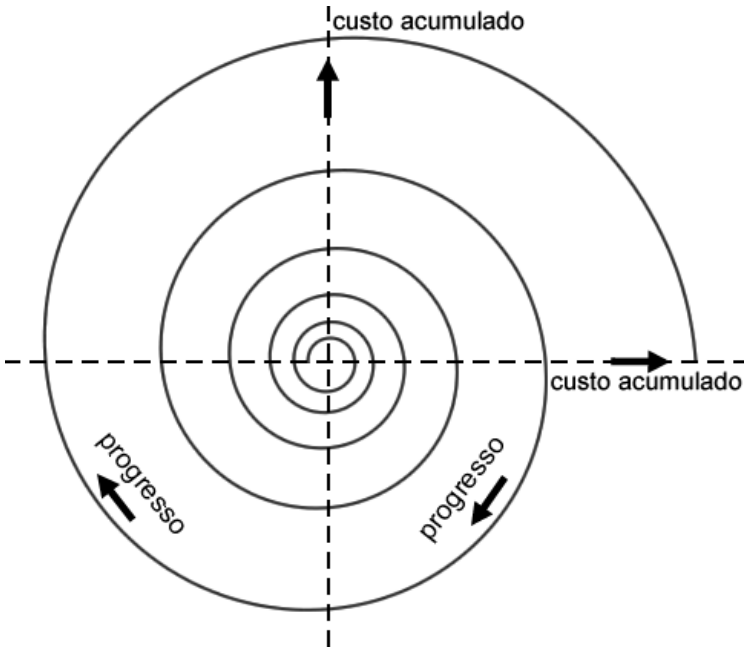


**Figura 2.5 - Desenvolvimento iterativo em espiral**

No desenvolvimento ágil cada setor da espiral corresponde a um ciclo de desenvolvimento, sendo que estes ciclos devem ser curtos e ter tamanho(tempo) fixo.

Cada ciclo é precedido por uma retrospectiva que objetiva manter um processo de evolução contínua com base no aprendizado adquirido até o momento através dos ciclos anteriores.

Os ciclos são chamados de iterações e sempre acrescentam valor ao projeto na forma de novas funcionalidades implementadas, testadas e aprovadas. Isto permite que o cliente acompanhe na prática e gradativamente a evolução do projeto, obtenha muito mais cedo os benefícios do sistema, avalie a evolução do projeto e dê *feedback* constante para os desenvolvedores sobre as funcionalidades implementadas [TELES, 2004].

## 2.6. Premissas do Desenvolvimento Ágil

As abordagens ágeis partem de premissas freqüentemente muito diferentes das tradicionalmente adotadas. Estas premissas levam em consideração outros pontos de vista e justamente por isto parecem não fazer sentido, quando analisadas segundo a ótica do desenvolvimento tradicional. A maior parte delas são novas e muitas vezes opostas às adotadas tradicionalmente. Entender estas premissas é muito importante, pois elas justificam o porque da maioria das práticas ágeis.

### 2.6.1 O cliente aprende ao longo do desenvolvimento

Um dos problemas mais complexos que afetam o desenvolvimento de software é a enorme quantidade de detalhes que precisam ser considerados. Normalmente, ao especificar um sistema, o cliente tem o conhecimento de alguns aspectos do software que deseja. Entretanto, muitos outros só ficam claros quando ele tem a oportunidade de utilizar o sistema. Portanto, o cliente não especifica estes detalhes no início do projeto por uma razão muito simples: ele não os conhece [BECK, 2000]. Além do mais, mesmo que tais detalhes fossem conhecidos previamente, seria muito difícil especificá-los através de documentos, devido à grande quantidade de elementos que precisariam ser descritos.

Perceber que o cliente aprende ao longo do desenvolvimento é a chave para se compreender o grande desafio existente no desenvolvimento de software. O cliente aprende à medida que tem a oportunidade de manipular o sistema e à medida que se envolve em discussões sobre ele. Este aprendizado pode ser utilizado para realimentar o processo de desenvolvimento ou pode ser ignorado. Esta última alternativa é o caminho adotado normalmente no desenvolvimento tradicional, pois ele parte da premissa que o cliente já sabe o que quer no início do projeto e a ele cabe apenas descrever os requisitos. Depois, ao final, ele receberá o sistema e não haverá ciclos de realimentação entre a especificação e a entrega do sistema [TELES, 2004].

### 2.6.2 O custo de uma alteração tende a se estabilizar

Ao contrário do desenvolvimento tradicional que acredita que o custo de uma mudança cresce exponencialmente à medida que o tempo de desenvolvimento avança, no desenvolvimento ágil acredita-se que o custo de mudança do software ao longo do tempo tende a se tornar constante (figura 2.6). Tal idéia se fundamenta principalmente nas vantagens da forma como o processo iterativo e em espiral é utilizado nas metodologias ágeis, mas também se beneficia de vários outros fatores, como avanços ocorridos na microinformática, adoção da orientação a objetos, uso da refatoração para aprimorar e simplificar o design, adoção de testes automatizados, melhores linguagens e ambientes de desenvolvimento [TELES, 2005].

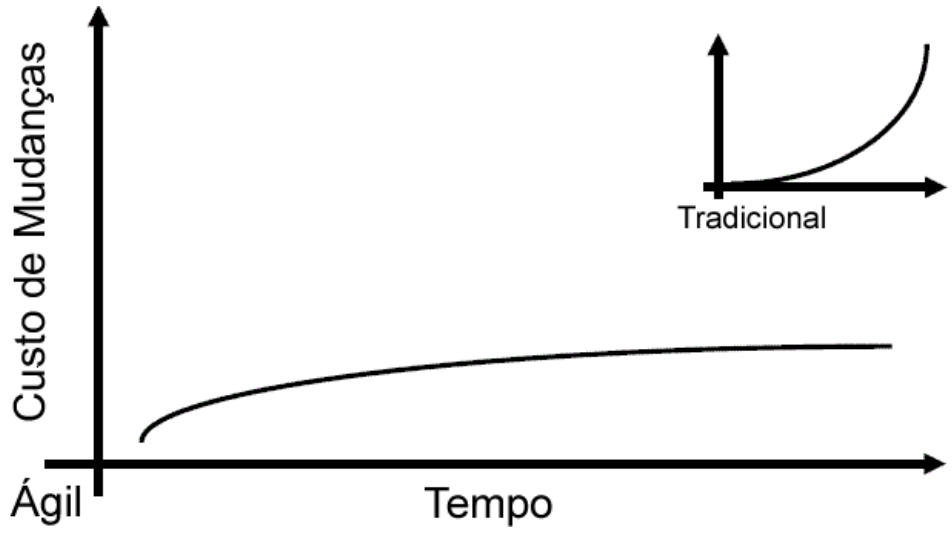


**Figura 2.6 – Custo de alterações no desenvolvimento ágil e no desenvolvimento tradicional.**

As duas premissas anteriores ajudam a explicar algumas das posições totalmente opostas adotadas pelos dois modos de desenvolvimentos. Por exemplo, no desenvolvimento ágil, modificações no projeto são naturais, pois não existe nenhum custo exponencial associado às alterações. Ou seja, não existe nenhuma necessidade de tentar especificar detalhadamente tudo que ocorrerá durante a implementação do sistema, para minimizar possíveis alterações, até porque, isto dificilmente traz os resultados esperados.

## 2.7. Resultados do Desenvolvimento Ágil

Nos anos de 2006 e 2007, Scott Ambler organizou pesquisas para medir a adesão dos profissionais e das empresas aos valores, princípios e práticas comumente usadas no paradigma ágil. Os pesquisados tiveram que responder um questionário com perguntas relacionadas ao objetivo da pesquisa. Os resultados das duas pesquisa foram publicados na revista Dr. Dobb's [DOBBS, WEB].

Os resultados mostram, entre outras coisas, que a grande maioria das empresas pesquisadas já adota técnicas ágeis, e que um adicional de aproximadamente 7% pretende adotar em no máximo 1 ano (figura 2.7).

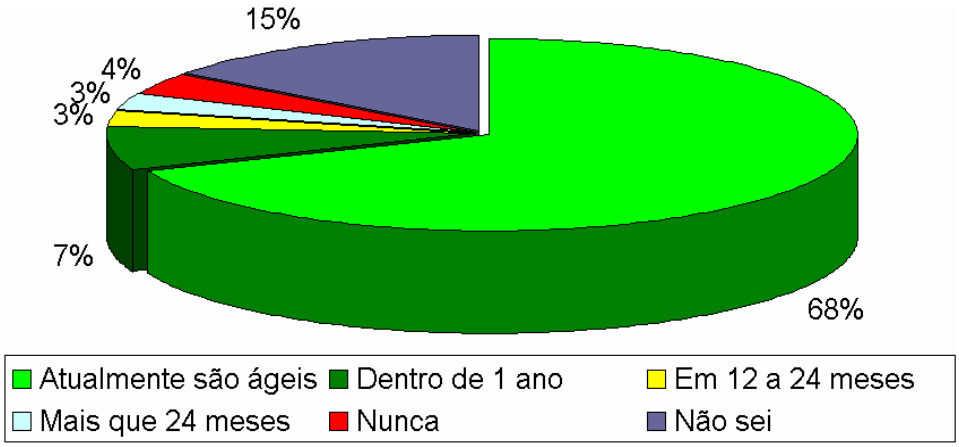


**Figura 2.7 - Taxa de adoção de técnicas ágeis pelas organizações**

Outra constatação muito importante é que a adoção de técnicas ágeis não está restrita a projetos pilotos, isto fica claro na figura 2.8 que mostra a quantidade de projetos ágeis em andamento por organização.

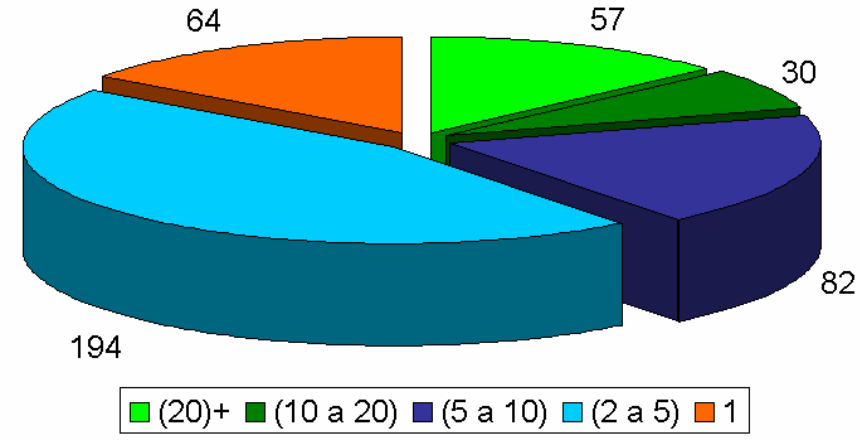


**Figura 2.8 – Número de projetos ágeis em andamento**

Os dados da pesquisa de 2007 também mostram claramente que as técnicas ágeis foram implantadas com sucesso na maioria das empresas pesquisadas. A figura 2.9 mostra a porcentagem global de sucesso nos projetos. Neste caso não houve na pergunta uma definição formal de sucesso, já que esta definição em projetos de TI costuma variar por organização e freqüentemente até mesmo por projeto.

As fatias do gráfico indicam as porcentagens de pessoas que acreditam que seus projetos ágeis estão dentro da faixa de sucesso representada pela cor. A faixa de sucesso que cada cor representa está descrita na legenda.

Ex.: 77%(verde claro + verde escuro) das pessoas pesquisadas indicaram que seus projetos ágeis tiveram mais que 75% de sucesso.

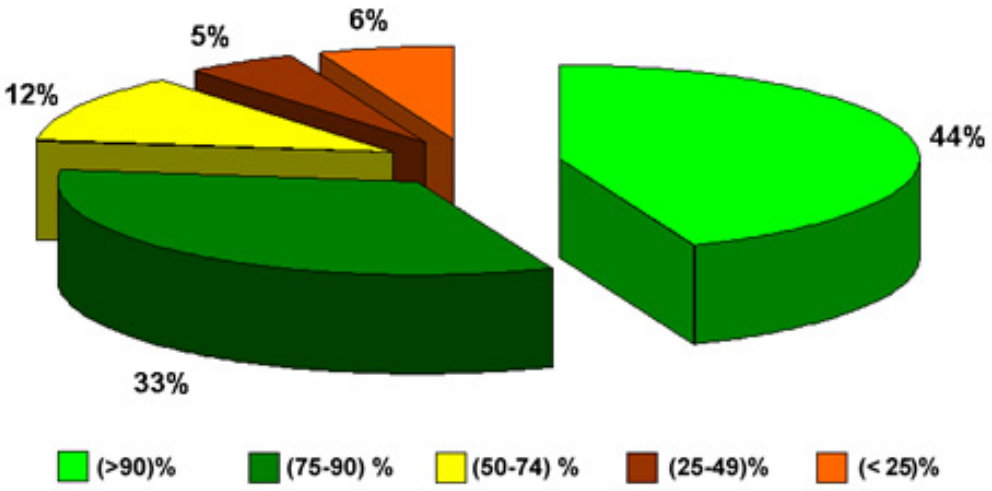


**Figura 2.9 – Porcentagem global de sucesso nos projetos**

Apesar da diferença de tamanho da amostra, em relação ao *Chaos Research* [STANDISH, 1994], os resultados mostrados na figura 2.9 são muito significativos e esclarecedores, pois dão uma boa idéia da grande diferença que existe em termos de resultados, entre os processos ágeis e os tradicionais. A pesquisa, feita através da Internet, recebeu respostas de 781 pessoas da área de TI, sendo 52% desenvolvedores e 22% gerentes, em março de 2007 [DOBBS, WEB; AMBLER, WEB].

Com base nos dados mostrados, fica claro que as metodologias ágeis, seis anos após seu surgimento, estão deixando de ser algo incerto, adotado somente por empresas jovens e com cultura fortemente voltada para inovação. O grande crescimento dos projetos que usam

técnicas ágeis, em número e tamanho, nas mais variadas organizações, incluindo as grandes e tradicionais, mostra que o desenvolvimento ágil está tornando-se rapidamente a opção comum da maioria das empresas para projetos de software.

## 2.8. Conclusões

No desenvolvimento ágil os envolvidos no desenvolvimento são tratados como trabalhadores do conhecimento e conseqüentemente são estimulados a aprender durante o desenrolar do próprio trabalho e tomar decisões melhores com base neste aprendizado. Não existe um processo rígido que impõe o que pode ou não ser feito e desestimula a criatividade.

No desenvolvimento tradicional, os especialistas do negócio conversam com os analistas, que digerem, abstraem e passam as informações mastigadas aos programadores, que não pensam nada além do necessário para escrever o código do software. Esta especialização (seção 2.2.3) de atividades é péssima porque é completamente desprovida de *feedback*. O analista tem toda a responsabilidade de criar o modelo de domínio, baseado somente nas informações fornecidas pelos especialistas do negócio. Não existe a oportunidade de aprender com o desenvolvimento ou ganhar experiência com as versões iniciais do software [EVANS, 2004].

O desenvolvimento ágil sabe que trabalhadores do conhecimento rendem mais e melhor em ambientes que estimulam o uso intensivo da criatividade e do conhecimento. Escrever softwares é um trabalho tão ou mais criativo que escrever um livro, um artigo ou uma monografia. É uma atividade tipicamente intelectual, onde é muito comum ocorrerem idas e vindas, já que o aprendizado adquirido com o seu desenrolar torna possível aos envolvidos perceberem maneiras cada vez melhores de fazerem as coisas. Este tipo de trabalho – não linear – funciona muito melhor de forma iterativa e incremental (em espiral).

# 3. Extreme Programming

Extreme Programming ou simplesmente XP, é a mais conhecida das abordagens para desenvolvimento de software que obedecem aos princípios do Manifesto Ágil. É uma metodologia que vem fazendo muito sucesso em diversos países, por ajudar a criar sistemas de melhor qualidade, que são produzidos em menos tempo, de forma mais econômica e com menos estresse para os desenvolvedores e clientes que o habitual.

A XP é fruto de um processo de evolução e junção de bons princípios e práticas de desenvolvimento de software. Um marco importante em seu processo de criação deu-se em 1996, ano em que Kent Beck começou a trabalhar no desenvolvimento de um grande sistema de folha de pagamento, chamado *Chrysler Comprehensive Compensation* (Sistema de Compensação Abrangente da Crysler), ou C3. Apesar do longo processo de criação e evolução dos princípios e práticas usados da XP, foi durante o desenvolvimento do C3 que Kent Beck pela primeira vez os reuniu e aplicou de forma harmônica e coesa num grande projeto. Além disso, foi durante este projeto que tanto a metodologia quanto as práticas iniciais foram batizadas com os nomes atuais [TELES, 2004].

A XP concentra os esforços da equipe de desenvolvimento em atividades que geram resultados rapidamente na forma de software intensamente testado e alinhado às necessidades de seus usuários. Além disso, simplifica e organiza o trabalho combinando técnicas comprovadamente eficazes e eliminando atividades redundantes. Por fim, reduz o risco dos projetos desenvolvendo software de forma iterativa e reavaliando permanentemente as prioridades dos usuários [TELES, 2004].

Cada vez mais as empresas convivem com ambientes de negócios que requerem mudanças freqüentes em seus processos, as quais afetam os projetos de software. Os processos de desenvolvimento tradicionais são caracterizados por uma grande quantidade de atividades e artefatos que buscam proteger o software contra mudanças, o que faz pouco ou nenhum sentido, visto que os projetos devem se adaptar a tais mudanças ao invés de evitá-las [TELES, 2004]. A XP encara mudanças com muita naturalidade, pois uma de suas premissas diz que a única certeza que se pode ter em relação a um projeto de software é que muitas mudanças ocorrerão.

A XP valoriza extremamente a construção de interações sociais boas e confiáveis, “XP é sobre mudança social” [BECK, 2005]. É uma metodologia leve, voltada para times de qualquer tamanho, focada no desenvolvimento de software e que se adapta bem em face a requisitos vagos e que mudam rapidamente [BECK, 2005]. Possibilita a criação de software de alta qualidade, de maneira ágil, econômica e flexível. Vem sendo adotado com enorme sucesso nos Estados Unidos, na Europa e, mais recentemente, no Brasil. É composta por um conjunto de valores, princípios e práticas, que serão detalhados nas próximas seções, e difere substancialmente da forma como se desenvolve software nos projetos tradicionais.

## 3.1. Valores, Princípios e Práticas

Práticas são claras e concretas; ou se escreve um teste antes de mudar o código ou não. Também indicam ações que normalmente são específicas para certos contextos; refatoração é aplicada somente a trechos de código que estão funcionalmente corretos. Além disso, são úteis porque dão um lugar para começar; pode-se começar escrevendo os testes e ganhar os benefícios de se fazer assim, muito antes de entender o desenvolvimento de software de um modo mais profundo [TELES, 2004; TELES, 2005; IMPROVE IT, XP; BECK, 2005].

Antes de explorar as práticas da XP é fundamental conhecer bem os valores e princípios que justificam e direcionam o uso destas práticas. Entender bem estes valores e princípios é importante para obter-se uma compreensão das forças que, em conjunto, devem mover a equipe de desenvolvimento.

Valores são critérios gerais e abstratos usados para justificar o que se vê, pensa ou faz. Princípios servem de ponte entre os valores e as práticas (figura 3.1).

Valores representam a essência daquilo que gostamos ou não a respeito de alguma coisa. Ter os valores explícitos é importante porque sem valores, as praticas rapidamente perdem o sentido e tornam-se atividades feitas por fazer, sem qualquer propósito ou direção. Valores trazem propósito às práticas.

No entanto é preciso tomar cuidado, pois os valores são expressos num nível tão alto que se pode fazer qualquer coisa em nome deles. "Eu escrevi este documento de mil páginas porque valorizo a comunicação". Isto é muito relativo, pois manter conversas de

quinze minutos uma vez por dia pode ser algo mais produtivo do que produzir um longo documento, ou seja, documentação longa pode ser desvalorizar a comunicação. Comunicar-se do modo mais efetivo é uma forma de valorizar a comunicação [TELES, 2004; TELES, 2005; IMPROVE IT, XP; BECK, 2005].

Os princípios são direcionados a domínios específicos e foram criados para servir de ponte entre os valores e as práticas (veja figura 3.1). Valores são abstratos demais para guiar práticas (comportamentos) em contextos específicos. Tanto longos documentos quanto diálogos têm a intenção de comunicar, então qual o mais efetivo? A resposta depende em parte do contexto e em parte de princípios intelectuais. O princípio da humanidade sugere que diálogos atendem à necessidade humana de conexão, portanto é a forma de comunicação preferível, sem levarmos em conta outros fatores [IMPROVE IT, XP].

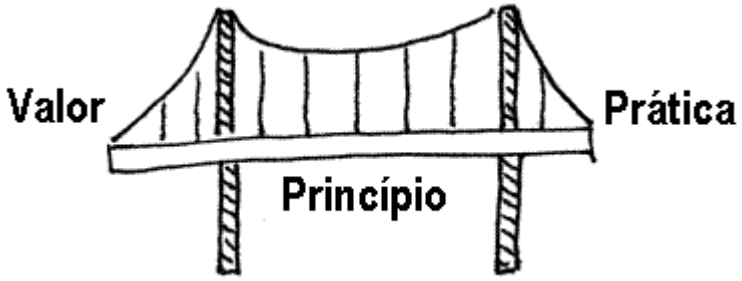


**Figura 3.1 – Um princípio atua como uma ponte.**

Em conjunto, os Valores, Princípios e Práticas da XP, tentam comunicar claramente, o que é necessário pensar e praticar no desenvolvimento de software. Apesar disso, ler e entender tudo sobre eles, mesmo no melhor dos livros, não é suficiente. É essencial obter experiência, programando no estilo extremo, participando de comunidades de pessoas que compartilham estes valores, princípios e pelo menos algumas de suas práticas, e compartilhando o aprendizado com outros.

A seguir, serão detalhados os valores, princípios e práticas da XP.

## 3.2. Valores

Os valores dão um norte para o comportamento da equipe de desenvolvimento. Eles mostram o que é importante para a equipe e conseqüentemente para seus membros. O mais importante é alinhar o comportamento da equipe aos seus valores, e assim, minimizar o desperdício gerado ao tentar-se manter múltiplos conjuntos de valores simultaneamente [TELES, 2004; IMPROVE IT, XP; BECK, 2005].

A XP se baseia em cinco valores para guiar o desenvolvimento de software: *feedback*; comunicação; simplicidade; coragem e respeito.

Vale destacar que estes não são os únicos valores possíveis para o desenvolvimento de software. Eles são os valores essenciais, mas uma organização, um time ou uma pessoa, pode escolher outros [BECK, 2005].

### *3.2.1 Feedback*

*Feedback*, basicamente, é o processo de troca de informações relacionadas ao que é produzido e consumido durante um projeto, entre o cliente e a equipe de desenvolvimento e vice-versa. Este processo de troca não foi criado com a XP. Ele existe no desenvolvimento tradicional, mas com uma grande diferença em relação aos tempos de realização, pois no desenvolvimento tradicional é normal existir uma grande defasagem entre os ciclos de produção e consumo [TELES, 2004].

A filosofia da XP e dos processos ágeis de desenvolvimento, de uma forma geral, se baseia fortemente no *feedback* baseado em ciclos curtos, o que garante que pouco trabalho seja efetuado e concluído por ciclo. Assim, caso surjam falhas, elas estarão contidas num escopo menor e mais simples, conseqüentemente serão mais fáceis de localizar e corrigir.

Definir corretamente quais são as necessidades dos usuários é uma das atividades mais difíceis e importantes de serem realizadas pelos desenvolvedores de um software, pois ela direciona todos os demais esforços.

O *feedback* é a realimentação que o **cliente** fornece a equipe de desenvolvimento, em relação ao que está sendo produzido. Através do *feedback* contínuo, a equipe de desenvolvimento, e também o próprio cliente, pode avaliar e aprender continuamente sobre os verdadeiros requisitos do sistema. “Um princípio psicológico bem conhecido indica que

para maximizar a taxa de aprendizado, a pessoa precisa receber *feedback* sobre quão bem ou mal ele está indo" [WEINBERG, 1971; TELES, 2005].

Analogamente, o *feedback* também reflete a realimentação que os **desenvolvedores** fornecem ao cliente através da apresentação de estimativas, riscos técnicos, alternativas de *design*, entre outras coisas. Esta manifestação do aprendizado da equipe de desenvolvimento sobre aquilo que precisa ser feito pode se dar verbalmente, mas geralmente ocorre através de uma nova versão do próprio software, já que assim os usuários aprendem mais facilmente.

Através do *feedback*, o cliente aprende como o sistema pode contribuir da melhor forma possível com ele próprio, e realimenta a equipe de desenvolvimento com este aprendizado. A equipe de desenvolvimento também aprende, podendo então aplicar o aprendizado no aperfeiçoamento do projeto, e depois realimentar o usuário com algo um pouco mais alinhado com as reais necessidades deste. Quanto mais se repetir este ciclo, mais o software irá convergir para um produto final que atenda as necessidades reais de seus usuários.

### 3.2.2 Comunicação

Uma das coisas mais importantes para o desenvolvimento de software é a comunicação. Seja entre os membros da própria equipe ou entre estes e os clientes, a comunicação é uma das principais ferramentas para a solução de problemas durante o desenvolvimento. Quando um problema aparece, freqüentemente alguém próximo à equipe já sabe a solução, mas numa equipe que não se comunica bem, é comum o conhecimento não chegar até onde está o problema. Além disso, para que o valor *feedback* exista em um projeto de software, é necessário que a comunicação esteja presente de forma muito intensa [TELES, 2004; IMPROVE IT, XP; BECK, 2005].

A XP procura assegurar que a comunicação ocorra da forma mais efetiva possível. Sendo assim, ela busca sempre estimular a forma mais rica de comunicação que estiver disponível.

Para a XP a forma ideal de comunicação é a face-a-face, pois o interlocutor pode interpretar as idéias levando em conta o conteúdo emocional da fala, dos gestos, da

expressão facial, entre outros. Além disso, em caso de dúvida, o interlocutor pode apresentar suas questões rapidamente, às quais serão respondidas logo em seguida. Ou seja, a XP procura explorar, tanto quanto possível, a interação direta entre as pessoas, de modo a diminuir as falhas de comunicação e evitar re-trabalhos desnecessários. Para que isso seja possível, é preciso aproximar ao máximo todas as pessoas que formam a equipe do projeto, inclusive e sobretudo o cliente [TELES, 2004; IMPROVE IT, XP; BECK, 2005].

A comunicação técnica escrita é muito pobre, principalmente por ser fria, ou seja, desprovida de conteúdo emocional, e por dificultar a interação rápida e continua entre todos os participantes do projeto, prejudicando assim a criação de um senso de time e a cooperação [TELES, 2004].

Na indústria de software existe um grande esforço para melhorar a comunicação dentro dos projetos. Entretanto, em grande parte dos casos, este esforço é canalizado de forma ineficaz, já que boa parte se destina à comunicação escrita, mesmo esta sendo uma das formas mais pobres de comunicação.

Mesmo não produzindo tantos documentos, principalmente em papel, quanto no modelo tradicional, a documentação continua sendo muito importante na XP. A diferença é que a documentação exerce um papel diferente. No desenvolvimento tradicional a documentação tem um caráter **prescritivo**, pois sua principal preocupação é comunicar de forma muito detalhada o que deverá ser feito pelos desenvolvedores. Já na XP, a documentação tem um caráter **descritivo**, pois o papel principal é ser um instrumento para registro do trabalho, e não um instrumento para **comunicar** requisitos. Uma forma de documentação descritiva muito valorizada pela XP é feita no próprio código fonte. Identificadores com nomes apropriados, código limpo, testes de aceitação e unidade, design simples e bem feito, fazem com que o código dependa muito pouco ou nada de outros tipos de documentos [TELES, 2004].

### 3.2.3 Simplicidade

O valor simplicidade é um dos mais sofisticados da XP, pois exige muito conhecimento técnico e bom senso. Ele procura sempre manter o projeto o mais simples possível, tornando-o ágil e maleável.

A simplicidade nos ensina a implementar apenas aquilo que é suficiente para atender as necessidades atuais do cliente. Ou seja, ao codificar uma funcionalidade o desenvolvedor deve se preocupar apenas com os problemas de hoje e deixar os problemas do futuro para o futuro. Não se deve tentar prever o futuro, pois raramente obter-se-á êxito nas previsões [TELES, 2004]. Fazer um sistema simples o bastante para resolver elegantemente somente os problemas do momento é algo muito difícil que depende muito dos valores comunicação e *feedback*, pois sem eles é impossível obter corretamente os requisitos necessários para o momento atual.

Ao contrário do desenvolvimento tradicional, a XP acredita que é muito bom adiar decisões, pois quanto mais tarde se toma uma decisão, mais informações confiáveis existirão para embasá-la. Por isso que os requisitos vão sendo detalhados e até definidos no decorrer do projeto com base no *feedback* de todas as partes envolvidas. Porém, para isto funcionar bem, é essencial que a equipe compreenda e utilize o valor simplicidade, que antecipa e aumenta a freqüência de produção de resultados, e conseqüentemente o contato do usuário com o sistema [IMPROVE IT, XP; ASTELS, 2002; BECK, 2005].

Um estudo recente e revelador, que mostra a dificuldade em se prever que funcionalidades devem ser implementadas ou não em um sistema, foi apresentado pelo presidente do *Standish Group*, Jim Johnson, na terceira Conferência Internacional sobre *Extreme Programming*, ocorrida na Itália em 2002. Os resultados demonstram que o grau de utilização (36%) das funcionalidades (figura 3.2) colocadas em produção é baixíssimo, ou seja, grande parte das funcionalidades (64%), se não fossem implementadas, causariam pouquíssimo impacto no sistema como um todo [JOHNSON, 2002].

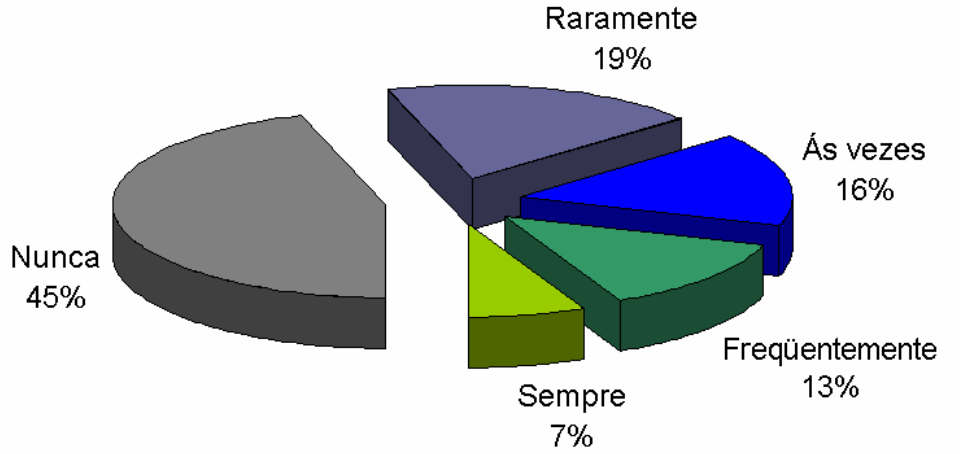


**Figura 3.2 – Freqüência de utilização de funcionalidades**

Outro estudo do *Standish Group*, realizado em 1994, fruto de uma pesquisa com 350 empresas e 8.000 projetos de software, revelou que nas grandes companhias americanas, mesmo entre os poucos (9%) projetos entregues dentro do prazo e do orçamento, somente 42% dos requisitos definidos inicialmente são encontrados no produto final [STANDISH, 1994]. Isto revela claramente o quanto é arriscado investir uma grande quantidade de recursos para detalhar antecipadamente os requisitos de um sistema.

O valor Simplicidade também evita um erro muito freqüente no desenvolvimento de software, chamado trabalho especulativo. O trabalho especulativo é aquele que é executado utilizando premissas sobre as quais não se tem total certeza. Ele ocorre quando o desenvolvedor implementa por conta própria uma determinada funcionalidade diante de uma dúvida. Um exemplo ainda mais perigoso de trabalho especulativo é quando o desenvolvedor assume requisitos futuros, mesmo quando no momento atual, estes requisitos não são visíveis ao usuário. Neste caso, o desenvolvedor implementa a funcionalidade de forma “genérica” para que ela possa se adaptar a todas as futuras necessidades do cliente. Em inglês isto é chamado de *overengineering* e significa criar uma solução excessivamente sofisticada para um dado problema [TELES, 2004].

O valor simplicidade só faz sentido em um contexto, por isso, a solução simples de ontem pode tornar-se simplista ou complexa à medida que a aplicação evoluir. Neste caso, é essencial recuperar a simplicidade com base no contexto atual. O principal objetivo do valor simplicidade é, portanto, evitar o trabalho extra que resulta do desconhecimento ou da precipitação. Implementar uma funcionalidade com mais detalhes que o necessário é especular sobre as necessidades do cliente. Neste caso, somente o *feedback* do cliente poderá dizer se a parcela especulativa do código é mesmo necessária. Tendo consciência disso, percebemos que é preferível evitar o trabalho especulativo implementando as funcionalidades sempre da forma mais simples possível [TELES, 2004].

### 3.2.4 Coragem

Ter coragem é agir de forma efetiva em face do medo. Certamente é verdade que as pessoas envolvidas no desenvolvimento de software sentem medo. É a forma como elas lidam com seus medos que vai influenciar na qualidade do trabalho ou não.

Às vezes a coragem manifesta-se como uma propensão a uma ação; se você souber qual é o problema, faça algo sobre ele. Às vezes a coragem manifesta-se como paciência; se você sabe que há um problema mas não sabe a solução, é preciso coragem para esperar o problema tornar-se claro [BECK, 2005].

Os valores da XP trabalham melhor em conjunto e em muitos casos são essenciais uns aos outros. O valor *feedback* é melhor com comunicação e simplicidade. O valor simplicidade é melhor com comunicação e *feedback*. O valor comunicação é melhor com simplicidade e *feedback*. Com o valor coragem não é diferente. Utilizar o valor coragem de forma isolada é agir sem se preocupar com as conseqüências, o que é muito perigoso e desestimulante para o trabalho em equipe.

Para diminuir riscos e encorajar o trabalho em equipe é recomendável sempre considerar todos os valores, não somente a coragem, na hora de tomar uma decisão numa situação em que o medo estiver presente.

Se a coragem sozinha é perigosa, em harmonia com os outros valores é poderosa. A coragem para falar verdades, agradáveis ou desagradáveis, estimula a comunicação e a confiança. A coragem para descartar soluções fracassadas e buscar novas, estimula a simplicidade. A coragem para sempre buscar respostas concretas cria *feedback* [BECK, 2004].

A XP é uma metodologia de software nova que se baseia em diversas premissas. Algumas destas premissas já foram detalhadas nesta monografia, as outras estão detalhadas nos próximos capítulos. A maioria das premissas são novas e contrariam os processos tradicionais de desenvolvimento, por isto mesmo, é preciso ter coragem para adotá-las. Segue abaixo uma listagem [TELES, 2004, pg. 50] das principais premissas da XP:

- Desenvolver o software de forma incremental
- Manter o sistema simples
- Permitir que o cliente priorize as funcionalidades
- Fazer os desenvolvedores trabalharem em par
- Investir tempo em *refactoring*
- Investir tempo em testes automatizados
- Estimar as histórias na presença do cliente
- Expor o código a todos os membros da equipe

- Integrar o sistema diversas vezes ao dia
- Adotar um ritmo sustentável
- Abrir mão de documentações que servem como defesa
- Propor contratos de escopo variável
- Propor a adoção de um processo novo.

É ingenuidade achar que bons profissionais não sentem medo. Coragem e medo não são mutuamente exclusivos, pelo contrário, coragem e medo costumam andar sempre lado a lado. Coragem tem a ver com fazer o que deve ser feito. Ausência de medo é um sinônimo para imprudência. Como diz Kent Beck, “Medo é o jeito da natureza de dizer, ‘seja cuidadoso!’” [BECK, 2005].

### 3.2.5 Respeito

No desenvolvimento de software com XP, todas as pessoas são igualmente importantes como seres humanos que são. Nenhum membro da equipe vale em essência, mais ou menos do que os outros. Para uma melhora simultânea na humanidade e na produtividade, as contribuições de cada pessoa da equipe precisam ser respeitadas [BECK, 2005].

Basta que apenas um membro de uma equipe deixe de respeitar, devidamente, algum outro membro para que ocorram prejuízos. Se por exemplo, membros de uma equipe agem como se o papel que exercem fosse mais importante do que o de outros membros, acabarão não respeitando o que os outros estão fazendo, conseqüentemente não se interessarão pelo verdadeiro trabalho em equipe.

O valor Respeito atua como um alicerce para todos os outros valores, por isso ele sempre deve estar presente.

Se todos na equipe não são tratados de forma igual e não se preocupam uns com os outros, então os membros desta equipe não buscarão ***feedback*** com a intenção de aprimorar, não tentarão **comunicar-se** da forma mais efetiva possível, não tentarão fazer o sistema da forma mais **simples** possível, não terão **coragem** para agir efetivamente ou

aguardar a hora certa de agir. Ou seja, sem respeito todos os valores da XP são prejudicados.

Respeito é o mais básico de todos os valores e é indispensável para que a XP funcione. Se ele não existir em um projeto, não há nada que possa salvá-lo [IMPROVE IT, XP].

## 3.3. Princípios

Os valores não fornecem conselhos concretos sobre qual o melhor caminho a ser seguido. Valores dizem aonde chegar e dão razões para isto. As práticas são ações concretas que objetivam facilitar um percurso, mas sem deixar claro o porquê de sua importância. Valores são abstratos demais para guiar práticas (comportamentos) em contextos específicos [IMPROVE IT, XP; BECK, 2005].

Por causa dessa distância entre os valores e as práticas, é necessário algo para aproximá-los. Este é o papel dos princípios. Os princípios são diretrizes direcionadas para domínios específicos, que têm o objetivo de fortalecer o vínculo entre os valores e as práticas [IMPROVE IT, XP; BECK, 2005].

Esta seção dará uma visão geral de cada um dos princípios da XP e tem o objetivo de ser um recurso a mais no entendimento das práticas que são detalhadas na sessão 3.4. Além disso, estes princípios podem ser usados para improvisar práticas complementares quando não for possível encontrar nenhuma prática que satisfaça ao que se deseja [BECK, 2005].

### 3.3.1 Humanidade

Softwares são desenvolvidos por pessoas e para pessoas, logo uma melhor compreensão das pessoas – como elas trabalham individualmente e em equipe, por exemplo – e das questões humanas em geral, é fundamental para a criação e evolução deste produto. Um bom software é resultado da ação de pessoas. Assim como é o caso de softwares ruins [CONSTANTINE, 2001].

Freqüentemente, o desenvolvimento de software não reconhece as fragilidades humanas, nem satisfaz suas necessidades. Isto é o mesmo que ignorar a importância das pessoas para a qualidade de um software [TELES, 2005; BECK, 2005].

Existem alguns fatores que afetam diretamente a qualidade do trabalho de um desenvolvedor:

- Segurança básica – não se sentir ameaçado, inclusive em relação a manutenção do emprego;
- Realização – sentir-se útil, valorizado e orgulhoso com o trabalho realizado;
- Participação – identificar-se com um grupo e dentro deste, assumir responsabilidades e ajudar a alcançar objetivos;
- Crescimento – desenvolver as próprias habilidades e pontos de vista;
- Intimidade – compreender e ser compreendido pelos outros.

Na XP, as pessoas são fundamentais no processo de desenvolvimento, por isso suas práticas são voltadas para potencializar o melhor que as pessoas têm para oferecer, bem como suprimir suas falhas [IMPROVE IT, XP].

As práticas da XP também procuram balancear as demandas dos negócios com as necessidades pessoais dos desenvolvedores. Por exemplo, limitar a carga horária dá tempo para que necessidades pessoais sejam satisfeitas – já que muitas delas, como descansar, exercitar-se e socializar-se, ocorrem fora do ambiente de trabalho – e também aumenta o rendimento e as perspectivas das pessoas durante o trabalho [BECK, 2005].

### 3.3.2 Economia

Para desenvolver software é preciso investir tempo e recursos. Este investimento é considerado satisfatório quando é compensado através do valor gerado, ou seja, quando o software satisfaz as expectativas de quem investiu nele. É um erro esquecer-se do lado econômico do desenvolvimento e preocupar-se somente com o “Sucesso Técnico” [BECK, 2005].

O cliente investe em software com a expectativa de que este gere valor comercial. A XP reconhece esta premissa e suas práticas são organizadas para antecipar receitas e adiar despesas [IMPROVE IT, XP].

### 3.3.3 Benefício Mútuo

Cada atividade deve ser benéfica para todos os envolvidos em um projeto de software. Este é o mais importante princípio da XP, e o mais difícil de cumprir.

Há sempre soluções mais fáceis em que alguns ganham e outros perdem para qualquer problema. Apesar de tentadoras, principalmente quando as pressões externas são intensas, estas soluções sempre causam mais perdas que benefícios, já que, além das perdas diretas, existe a perda causada pela deterioração das relações de trabalho.

Desenvolver software é um negócio focado nas pessoas e que depende muito das relações entre os envolvidos, por isso, deve-se adotar práticas que beneficiem ambos, criadores e clientes do software, agora e no futuro. Programação em par, por exemplo, beneficia os **programadores** de inúmeras formas. Mas, também beneficia os **clientes**, porque costuma ser raro encontrar *bugs* em funcionalidades implementadas em par, e **gerentes**, já que a disseminação do conhecimento fruto da programação em par, torna o projeto menos suscetível a perdas em função da saída definitiva ou temporária de um dos desenvolvedores [IMPROVE IT, XP; BECK, 2005].

Documentação abrangente no código é um exemplo de uma prática que viola o princípio do benefício mútuo. Esta prática costuma diminuir a velocidade de desenvolvimento consideravelmente, objetivando que no futuro seja mais fácil fazer alterações. Realmente existe um possível benefício futuro, desde que a documentação ainda esteja válida, mas nenhum benefício para o presente.

A XP recomenda para resolver o problema anterior de maneira mutuamente benéfica, agir da seguinte forma [BECK, 2005]:

- Escrever testes automatizados que ajudam a melhorar o *design* e implementação no presente. Deixar estes testes para os futuros programadores, para que estes possam ter uma documentação sincronizada com o código e um mecanismo de verificação automatizado que dá segurança para qualquer possível alteração.
- Refatorar cuidadosamente para eliminar qualquer complexidade acidental, isto gera satisfação e menos defeitos no presente, além de tornar o código mais fácil de compreender e modificar no futuro.

- Escolher nomes de um conjunto coerente e explícito de metáforas relacionadas ao domínio da aplicação. Isto aumenta a velocidade de desenvolvimento no presente e torna o código mais claro para o futuro.

### 3.3.4 Auto-semelhança

Deve-se tentar aplicar a estrutura de uma solução para outros contextos, mesmo em escalas diferentes. É algo semelhante a proposta de padrões de projetos (*Design Patterns*) descrita por Gamma [GAMMA, 1995], onde estruturas de soluções eficientes para problemas freqüentes foram catalogadas para facilitar o reuso em vários contextos.

Por exemplo, uma prática básica na XP é escrever os testes antes da implementação das funcionalidades (seção 3.5.12). Esta prática dita um ritmo de desenvolvimento que opera em diferentes escalas.

No início de cada trimestre listam-se os temas (metas) que são importantes para o período e direcionarão as histórias do trimestre. Depois disto, os testes guiam diretamente o desenvolvimento em duas escalas:

1. Em cada semana, os usuários reunidos com os desenvolvedores priorizam e selecionam, entre as histórias, algumas para implementação. Então os testes que expressam cada uma destas histórias são escritos para depois serem colocados em funcionamento. Estes são os testes de aceitação (seção 5.2.2);
2. Em poucas horas é possível fazer uma lista com os testes necessários para uma funcionalidade. Então se escreve o código do primeiro teste e o coloca em funcionamento, depois o segundo é escrito e colocado pra funcionar junto com o primeiro, depois o terceiro e assim sucessivamente até que todos os testes da lista estejam funcionando. Estes são os testes de unidade (Apêndice A - seção 5.2.1).

Auto-semelhança não é o único princípio que influencia o desenvolvimento de um software, pelo contrário, existem muitos outros. O fato de se copiar a estrutura de uma solução que funcionou bem em um ou mais contextos, não significa que ela irá funcionar sempre. De qualquer forma, é um bom lugar por onde começar. Em contrapartida, o fato de

uma solução não ser reutilizável, não significa que esta seja ruim, pois existem casos onde esta é a melhor escolha [BECK, 2005].

### 3.3.5 Melhoria

Thomas L. Friedman no livro O Mundo é Plano atribuiu a Brian Behlendorf a seguinte frase: “Software não é ouro, é alface – ele é um bem perecível”. O significado desta frase é que se não existir uma melhoria contínua, o software acaba se deteriorando. A XP adota esta premissa e dá uma atenção especial ao refinamento das atividades ao longo do tempo.

Em desenvolvimento de software não existe a solução perfeita, por este motivo a XP não se preocupa em construir softwares perfeitos. O objetivo da XP é a melhoria contínua, que se traduz na grande preocupação em aperfeiçoar cada vez mais o design, os processos, as histórias e todos os outros aspectos ligados a qualidade [IMPROVE IT, XP; BECK, 2005].

A XP busca a excelência no desenvolvimento de software através do aperfeiçoamento em ciclos. O objetivo é fazer o melhor que se pode hoje e lutar sempre para obter o conhecimento e o entendimento necessários para obter algo melhor amanhã. Isto não significa esperar por soluções perfeitas para começar. O objetivo é encontrar um ponto de partida com a melhor qualidade possível, começar, e então melhorar a partir deste ponto.

Na XP, o princípio da melhoria contínua aparece principalmente em práticas que aperfeiçoam atividades iniciadas em algum momento anterior. Os ciclos trimestrais possibilitam a melhoria do planejamento de longo prazo com base na experiência obtida em ciclos anteriores. Os ciclos semanais permitem que os resultados obtidos em semanas anteriores sejam usados para aprimorar o trabalho nas semanas seguintes.

### 3.3.6 Diversidade

Equipes de desenvolvimento de software em que todos os membros são semelhantes, apesar de mais fáceis de administrar e trabalhar, não são efetivas. As equipes precisam reunir uma variedade de habilidades, conhecimentos e personalidades, para que os

desafios sejam visualizados sobre diferentes ângulos. Desta forma, fica mais fácil enxergar problemas e armadilhas, pensar em múltiplas soluções e escolher as que melhor se encaixam num contexto [BECK, 2005].

Equipes precisam de diversidade. Se não há diversidade entre os membros de uma equipe, todos tendem a ter pontos de vista semelhantes. Isto é péssimo para o surgimento de idéias diferentes e inovadoras. Além disso, em desenvolvimento de software, frequentemente, mudar a forma como um problema é visualizado, traduz-se em grandes mudanças no tempo e custo de implementação de uma solução [IMPROVE IT, XP].

Um efeito colateral inevitável da diversidade é o aumento dos conflitos, principalmente de opiniões. Estes conflitos só são indesejáveis quando envolvem desrespeito ou prejudicam o relacionamento entre os membros da equipe, caso contrário, devém ser encarados como algo natural [BECK, 2005].

É importante saber administrar os conflitos para solucioná-los sempre de forma produtiva. Diferentes idéias em um mesmo projeto representam oportunidades, não problemas. O princípio da diversidade sugere que os desenvolvedores devem trabalhar juntos avaliando e respeitando sempre todas as opiniões [BECK, 2005].

O princípio da diversidade é adotado pela prática equipe completa, por exemplo, que aconselha a manter na equipe, além de um grupo heterogêneo de pessoas envolvidas diretamente com o desenvolvimento do sistema, o cliente e quaisquer outras pessoas que precisarem ser ouvidas durante o desenvolvimento.

### 3.3.7 Reflexão

Boas equipes não se limitam à execução do trabalho. Elas refletem sobre como estão trabalhando e porque estão fazendo de uma determinada maneira e não de outra. Elas analisam as razões que estão por trás de um sucesso ou de um fracasso, e nunca escondem erros, pelo contrário, fazem questão de expô-los, sempre visando aprimorar-se com os resultados do próprio trabalho. Não existe a preocupação de localizar e punir **quem** comete algum erro, mas existe uma grande preocupação em descobrir o **por quê** do erro, para que em função do aprendizado, este não se repita [BECK, 2005].

Práticas como o ciclo trimestral, ciclo semanal, programação em pares e integração contínua, pregam que seja reservado um tempo para reflexão entre os membros da equipe. Mas o tempo dedicado à reflexão não deve limitar-se ao estipulado formalmente nas práticas. Conversas informais com colegas de trabalho fora do expediente, ou com parentes e amigos, além de atividades recreativas em geral como férias, cinema, esporte e leituras não relacionadas a software, são ótimas para refletir sobre o modo como se está realizando o próprio trabalho e para ter novas idéias a este respeito [BECK, 2005].

Reflexão não é um exercício puramente intelectual. É possível aprender analisando e discutindo dados, mas também é importante reconhecer que é possível aprender através da intuição. Emoções “negativas” como raiva, medo ou ansiedade, por exemplo, são boas pistas de que algo de ruim pode estar para acontecer. É preciso esforço para escutar e interpretar corretamente o que as emoções sugerem sobre o próprio trabalho, mas somar este conhecimento ao adquirido pela análise intelectual dos dados é uma ótima fonte de perspicácia [BECK, 2005].

Em desenvolvimento de software, reflexão costuma ser levada longe demais, pois existe uma longa tradição de pessoas tão ocupadas pensando sobre a melhor forma de desenvolver o software que não têm tempo para realmente desenvolver o software. Na XP não existe este problema, pois a reflexão é feita em cima do *feedback* fornecido pelo que já foi feito. Ou seja, a reflexão depende essencialmente das ações e não o contrário [BECK, 2005].

### 3.3.8 Fluxo

Na XP, as atividades características do desenvolvimento são executadas **simultaneamente** e não em fases discretas como no desenvolvimento tradicional. O objetivo é manter um fluxo constante de pequenas entregas durante todo o projeto e assim gerar valor continuamente na forma de software funcional. Para isto, a XP trabalha com pequenas funcionalidades representadas sem muito detalhamento através de histórias. A cada iteração é alocado um pequeno conjunto de histórias, de forma que estas possam ser detalhadas, implementadas e colocadas à disposição do usuário rapidamente, fazendo o sistema evoluir de forma natural e segura [IMPROVE IT, XP].

O fluxo estimula a manutenção de um processo de melhoria contínua, pois permite que os desenvolvedores recebam *feedback* rápido e freqüente sobre o que estão produzindo. Além disso, permite que os clientes possam beneficiar-se rapidamente e sem grandes investimentos das vantagens do sistema, evitando o desconforto e o risco de ter que esperar muito por um possível retorno sobre um grande investimento [BECK, 2005; IMPROVE IT, XP].

Apesar das muitas vantagens, o desenvolvimento tradicional segue uma rota contrária à descrita acima. O desenvolvimento tradicional valoriza entregas de grandes porções do software de uma vez, que são precedidas por integrações feitas num grande e único passo. Um projeto é composto por uma ou mais grandes entregas divididas em fases discretas, tais como, análise, design, implementação e testes. Entregas grandes fazem o cliente esperar muito para desfrutar do valor gerado, inibem melhorias, geram maior complexidade e conseqüentemente aumentam os riscos.

É preciso ressaltar porém que manter um fluxo de desenvolvimento de software é algo complexo que só funciona bem quando auxiliado pelo bom funcionamento de algumas práticas. Por exemplo, o *buid* diário é uma prática que auxilia a manutenção do fluxo, mas compilar e integrar diariamente não é o suficiente. É preciso ter certeza que o software está funcionando corretamente todos os dias, ou melhor ainda, várias vezes durante ao dia; com a ajuda da prática TDD(Desenvolvimento Guiado por Testes), por exemplo [BECK, 2005].

### 3.3.9 Oportunidade

Aprenda a ver problemas como oportunidades de aprendizado e mudança. Isto não quer dizer que não existam problemas em desenvolvimento de software. Significa que dependendo do modo como os problemas são encarados e da reação tomada, eles podem se transformar em boas oportunidades [BECK, 2005; IMPROVE IT, XP].

### 3.3.10 Redundância

Em desenvolvimento de software é interessante que os problemas críticos e difíceis sejam resolvidos de várias maneiras diferentes. Neste caso, o custo da redundância é pago

pela maior segurança, pois mesmo que uma solução falhe totalmente, existem outras prontas que evitarão um desastre [BECK, 2005].

Apesar das redundâncias poderem significar desperdício, não é interessante removê-las quando elas servem a propósitos válidos. Por exemplo, ter uma fase de testes depois que o desenvolvimento está terminado pode ser redundante, mas só é interessante eliminá-la quando ela provar-se redundante na prática, não achando nenhum defeito em muitas implantações e durante várias vezes seguidas [BECK, 2005].

### 3.3.11 Falha

Se estiver com dificuldade em obter sucesso, então falhe. Se a falha gerar conhecimento, ela não pode ser considerada um desperdício, pois conhecimento é algo muito valioso e algumas vezes difícil de conseguir.

No caso de existirem dúvidas sobre qual é a mais indicada, entre algumas formas de implementar uma história, é melhor tentar todas e descobrir na prática, do que discutir longamente os detalhes para selecionar uma das opções. Até mesmo se todas falharem, o conhecimento adquirido será certamente valioso. Se a maneira mais indicada fosse conhecida previamente, ela seria a única a ser implementada. Porém, quando existem dúvidas, implementar de várias formas pode ser mais fácil e barato do que longas discussões [BECK, 2005; IMPROVE IT, XP].

### 3.3.12 Qualidade

Maior qualidade significa menos defeitos e retrabalho, menos aborrecimentos e maior segurança para clientes e desenvolvedores, maior confiança, menos ansiedade, maior motivação, maior efetividade, maior produtividade e maiores lucros, entre outras coisas. Ou seja, maior qualidade significa gerar maior valor de forma mais simples e eficientes, e com menos custos [IMPRVE IT, XP].

Qualidade motiva a equipe, entre outras coisas, porque as pessoas geralmente não gostam de fazer trabalhos medíocres, pelo contrário, elas sentem-se muito mais motivadas quando têm a oportunidade de trabalhar com qualidade e se orgulham disto [BECK, 2005].

Existe uma crença de que alta qualidade signifique gastos mais elevados. Não há dúvidas de que a qualidade tem um preço, mas a falta dela tem um preço ainda maior [IMPROVE IT, XP]. Sacrificar a qualidade nunca é uma boa forma de controlar o tempo ou os custos. Projetos não se tornam mais rápidos ou mais baratos, de acordo com a diminuição da qualidade, assim como não se tornam mais lentos ou mais caros de acordo com o aumento. Geralmente ocorre o contrário, ou seja, o aumento da qualidade gera entregas mais rápidas e com menores custos, enquanto que a diminuição resulta freqüentemente em atrasos e maiores custos [BECK, 2005].

A XP acredita que qualidade não é uma boa variável de controle para projetos. Para a XP, uma forma eficiente de gerenciar o andamento de um projeto com alguma flexibilidade é através do controle do escopo, já que este nunca é conhecido precisamente e em detalhes com uma boa antecedência. Além disso, o tempo e o custo são comumente fixados logo no início do projeto. Os ciclos semanais e trimestrais provêem pontos explícitos para acompanhamento e escolha do escopo [BECK, 2005].

Qualidade não é um argumento para falta de ação. Caso não se conheça uma solução clara para uma tarefa urgente, o indicado é resolver da melhor maneira possível. Se existe uma solução clara, mas que vai levar muito tempo, deve-se fazer com a melhor qualidade possível de acordo com o tempo disponível [BECK, 2005]. O mais importante é sempre ter em mente o significado e a importância de princípios como qualidade e melhoria, e assim agir da forma mais efetiva de acordo com cada contexto.

Talvez a única característica que realmente justifica o nome programação extrema para a metodologia, seja a **extrema** importância dada à geração do máximo de valor da forma mais simples e eficiente possível, e é por este motivo que a qualidade é tão importante para a XP.

### 3.3.13 Passos Pequenos (Passos de Bebê)

É sempre tentador fazer grandes mudanças em passos grandes. Afinal de contas, o comum é que exista um longo caminho a seguir e pouco tempo para chegar lá. Mudanças importantes feitas em um único passo são perigosas. A execução do trabalho em passos pequenos diminuí muito este rico [BECK, 2005].

Passos pequenos são muito comuns na XP. Eles são básicos em práticas como: TDD, que constrói e faz passar sempre um teste por vez; integração contínua, que integra e testa a cada vez, sempre o que foi gerado por poucas horas de desenvolvimento.

### 3.3.14 Responsabilidade Aceita

Responsabilidade deve ser aceita e não atribuída. Só a pessoa que recebe a responsabilidade pode decidir se a aceita ou não.

As práticas refletem responsabilidade aceita, sugerindo que quem quer que aceite fazer um trabalho também o estime. Da mesma forma, a pessoa responsável por implementar uma história também é responsável pelo design, implementação e teste da mesma [BECK, 2005].

## 3.4. Práticas da XP

A XP adota um conjunto de práticas que representam a parte mais visível, principalmente aos olhos externos, do trabalho da equipe de desenvolvimento de software.

As práticas, por si só, não fazem muito sentido, pois não indicam muito claramente o porque de sua importância. O Desenvolvimento Dirigido por Testes (TDD), por exemplo, é uma prática poderosa, com um conjunto bem definido de regras, mas que apesar disso, pode gerar bastante desconforto e resistência numa equipe que não conhece os valores que norteiam sua aplicação. Seguir as regras da TDD simplesmente porque alguém hierarquicamente superior mandou, não é algo muito estimulante para a equipe. Seguir estas regras para obter *feedback* mais rápido e eficiente, estimular a comunicação no próprio código, simplificar o sistema, aumentar a confiança e conseqüentemente a coragem, é dar um sentido ao que se está fazendo.

As práticas definem regras claras e concretas, e se comprometem em satisfazer alguns valores e princípios básicos. Também indicam ações que normalmente são específicas para certos contextos. Mudar o contexto normalmente significa mudar de alguma forma as práticas adotadas, com a restrição de que as mudanças não podem alterar o compromisso de

sempre estar de acordo com os valores e princípios, pelo contrário, as mudanças devem visar a manutenção deste compromisso.

A práticas da XP são úteis tanto isoladamente quanto em conjunto, mas a forma ideal de aplicá-las é em conjunto, pois são bastante inter-relacionadas e costumam ter seus resultados multiplicados quando aplicadas desta forma. Ou seja, a XP não impõe o uso de um determinado conjunto de práticas, mas defende que é o uso em conjunto que trás sempre os melhores resultados.

Não existe uma ordem exata para aplicação das práticas num projeto, mas existe uma dependência das práticas corolário em relação às primárias, como é explicado detalhadamente na seção 3.6. A escolha de quais práticas utilizar depende do contexto do projeto e das características da equipe.

O ritmo de adoção das práticas por uma equipe, deve ser o mais natural possível. Isto geralmente se traduz numa adoção incremental, que costuma ser a mais fácil e estimulante, já que permite de forma clara, visualizar os efeitos positivos de cada nova prática adotada e a influência positiva desta sobre as práticas anteriores [CARDIM, 2006].

Devido a liberdade evolutiva da XP e de suas práticas, não é correto fixar algumas práticas como sendo as “oficiais” da metodologia. As práticas nascem, evoluem e morrem pragmaticamente de acordo com o *feedbeck* obtido por aqueles que as adotam. Já os princípios e principalmente os valores, apesar de também estarem em evolução contínua, tendem a mudar com uma freqüência menor [TELES, XPERS].

Nesta seção, são abordados os fundamentos das práticas mais populares da XP. A definição da listagem de práticas e da forma de classificação destas, tem como base o que está definido em [BECK, 2005] e em [IMPROVE IT, XP].

## 3.5. Práticas Primárias

Pode-se começar com qualquer uma delas ou aplicá-las em qualquer momento no decorrer do projeto, e obter benefícios imediatos de forma segura. A ordem em que elas devem ser adotadas não é determinante para a melhoria da qualidade. A decisão de qual e quando aplicar, depende unicamente do que a equipe identificar como a melhor oportunidade de melhoria para o projeto atual [IMPROVE IT, XP].

### 3.5.1 Sentar-se Junto

A XP é um processo de desenvolvimento totalmente focado no trabalho em equipe. Por isso, com o objetivo de estimular ao máximo a troca de informações e idéias, os membros de uma equipe XP devem ficar todos juntos num ambiente grande e confortável, para que todos possam trabalhar em conjunto de forma rápida e fácil.

A prática indica que é muito bom manter o contato presencial da equipe, mas não propõe nenhum empecilho ao trabalho distribuído. Quando o contato presencial não é possível, existem outras práticas recomendadas pela XP, que também estimulam o trabalho em equipe e podem ser casadas com desenvolvimento distribuído, sem desobediência aos valores da XP [IMPROVE IT, XP].

### 3.5.2 Equipe Completa

Equipe completa é mais uma das práticas primárias propostas por [BECK, 2005] e, direta ou indiretamente, comentadas por [TELES, 2004; IMPROVE IT, XP]. É de acordo com estas referências que a prática será explicada.

É aconselhável que os membros de uma equipe tenham habilidades e perspectivas diferentes, que em conjunto representam o necessário para o sucesso do projeto. A equipe deve ser dinâmica, ou seja, se alguma habilidade específica se faz necessária, sempre que possível, deve-se trazer alguém com esta habilidade. Seguindo este raciocínio, se alguma habilidade especifica não se faz mais necessária, a pessoa que a exerce deve ir para outra equipe, ou então assumir novas responsabilidades dentro da equipe.

Dentro da XP, uma equipe não é formada somente pelo grupo de pessoas que desenvolve o sistema. Além destes, o cliente e quaisquer outras pessoas que precisarem ser ouvidas durante o desenvolvimento, fazem parte da equipe e são tão importantes quanto os desenvolvedores.

### 3.5.3 Local de Trabalho Informativo

Um local de trabalho adequado estimula a adoção das outras práticas e a obediência aos valores da XP, por isso tem importância fundamental. Dentro deste contexto, manter o local de trabalho com informações acessíveis a todos e atualizadas tem grande peso. Segundo Beck [BECK, 2000], sem um local de trabalho adequado, dificilmente um projeto terá êxito.

Um local de trabalho informativo é aquele que transmite informações atualizadas sobre os projetos de forma simples e direta. Isto significa que um observador interessado pode andar no ambiente de trabalho de uma equipe e obter, em poucos segundos, informações gerais sobre como um projeto está progredindo. Além disso, observando-se com um pouco mais de atenção, deve-se ser possível obter informações sobre problemas reais ou potenciais [TELES, 2004; IMPROVE IT, XP].

Para atingir tais objetivos, equipes XP utilizam diversos instrumentos, tais como: cartões com histórias, colocados em um mural na parede; quadro branco, que além de informativo é uma ótima ferramenta de apoio a discussões e debates; lembretes em papel colados nas paredes (“Post It”); bloco de folhas de papel grande (“Flip chart”); qualquer ferramenta ágil, simples e acessível a todos da equipe, para troca de informações no local destinado ao trabalho [TELES, 2004; IMPROVE IT, XP].

### 3.5.4 Trabalho Energizado

Deve-se trabalhar somente a quantidade de horas que se pode agüentar, sem declínio da produtividade pessoal ou riscos para a saúde. Ou seja, é preciso manter um ritmo sustentável de trabalho, respeitando os próprios limites, como ser humano.

Dentre os problemas relacionados ao desenvolvimento de software, um dos mais comuns é não conseguir cumprir os prazos, como mostrado nos estudos feitos por *Standish*

*Group International* [STANDISH, 1994]. Este é um dos principais motivos que leva um gerente de projeto de software a pressionar sua equipe em direção ao excesso de trabalho.

Os gerentes costumam esquecer que os desenvolvedores não são máquinas e sim seres humanos, e como tais, precisam descansar, comer e dormir, entre outras necessidades básicas, para manterem-se “Energizados” e produtivos. Trabalhar além do normal pode gerar um ganho de produtividade inicial, mas em pouco tempo o desenvolvedor começará a apresentar dificuldades de concentração. Isto se reflete negativamente na velocidade de desenvolvimento e na qualidade do sistema, já que estas são diretamente influenciadas pela capacidade do desenvolvedor manter-se atento, criativo e disposto a solucionar problemas.

Todo o trabalho de desenvolvimento de um software ocorre na mente do desenvolvedor, no máximo existem ferramentas que o ajudam a materializar o que foi pensado. Um desenvolvedor mentalmente esgotado tende a produzir valor muito lentamente e com muitos erros.

É muito fácil remover valor de um software, mas quando se está cansado é muito difícil perceber que se está removendo valor [BECK, 2005]. Assim, é comum o tempo extra ser gasto em vão, pois o fato de trabalhar mais não significa que o projeto avançará [IMPROVE IT, XP]. Pelo contrário, existe um grande risco que, mesmo com todo o esforço extra, o projeto regrida em função do aumento na quantidade de bugs.

Nos projetos da XP, as pressões de tempo são tratadas através do processo de priorização e ajuste do escopo de cada iteração para a real capacidade de entrega da equipe [TELES, 2005]. O mais importante não é trabalhar mais e sim trabalhar da forma mais inteligente, em um período de tempo semanal que as pessoas sejam capazes de sustentar sem ficarem esgotadas, e sem prejudicarem o trabalho com o déficit de atenção decorrente da fadiga [IMPROVE IT, XP].

### 3.5.5 Programação em Par

Programação em par é uma das práticas mais conhecidas e mais polêmicas utilizadas em projetos XP. Quando é adotada, os desenvolvedores implementam as funcionalidades em pares, ou seja, diante de cada computador, existem sempre dois desenvolvedores que trabalham juntos para produzir o mesmo código. É uma prática difícil

de aceitar, seja pela falta de hábito, ou pela falsa impressão de que se está desperdiçando recursos ou atrasando o desenvolvimento do sistema [TELES, 2004; IMPROVE IT, XP; BECK, 2005].

A verdade é que esta prática permite que o código seja revisado permanentemente enquanto é construído, o que diminui consideravelmente a incidência de erros. Além disso, contribui para que a implementação seja mais simples e eficaz, já que os desenvolvedores se complementam e têm mais oportunidades de gerar soluções inovadoras.

Outra característica muito importante da programação em pares é a disseminação do conhecimento. Com o tempo, isto ajuda a aumentar o nível técnico da equipe como um todo e, conseqüentemente, a melhorar tanto a agilidade no desenvolvimento quanto a qualidade dos softwares desenvolvidos. Além disso, a disseminação do conhecimento, por evitar que um único desenvolvedor se especialize em determinadas partes do sistema, é uma ótima proteção contra os eventuais prejuízos causados pela saída de algum membro de uma equipe.

O conjunto de características apresentadas acima decorre, entre outras coisas, da maior oportunidade que os desenvolvedores têm de dialogar, trocar idéias e explorar mais alternativas, do que teriam se estivessem programando sozinhos. Essas características fazem com que a programação em par acelere o desenvolvimento e o torne mais econômico, embora à primeira vista pareça o contrário [IMPROVE IT, XP].

### 3.5.6 Histórias

Equipes XP planejam utilizando unidades funcionais visíveis ao cliente, escritas em pequenos cartões e chamadas de histórias. As histórias expressam as funcionalidades de forma resumida e preferencialmente devem ser escritas pelo próprio cliente. A regra é escrever algo simples e resumido que respeite o espaço do cartão.

O registro da história, feito no cartão, serve basicamente como um convite ao diálogo, já que os detalhes de cada história devem ser obtidos preferencialmente através do diálogo presencial. Exemplos de histórias: “Apresente ao cliente as dez tarifas mais baratas para uma determinada rota” [BECK, FOWLER, 2001]; “Forneça uma alternativa com dois cliques para usuários discarem números freqüentemente usados” [BECK, 2005].

Para que o cliente possa priorizar as histórias da melhor forma é preciso que, tão logo ela seja escrita, o esforço para implementação seja estimado pelo desenvolvedor. Desta forma, o cliente pode visualizar o possível custo de cada história e não somente os benefícios. Em conseqüência disto, fica mais fácil para o cliente tanto priorizar as histórias mais importantes, quanto negociar ajustes ou alternativas.

A XP recomenda que as estimativas sejam sempre feitas em equipe, de modo que todos possam avaliar e levantar aspectos importantes. Estimativas geradas em conjunto tendem a ser mais precisas. Também é muito importante que o cliente esteja presente durante a definição das estimativas, para tirar eventuais dúvidas e evitar interpretações erradas [TELES, 2004].

Os cartões são colocados na parede, geralmente em um mural, de modo que possam ser acompanhados facilmente por todos os envolvidos com o projeto. Devido à eficácia superior comprovada dos cartões tradicionais, o ideal é evitar substituí-los por softwares, a não ser que as circunstâncias imponham esta substituição, como pode acontecer por exemplo, em alguns projetos distribuídos [IMPROVE IT, XP].

### 3.5.7 Ciclo Semanal

Em projetos XP, o software é desenvolvido de modo iterativo e incremental. Numa iteração, representada por um ciclo semanal, algumas histórias escolhidas pelo cliente são implementadas e testadas completamente. Terminado o ciclo, o cliente tem a oportunidade de utilizar e avaliar o que foi produzido [IMPROVE IT, XP].

Existe uma medida que serve basicamente para avaliar a quantidade de valor que uma equipe é capaz de produzir por iteração. Ao final de cada iteração a velocidade real da equipe é comparada com a estimada no início desta, e em função da diferença entre o valor real e o esperado, a velocidade é ajustada para ser utilizada como valor inicial da próxima iteração. Esta estimativa pode ser feita em horas, pontos, dias, ou alguma outra medida com que a equipe esteja habituada, e devido ao refinamento contínuo feito a cada iteração, tende a tornar-se cada vez mais confiável [BECK, FOWLER, 2001].

Cada ciclo semanal se inicia com uma reunião chamada de jogo do planejamento. Nessa reunião o cliente escreve e prioriza histórias de acordo com o aprendizado adquirido

através do contato com as funcionalidades implementadas na iteração anterior, ou com as necessidades imaginadas normalmente ao longo do projeto.

É também no jogo do planejamento que, com base na velocidade da equipe da semana anterior, além das considerações técnicas de cada história, os desenvolvedores fornecem uma estimativa para o cliente indicando a quantidade de histórias que poderão ser implementadas. De posse dessas informações, o cliente prioriza as histórias que, em conjunto, mais possam gerar valor para ele quando estiverem implementadas [IMPROVE IT, XP]. As histórias que não forem escolhidas ficam para serem reavaliadas nos próximos ciclos.

Qualquer história do projeto pode ser modificada pelos usuários, com exceção daquelas que foram escolhidas para a iteração (ciclo semanal) atual. Esta regra é necessária para que o fluxo de desenvolvimento se mantenha contínuo, as estimativas dos ciclos semanais não se tornem inviáveis, e os clientes sejam estimulados a tomar decisões com base no *feedback* do software. Apesar de parecer rígida, esta regra é fácil de ser seguida, já que em muito pouco tempo o cliente tem as histórias da iteração atual implementadas. Assim, no próximo jogo do planejamento, com base também no *feedback* da funcionalidade na forma de software, o cliente pode avaliar melhor a modificação desejada e pedi-la ou não.

Muitos projetos dedicam esforços significativos à implementação de funcionalidades que não são utilizadas, gerando grandes desperdícios (figura 3.2). Decidir bem o que implementar é uma atividade que deve ser conduzida com o maior cuidado e durante todo o desenvolvimento. Com o ciclo semanal, o risco de implementar uma funcionalidade que não interessa ao cliente reduz-se muito, pois o cliente interage o tempo todo com as funcionalidades, à medida que estas vão sendo implementadas. Ou seja, o cliente além de receber valor muito mais rapidamente, tem a oportunidade de, ao interagir com o valor gerado, verificar continuamente se o caminho seguido é realmente o desejado [BECK, 2005; TELES, 2004; IMPROVE IT, XP].

### 3.5.8 Ciclo Trimestral

O planejamento de um projeto XP é dividido em trimestres. No inicio de cada trimestre reúne-se toda a equipe, incluindo clientes e demais envolvidos, para avaliar o projeto, seu progresso e seu alinhamento com as metas principais.

Nesta reunião deve-se: identificar gargalos; iniciar reparos; definir o tema ou os temas para o trimestre; estimar quais histórias, de uma forma geral, gerarão o valor necessário aos temas escolhidos para o trimestre; focar no todo, ou seja, onde o projeto se encaixa na organização.

Temas são descrições curtas para conjuntos de funcionalidades que solucionam as necessidades de um ou mais processos de negócios da organização [IMPROVE IT, XP]. A intenção é diminuir a tendência dos desenvolvedores de focarem muito nos detalhes específicos de cada funcionalidade ou história, e esquecerem do todo. Os temas ajudam os desenvolvedores a lembrarem de refletir durante a implementação, sobre como as histórias da semana se encaixam melhor no todo. Também servem para planejamento em maior escala [BECK, 2005].

No planejamento do ciclo trimestral a equipe também deve refletir sobre o andamento do projeto no trimestre anterior, identificar problemas e propor ações para solucioná-los no trimestre que se inicia [IMPROVE IT, XP].

### 3.5.9 Folga

É interessante incluir no cronograma algumas tarefas menores que podem ser removidas se ocorrer algum atraso, já que equipes de desenvolvimento de software tendem a se comprometer a entregar mais funcionalidades do que podem. Quando a equipe não honra seus compromissos, os clientes perdem a confiança. Desconfiança gera inúmeros desperdícios, tais como: pressões por prazos muito curtos que aumentam os riscos de erros; baixa motivação que diminui a produtividade; conflitos de relacionamento [IMPROVE IT, XP].

Equipes XP devem saber estimar bem o que são capazes de fazer, para estabelecer uma folga na hora de comprometer-se com o cliente. Isto pode ser feito através da inclusão

de atividades secundárias no ciclo semanal que, caso seja necessário, podem ser ignoradas para não atrasar a implementação do que foi acordado com o cliente.

É possível incorporar folgas ao projeto de muitas formas, tais como: de cada oito semanas, separar uma para tarefas técnicas; reservar vinte por cento do comprometimento semanal para tarefas técnicas escolhidas pelo programador; considerar apenas 80% da equipe quando estiver selecionando histórias para uma iteração [BECK, 2005].

Estimar bem uma história é pensar na quantidade real de tempo necessário para implementá-la. Porém, nem sempre o cliente está preparado para comunicação clara e honesta. Neste caso, é prudente inserir uma folga que justifique para o cliente o tempo estimado [BECK, 2005].

### 3.5.10 *Build* de dez minutos

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

Algumas tarefas comuns durante o *build* são:

- Compilação do código do sistema;
- Compilação e execução dos testes unitários e de aceitação;
- Geração de relatórios dos resultados dos testes, de cobertura de código por testes e de análise estática do código, entre outros;

Deve ser possível compilar automaticamente todo o sistema e rodar todos os testes em no máximo dez minutos. Quanto maior for o tempo necessário para realizar um *build* e rodar todos os testes, menos vezes isto será feito no total, conseqüentemente menos *feedback* será gerado pelo sistema [BECK, 2005].

Se o *build* começar a ficar lento, a equipe precisa descobrir rapidamente uma forma de mantê-lo rápido. Isto deve ser encarado como uma prioridade durante o desenvolvimento do sistema [TELES, 2004]. A automatização é indispensável, porque dá condições para que os *builds* e testes sejam executados sem erros e num tempo satisfatório.

### 3.5.11 Integração Contínua

Consiste em integrar o trabalho diversas vezes ao dia, assegurando através dos testes principalmente, que a base de código permaneça consistente ao final de cada integração [IMPROVE IT, XP].

Integrar significa armazenar na base unificada de código alguma modificação. Para diminuir a ocorrência de conflitos de versões, o código deve ser integrado de forma serial e logo após qualquer implementação. Recomenda-se manter um intervalo de no máximo duas horas entre a finalização da alteração e a integração desta ao repositório de código.

Um modo simples de garantir o processo serial de integração é ter uma máquina dedicada somente para isto. Quando a máquina estiver livre, um par de desenvolvedores com código a integrar, a ocupa e carrega a versão corrente do sistema, depois carrega as alterações que fizeram, resolve possíveis colisões e roda os testes até que todos eles passem. Assim todos sempre sabem quando podem ou não integrar [TELES, 2004].

O processo serial em conjunto com o intervalo pequeno entre implementação e integração, torna mínimo o risco de dois pares editarem um mesmo arquivo ao mesmo tempo e gerarem versões conflituosas. Além disso, mesmo quando ocorrem conflitos, é fácil percebê-los com a ajuda de um software de controle de versão, e resolvê-los, seja automaticamente, através do próprio software de controle de versão, seja manualmente, através do diálogo [TELES, 2004].

É muito importante que as alterações só possam ser integradas se não gerarem nenhum erro e que o processo de integração seja muito dinâmico. Por isso, é fundamental que todo o código do sistema esteja coberto por testes automatizados, e que se use ferramentas que simplificam e automatizam o processo de integração continua, como as mostradas a seguir [TELES, 2004; BECK, 2005].

**Ferramentas de *Build***

Tornam mais rápido o *build* de um sistema, através da automatização das tarefas envolvidas neste processo, descritas na seção 3.5.10.

Duas ferramentas de *build* muito utilizadas são o Ant para Java e o NAnt para .Net.

**Ferramentas de controle de versão**

Servem para armazenar e manter versões, além de gerenciar o acesso e modificação de forma distribuída, de todo o código fonte do sistema. Também conhecidas como repositórios, automatizam tarefas como:

- Identificação de mudanças locais;
- Exibição das diferenças entre o código de um arquivo local e uma de suas versões armazenadas no repositório;
- Identificação de quem realizou uma determinada alteração;
- incorporação de uma mudança em um arquivo local a uma versão do mesmo arquivo armazenada no repositório;
- Sincronização do código local de um ou mais arquivos, com uma das versões armazenadas no repositório.

Alguns exemplos de ferramentas de controle de versão muito utilizadas são: Subversion, conhecido também por SVN, e CVS (*Concurrent Version System*), mais antigo e menos eficiente que o SVN.

### 3.5.12 Desenvolvimento Guiado por Testes (TDD)

Na XP existe uma grande preocupação em criar softwares que sejam saudáveis, ou seja, que além de funcionarem corretamente, se mantenham assim ao longo do tempo, à medida que recebam novas funcionalidades. Para isso é essencial investir em testes automatizados, implementados antes mesmo de escrever o código das funcionalidades.

Para lidar com esse desafio, a XP utiliza, entre outras coisas, o Desenvolvimento Guiado por Testes. Trata-se de uma técnica preventiva utilizada durante todo o projeto e a manutenção, que é conhecida por defender que se deve escrever um mecanismo de teste automatizado antes de codificar os elementos de cada história do sistema. Como explicado detalhadamente no apêndice A deste trabalho, TDD é uma técnica rica, muito bem embasada conceitualmente e que traz muitas vantagens práticas, entre elas: melhor *design*, melhor documentação, menos erros, menores custos, melhor aprendizado e maior produtividade.

Em termos gerais, equipes XP acreditam que “a única forma de se mover rapidamente e com confiança é tendo uma rede de testes, tanto de unidade, quanto de aceitação” [JEFFRIES, ANDERSON, 2001, p.33; TELES, 2005, p.109].

Os testes de aceitação documentam e monitoram automaticamente, num nível mais alto, as funcionalidades (histórias) que os clientes desejam para o sistema. Podem ser utilizados inclusive como uma lista de objetivos que se autoverificam [HUNT, 2004; BECK, 2002].

Os testes de unidade documentam e monitoram automaticamente os blocos de código (métodos de classes, por exemplo) necessários para se implementar as funcionalidades. Entre outras coisas, eles mostram como está o acoplamento entre as classes, e agilizam a correção de erros pois mostram rapidamente onde e porque eles ocorrem [HUNT, 2004; BECK, 2002].

No desenvolvimento tradicional, escrever os testes antes das funcionalidades é algo que não faz muito sentido. É inegável que não é intuitivo começar escrevendo os testes de algo que ainda não existe, e é justamente por isto que o desenvolvimento guiado por testes é fruto de uma evolução gradativa e baseada fundamentalmente em resultados práticos, ou seja, ele não surgiu de um estudo teórico prévio. Foram profissionais experientes e muito qualificados que descobriram com base em muitos sucessos e fracassos, que é mais vantajoso escrever os testes primeiro. Isto é comum na filosofia da própria XP, cujas práticas mudam e evoluem continuamente com base no *feedback* de quem as utiliza no dia a dia.

Informações mais detalhadas sobre a estrutura e funcionamento das classes de teste estão disponíveis no Apêndice A.

### 3.5.13 *Design* Incremental

Deve-se investir no *design* do sistema todos os dias, para mantê-lo o mais eficiente possível para as necessidades atuais [BECK, 2005]. O *design* em XP evolui de forma iterativa e incremental junto com o resto do projeto. A evolução é contínua e proporcional às necessidades identificadas para a iteração atual do sistema.

A XP, ao contrário das metodologias tradicionais, não faz longas definições do *design* do sistema em documentos com diversas páginas e diagramas variados, que servirão de base para que outras pessoas realizem a implementação. Porém é errado pensar que a XP defende a idéia de minimizar os investimentos em *design*, pelo contrário, o objetivo é até aumentar o investimento total em *design*, desde que o projeto realmente precise disto.

Quaisquer características que possam ser implementadas para dar apoio às futuras funcionalidades, só serão codificadas de fato, se e quando tais funcionalidades forem priorizadas para uma iteração. Assim, busca-se concentrar os esforços da equipe naquilo que se tem certeza absoluta de que será necessário, de acordo com as prioridades definidas pelos clientes, para a iteração corrente. Aquilo que poderia ser útil no futuro é deixado para resolver no futuro, quando houver certeza da necessidade [IMPROVE IT, XP].

Isso pode levantar a preocupação de que não seja possível solucionar os problemas futuros com rapidez. Entretanto, as demais práticas da XP ajudam a equipe a ter agilidade e segurança, de modo que ela possa esperar o futuro chegar sem se comprometer com ele cedo demais. Além disso, a prática regular de tentar antecipar-se ao futuro nunca foi uma boa alternativa, já que de acordo com os padrões da indústria de software, 64% das funcionalidades (figura 3.2) de um sistema comercial, praticamente não são utilizadas [IMPROVE IT, XP].

## 3.6. Práticas Corolário

Cada prática corolário, para funcionar corretamente, depende da implantação efetiva de uma ou mais práticas primárias. Por este motivo, as práticas corolário tendem a ser difíceis ou perigosas de aplicar, sem antes ter domínio sobre as práticas primárias. “Se você começar a implantar o software diariamente, por exemplo, sem baixar a taxa de defeitos para algo muito próximo de zero (com programação em par, integração contínua e desenvolvimento guiado por testes), você terá um desastre nas mãos.” [IMPROVE IT, XP].

As práticas primárias e as corolário não são tudo que é necessário para desenvolver software com sucesso. Porém, elas elevam enormemente as chances de se obter sucesso no desenvolvimento de software. O ideal é que, caso seja encontrado um problema não coberto

por nenhuma das práticas, a equipe examine bem o problema encontrado e, sem desrespeitar os valores e princípios da XP, crie uma solução apropriada para o problema.

### 3.6.1 Envolvimento Real do Cliente

Faça com que as pessoas cuja vida e negócios serão afetados pelo sistema participem da equipe. Clientes com visão de longo prazo devem participar do planejamento dos ciclos semanais e trimestrais. Eles podem ter um orçamento, ou seja, um percentual da capacidade de desenvolvimento da equipe, para pedirem o que quiserem. O propósito do envolvimento dos clientes é reduzir o esforço em vão, colocando quem tem as necessidades em contato direto com quem pode satisfazê-las [BECK, 2005; IMPROVE IT, XP].

Um mesmo projeto terá resultados diferentes se for feito sem e com o envolvimento real dos clientes. Não ter nenhum cliente envolvido, ou ter um representante do cliente, conduz ao desperdício representado pela: implementação de funcionalidades que não serão utilizadas; especificação de testes que não correspondem aos critérios reais de aceitação; perda da oportunidade de construir relacionamentos reais e saudáveis entre pessoas com as mais diversas visões do projeto [BECK, 2005; IMPROVE IT, XP]. De forma geral, quanto mais próximo estão as necessidades dos clientes, das funcionalidades desenvolvidas, mais valioso o desenvolvimento se torna.

Uma objeção ao envolvimento do cliente é que a pessoa envolvida vai obter exatamente o sistema que ela quer, mas não necessariamente o sistema que a maioria, ou todos, os futuros usuários querem. Este argumento desconsidera que pode haver mais de um cliente envolvido, ou seja, um pequeno grupo heterogêneo de clientes pode representar as principais necessidades de todos, cuja vida e negócios serão afetados pelo sistema. Ainda assim, é possível que algumas necessidades não sejam incluídas, mas neste caso, vale a máxima de que é bem mais fácil generalizar um pouco mais um sistema bem sucedido, do que especializar um sistema que não resolve a grande maioria dos problemas dos usuários [BECK, 2005; IMPROVE IT, XP].

Não existe a pretensão de ter o cliente disponível 100% do tempo, mas é muito importante tê-lo participando de forma ativa e intensa, junto com todas as outras pessoas importantes para o projeto, o máximo de tempo possível, pois isto permite que o projeto

seja guiado por uma série de pequenos ajustes e não por mudanças bruscas, reduzindo-se muito os riscos.

A participação intensa do cliente é muito importante, pois:

- Evita o trabalho especulativo, que ocorre quando a equipe não consegue ter alguma dúvida respondida, assume premissas e implementa algo por conta própria, achando que é do interesse do cliente.
- Torna mais ágil o processo de desenvolvimento, já que a comunicação entre desenvolvedores e clientes será sempre a mais rápida e fácil possível.
- Faz o cliente aprender sobre os desafios e limitações encontradas pelos desenvolvedores e valorizar mais facilmente o serviço prestado. Isto contribui para melhorar a confiança e o respeito de ambas as partes.
- Evita situações incorretas, já que o cliente tem maior visibilidade sobre o que está sendo feito e a equipe pode apontar facilmente questões ignoradas pelo cliente.

Em suma, a incorporação das pessoas que tem algo importante a dizer a equipe, com destaque para o cliente, estimula o *feedback*, comunicação, simplicidade, respeito e coragem dentro do processo de desenvolvimento, ou seja, promove os valores básicos da XP.

### 3.6.2 Implantação Incremental

Quando estiver substituindo um sistema legado, implante gradualmente e desde o início do projeto as partes do novo sistema que forem ficando prontas, substituindo as partes equivalentes no sistema legado. De vez em quando grandes implantações funcionam, mas são muito arriscadas e têm custos humanos e econômicos muito elevados.

A alternativa às grandes implantações é encontrar algumas funcionalidades ou um conjunto limitado de dados que possam ser controlados em separado e imediatamente, e então implantá-los. Enquanto a implantação não for concluída, terá que ser encontrada uma forma de rodar os dois sistemas em paralelo, dividindo ou fundindo arquivos ou treinando alguns usuários para usar ambos os sistemas por um tempo. Esta necessidade, técnica ou social, é o preço que se paga pela segurança [BECK, 2005; IMPROVE IT, XP].

### 3.6.3 Continuidade da Equipe

Mantenha equipes eficientes juntas. Há uma tendência em grandes organizações de abstrair pessoas para coisas, como se pessoas fossem unidades de programação *plug-and-play*. Valor em software é criado não apenas pelo o que as pessoas conhecem e fazem, mas também por seus relacionamentos e pelo que elas realizam juntas [IMPROVE IT, XP].

Organizações pequenas não têm esse problema. Há somente uma equipe e, uma vez que seus membros estejam integrados e confiando uns nos outros, somente uma verdadeira calamidade pode separá-los. Grandes organizações freqüentemente ignoram o valor das equipes, adotando ao invés disso, uma metáfora em que os desenvolvedores são meros "recursos de programação". Uma vez que o projeto está pronto, os desenvolvedores voltam para "para a fila" e aguardam até que sejam alocados para uma nova tarefa. O objetivo desse tipo de gerenciamento de "recursos" é fazer com que todos os desenvolvedores sejam integralmente utilizados. Essa estratégia maximiza micro-eficiências, mas mina a eficiência da organização como um todo. O principal erro é buscar uma eficiência ilusória em que indivíduos ficam alocados no trabalho o maior tempo possível, e em contrapartida, minimizar ou ignorar o valor do trabalho em equipe, principalmente quando os membros se conhecem bem e confiam uns nos outros [IMPROVE IT, XP].

Estimular que equipes bem integradas permaneçam juntas não significa que as equipes tenham que ficar inteiramente estáticas. É notável como novos membros começam a contribuir rápido em equipes XP já estabelecidas. Eles insistem em implementar tarefas de desenvolvimento já na primeira semana, e já estão contribuindo de forma independente após um mês. Preservando equipes e ainda encorajando uma quantidade razoável de rotação espontânea entre os membros, a organização ganha os benefícios tanto das equipes estáveis, quanto da dispersão consistente de conhecimento e da experiência [BECK, 2005; IMPROVE IT, XP].

### 3.6.4 Equipes que Encolhem

À medida que uma equipe cresce em capacidade, é importante manter a carga de trabalho constante, mas gradualmente reduzir o tamanho da equipe. Isso libera pessoas para formar mais equipes. Então, quando uma equipe ficar com muito poucas pessoas, junta-se

esta com outra equipe que também esteja muito pequena. Essa é uma prática usada pelo Sistema de Produção da Toyota, por exemplo. Outra estratégia muito usada para tentar aumentar a escala de produção é criar equipes cada vez maiores, mas funciona tão mal que alternativas devem ser consideradas [BECK, 2005; IMPROVE IT, XP].

No mundo do desenvolvimento de software ágil, uma forma de reduzir o tamanho das equipes é tentar descobrir quantas histórias o cliente realmente precisa a cada semana. E então tentar melhorar a capacidade de desenvolvimento até que algum dos membros fique ocioso, depois disto, a equipe estará pronta para diminuir de tamanho e continuar produzindo a mesma quantidade de histórias por semana.

### 3.6.5 Código Coletivo

Em um projeto XP, qualquer membro da equipe tem permissão para modificar qualquer parte do projeto, a qualquer momento. Ou seja, a propriedade do código é coletiva e todos são igualmente responsáveis por todas as partes. Com isso, os desenvolvedores ganham tempo, pois não precisam esperar a autorização de um colega para editar uma área do código e há maior disseminação de conhecimento. Além disso, quando diversas pessoas têm a chance de olhar uma mesma área do código, torna-se mais freqüente a identificação de oportunidades de melhoria, levando freqüentemente à refatorações mais precisas e eficientes [BECK, 2005; IMPROVE IT, XP].

Um argumento contrário ao não uso desta prática diz que se ninguém é responsável por um pedaço de código, então qualquer um poderá agir irresponsavelmente, e deixar modificações incompletas ou os problemas mais complicados para os outros resolverem. É justamente por causa deste risco que código coletivo é uma prática corolário. É preciso que a equipe desenvolva um senso de coletividade e responsabilidade para garantir que os membros não farão modificações sem se importar com as conseqüências para o todo [BECK, 2005].

Integração continua é outro importante pré-requisito para propriedade coletiva. Algumas horas de programação podem modificar muitas partes de um sistema, se existirem muitas oportunidades de melhoria. Principalmente quando estão espalhadas pelo sistema, incompatibilidades entre mudanças tendem a ter um alto custo de resolução. Reduzir o

intervalo entre as integrações diminui as incompatibilidades e ajudam a manter o custo de integração baixo [BECK, 2005].

### 3.6.6 Código e Testes

Mantenha somente código e testes como artefatos permanentes. Outros documentos devem ser gerados a partir do código e dos testes [BECK, 2005].

Clientes pagam pelo que o sistema faz hoje e pelas funcionalidades que a equipe acrescentar ao sistema amanhã. Quaisquer artefatos que contribuam para estas duas fontes de valor são valiosos. Qualquer outra coisa é desperdício [BECK, 2005].

Código e testes é uma prática que é mais fácil de ser implantada aos poucos. Um processo muito complicado, pode ficar mais leve um pouco de cada vez, à medida que a equipe torne-se mais habilidosa. Quanto melhor a equipe for em design incremental, menos decisões de design precisam ser tomadas com muita antecedência. Quanto mais claro se tornar o ciclo trimestral, em termos de expressar prioridades de negócio, mais fino o documento de requisitos precisa ser [BECK, 2005; IMPROVE IT, XP].

Formalismo interfere negativamente nos passos que levam a geração de valor, mesmo assim, faz décadas que o desenvolvimento tradicional de software acredita no oposto. As decisões mais valiosas em desenvolvimento de software são: O que faremos? O que não faremos? Como faremos o que tem que ser feito? Tomar estas decisões juntas, de forma que elas interfiram umas nas outras, suaviza o caminho até a geração de valor. Eliminar formalismos representados por artefatos obsoletos permite este tipo de melhoria [BECK, 2005].

### 3.6.7 Base Unificada de Código

Deve existir somente uma única base de código. É permitido desenvolver em ramificações temporárias, mas nunca a deixe viva por mais que algumas horas [BECK, 2005].

Múltiplas linhas de codificação são enormes fontes de desperdício em desenvolvimento de software, pois um defeito corrigido em uma das versões da base de código em uso, seja a principal ou não, tem que ser corrigido em todas as outras

ramificações existentes. Quando se descobre que a correção gerou outro erro numa das ramificações, todo o processo tem que ser repetido, talvez por muitas e muitas vezes [BECK, 2005].

Existem razões legitimas para se ter múltiplas versões do código fonte ativas ao mesmo tempo. Algumas vezes, entretanto, isto é feito por simples conveniência, sem se preocupar com as conseqüências em termos gerais. Múltiplas bases de código podem ser reduzidas de forma planejada e gradual [BECK, 2005].

O ideal é, ao invés de criar novas versões do código fonte, tentar corrigir os problemas de design que estão impedindo o trabalho com uma base unificada de código. Caso haja motivos legítimos para se ter múltiplas versões, pode-se aceitá-los, mas é preciso encarar esses motivos como verdades que precisam ser desafiadas, e não verdades absolutas. Pode ser que leve um tempo para desfazer premissas profundamente arraigadas, mas fazê-lo talvez abra a porta para a próxima rodada de melhorias [BECK, 2005; IMPROVE IT, XP].

### 3.6.8 Implantação Diária

Ponha código novo em produção toda noite. Qualquer diferença entre o que está na mesa dos programadores e o que está em produção é um risco. Um programador fora de sincronia com o software em produção corre o risco de tomar decisões sem ter um *feedback* preciso sobre essas decisões [BECK, 2005].

Implantação Diária é uma prática corolário porque tem muitos pré-requisitos. A taxa de defeitos deve ser muito pequena. Gerar um *build* tem que ser fácil e rápido, para isto é fundamental que o processo esteja bem automatizado. As ferramentas para colocar versões do sistema em produção devem ser automatizadas, e incluir a habilidade de implantação incremental e de voltar atrás (*roll back*) em caso de falhas. O mais importante, a confiança entre a equipe e os clientes, deve estar altamente desenvolvida [BECK, 2005; IMPROVE IT, XP].

Implantação diária pode parecer um sonho distante, para equipes que lançam versões com pouca freqüência, como por exemplo, uma vez por ano. Mas o comum é que este tipo de equipe lance uma única versão (*release*) oficial e muitas outras correspondentes

às correções (*patches*). A equipe é capaz de colocar no ar pequenos incrementos de funcionalidades, mas fica envergonhada por ter que fazer isso, ao invés de ver isso como uma oportunidade. Mais de dez *releases* soa bem melhor que mais de dez correções [BECK, 2005].

Existem muitas barreiras para as implantações freqüentes. Algumas são técnicas, como ter muitos defeitos ou precisar arranjar uma maneira acessível de fazer as implantações. Algumas são psicológicas ou sociais, como um processo de implantação tão estressante que as pessoas não queiram passar por ele com freqüência. Algumas são relacionadas ao negócio, como não ter uma forma de cobrar por *releases* freqüentes. Qualquer que seja a barreira, trabalhar para removê-la e então permitir que implantações freqüentes venham naturalmente, irá ajudar a melhorar o processo de desenvolvimento [BECK, 2005; IMPROVE IT, XP].

### 3.6.9 Contrato de Escopo Negociável

Como no desenvolvimento tradicional, projetos XP também definem previamente tempo, custos, qualidade e um escopo que indica o que será feito. A diferença é que este escopo não é engessado a um contrato assinado no início do projeto, e sim negociado à medida que as etapas do projeto progridem. Ou seja, ao invés de assinar contratos longos e generalistas, assina-se uma seqüência de contratos curtos, reduzindo os riscos [BECK, 2005].

Contratos de escopo negociável permitem ao cliente fazer ajuste no escopo para que o software leve em conta seu aprendizado ao longo do projeto, ou mudanças nas circunstâncias. A flexibilidade de poder revisar o escopo freqüentemente mediante negociação é um ótimo mecanismo para manter o contrato alinhado aos interesses dos envolvidos, para encorajar a comunicação e o *feedback*, para garantir que a equipe dedique seus esforços ao que é mais importante para o cliente em cada uma das etapas do projeto, e para permitir que se possa fazer o que parece melhor no momento e não fazer algo inútil apenas porque foi definido inicialmente no contrato [BECK, 2005; IMPROVE IT, XP].

### 3.6.10 Pagar pelo Uso

Em sistemas que seguem o padrão pague pelo uso (*pay-per-use*) o cliente paga a cada vez que usar o sistema. Neste caso, dinheiro é o maior *feedback*, pois não somente é concreto, como pode ser gasto. Conectar o fluxo de dinheiro diretamente ao desenvolvimento traz informações precisas e no momento certo que podem ajudar a guiar o desenvolvimento [BECK, 2005].

O mais comum atualmente é que o cliente pague por cada versão do software. Neste caso o padrão adotado é o pague por versão (*pay-per-release*), que costuma colocar em lados opostos os interesses dos fornecedores e os dos clientes. Os fornecedores sentem-se motivados de forma egoísta, a lançar muitas versões, cada uma contendo o mínimo de funcionalidades possível que sejam suficientes para fazer os clientes pagarem por elas. Os clientes querem poucas versões (por causa do pânico do *upgrade*), cada uma contendo muitas funcionalidades. Este conflito de interesses reduz a comunicação e o *feedback* [BECK, 2005; IMPROVE IT, XP].

Mesmo que não seja possível implementar *pay-per-use*, pode-se tentar implantar um modelo de subscrição, em que o software é comprado mensalmente ou trimestralmente. Com este modelo, a equipe tem ao menos a taxa de retenção (o número de clientes que continuam subscritos) como uma fonte de *feedback* sobre como está indo. Uma mudança ainda menor nos modelos tradicionais de negócios seria tentar fazer os contratos valorizarem mais as taxas de suporte e menos os pagamentos iniciais.

Uma objeção ao modelo pague pelo uso é que os clientes querem custos previsíveis, mas se a vantagem no custo do *pay-per-use* for suficientemente grande, o cliente nem se importará com isso. Uma equipe usando a informação gerada pelo modelo *pay-per-use* será capaz de fazer um trabalho mais efetivo que uma equipe que confia somente no *feedback* gerado pelas receitas de licenças.

## 3.7. Práticas supervalorizadas

A XP, por ser uma proposta nova, ainda é relativamente pouco conhecida até por pessoas que trabalham e conhecem bem outras metodologias de desenvolvimento de software. Por causa disto, ainda é comum o ceticismo em relação à XP, que na maioria das vezes decorrem da falta de conhecimento em relação à metodologia.

Um das formas mais comuns de justificar o ceticismo é tentar restringir a XP a uma ou duas práticas mais polêmicas, e depois tentar argumentar sobre as desvantagens destas práticas, como forma de desaconselhar a adoção da XP como um todo.

Um exemplo disto ocorre com a prática programação em par, uma das mais polêmicas da XP, pelo fato de intuitivamente e numa primeira impressão parecer bastante desvantajosa. Aproveitando-se desta polêmica, é comum ver pessoas tentarem reduzir a XP a apenas esta prática, ao debaterem as vantagens e desvantagens de adotá-la. Estas pessoas estão duplamente enganadas. Primeiro porque apesar de polêmica, a programação em par, usando-se ou não a XP, é uma prática muita bem embasada e que trás ótimos resultados para o desenvolvimento de software. Segundo porque a programação em par, apesar de seus benefícios, não se destaca como uma prática mais importante dentro da XP. Como já foi dito neste trabalho, todas as práticas são complementares e importantes, por isto em conjunto sempre trazem ganhos muito maiores.

Realmente é contra-intuitivo para desenvolvedores tradicionais e para leigos em geral, aceitar que duas pessoas trabalhando num problema produzem mais e melhor do que separadas trabalhando de forma independente em dois problemas. Justamente por isso, a programação em par, assim como muitas outras práticas da XP, é fruto de uma evolução gradativa e baseada fundamentalmente em resultados práticos. Foram profissionais experientes e muito qualificados que descobriram com base nos seus próprios resultados, sucessos e fracassos, que é mais vantajoso programar em par.

# 4. Conclusões

Este trabalho descreveu o paradigma ágil de desenvolvimento de software com foco na Programação Extrema ou XP (Extreme Programming), que é a metodologia de desenvolvimento ágil mais difundida na atualidade.

Com maior destaque no capítulo dois, mas com citações em vários outros pontos, foi descrito o modelo ágil de desenvolvimento, sem focar em nenhuma metodologia específica. Além disso, foi mostrado como o desenvolvimento ágil surgiu, quais suas vantagens e diferenças em relação ao paradigma tradicional, entre outras coisas. O trabalho também procurou mostrar os problemas do desenvolvimento tradicional e porque adotar uma metodologia ágil é uma excelente alternativa para resolver estes problemas.

O capítulo três detalha a metodologia de desenvolvimento ágil denominada XP. Entre outras coisas, foram descritos cada um dos valores, princípios e práticas desta metodologia. Além disso, sempre que possível foi mostrado a relação direta entre a adoção destes valores, princípios e práticas com a solução de problemas freqüentes nos projetos de software, tais como atrasos, orçamentos ultrapassados, funcionalidades desejadas não entregues, ou funcionalidades implementadas e entregues, mas não desejadas pelos usuários, entre muitos outros problemas.

Existiu uma preocupação em embasar solidamente a descrição do desenvolvimento ágil e dos valores, princípios e práticas da XP. O objetivo foi mostrar que o ceticismo natural provocado pelo primeiro contato com algumas das práticas ágeis, pode ser superado facilmente com um pouco mais de conhecimento específico sobre o assunto, pois existem sólidos argumentos a favor de tais práticas.

Devido a grande sintonia da empresa onde o trabalho foi realizado com o modelo tradicional de desenvolvimento e principalmente a inexistência de uma equipe, pois durante todo o tempo existiu somente uma pessoa envolvida diretamente com o desenvolvimento, não houve como aplicar a maioria das práticas da XP no estudo de caso e avaliar o efeito disto no ambiente de trabalho. Apesar disto, a conclusão final é que, apesar de longe do ideal, o estudo de caso não encontrou qualquer inconformidade da teoria descrita neste trabalho em relação a sua aplicação na prática.

Desenvolvimento guiado por testes foi aplicado amplamente e mostrou todas as vantagens descritas neste trabalho (seção 3.5.12 e APÊNDICE A). Apesar disso, vale

ressaltar que é uma prática difícil de ser dominada tecnicamente e que exige uma grande disciplina, principalmente quando se está trabalhando sozinho.

Local de trabalho informativo, apesar de ser uma prática muito direcionada ao trabalho em equipe, como descrito neste trabalho (seção 3.5.3), mostrou-se benéfica para o projeto e não exigiu nenhum grande esforço para ser aplicada. Foram utilizados principalmente cartões divididos por zonas com as histórias, além de lembretes dos mais variados, todos colados na parede.

Também foram usadas intensamente as histórias, e dentro do possível, o cliente esteve o tempo todo envolvido com o projeto, dando e recebendo *feedbeck*. Além disso, a prática ciclo semanal foi também aplicada, mas com ajustes no tempo dos ciclos, já que o projeto foi tocado por somente uma pessoa que além de codificar, tinha que parar constantemente para estudar o funcionamento das práticas e para escrever este trabalho.

Os valores e princípios, quando aplicáveis, foram seguidos e mostraram-se muito benéficos para o desenvolvimento do projeto. Têm argumentação sólida e são muito úteis, até mesmo no dia-a-dia e fora do ambiente de trabalho com software.

A principal contribuição deste trabalho está relacionada à disseminação de informações de qualidade sobre XP, como forma de motivar e facilitar o aprendizado daqueles que por ventura se interessarem pelo assunto. Esta contribuição torna-se ainda mais importante devido à adoção mínima da XP no Brasil em relação aos países desenvolvidos, e também devido ao número muito reduzido de livros em português sobre XP, principalmente livros atualizados que descrevem os valores, princípios e práticas atuais. Esta grande contribuição é a prova de que o trabalho atingiu os resultados esperados, já que apesar de amplo em assuntos abordados, o objetivo principal sempre foi o estudo teórico do desenvolvimento ágil com foco na XP para permitir uma melhor disseminação do conhecimento sobre este assunto.

Para motivar ainda mais a adoção da XP através da disseminação de informações de qualidade sobre o assunto, este trabalho ficará disponível na Internet para o público em geral.

# 5. Apêndice A - Desenvolvimento Guiado por Testes

O desenvolvimento guiado por testes é uma das práticas tecnicamente mais difíceis da XP. Porém é uma prática muito importante, principalmente pela amplitude e efetividade dos seus resultados práticos, mas também pela sua sofisticação teórica. Por estes motivos, a TDD recebeu um destaque maior neste trabalho, representado por este capítulo dedicado ao assunto. Este capítulo descreve a fundamentação teórica por trás da TDD e também apresenta alguns exemplos em código de como utilizar os principais conceitos.

Código limpo que funciona ou “Clean code that Works” [BACK, 2002]. Foi para alcançar este objetivo que o TDD surgiu. Um código será tanto mais limpo quanto melhor for seu *design* e menos erros ele tiver.

Uma forma de sempre ter código limpo que funciona é utilizar desenvolvimento fortemente guiado por testes automatizados, escritos antes do código das funcionalidades, cujo objetivo principal é a melhoria contínua do *design* e a diminuição dos erros. Este estilo de desenvolvimento foi chamado de Desenvolvimento Guiado por Testes ou *Test-Driven Development* (TDD), uma técnica para desenvolvedores e não para testadores.

TDD tem duas regras principais:

- Código novo só é escrito se um teste automatizado falhar;
- Refatoração somente é aplicada em código que possui testes automatizados e sem falhas.

São duas regras simples, mas que tem conseqüências complexas no comportamento individual e coletivo dos desenvolvedores. Elas impõem um ritmo constante e incremental baseado em passos pequenos (“baby steps”). O passo inicial é pensar um pouco sobre o que será desenvolvido e elaborar uma pequena lista dos testes que serão implementados. Depois disso, cada teste deve ser implementado, um por vez, de acordo com os seguintes passos principais, representados por três cores:

1. **Vermelho** – Escreva um pequeno teste, faça-o compilar e veja-o falhar. O teste muito provavelmente não compilará logo após ser escrito, freqüentemente por referenciar algo que não tem nem a assinatura declarada ainda. O teste deve necessariamente falhar, caso contrário estar-se-á implementando algo irrelevante;

2. **Verde** – Escreva o código mais simples possível que seja suficiente para fazer o teste passar. É permitido cometer alguns pecados, como os listados em livros sobre refatoração [FOWLER, 2004]. Deve-se preocupar fundamentalmente em fazer o teste passar;
3. **Refatore** – Torne o código mais fácil de entender e modificar. Deve-se começar eliminando todas as duplicações. O objetivo é evoluir em direção a um *design* simples, reduzindo o acoplamento e aumentando a coesão.

"**Vermelho** / **verde** / **refatore** é o mantra do TDD" [BECK, 2002]. A figura 5.1 sumariza os passos do TDD.

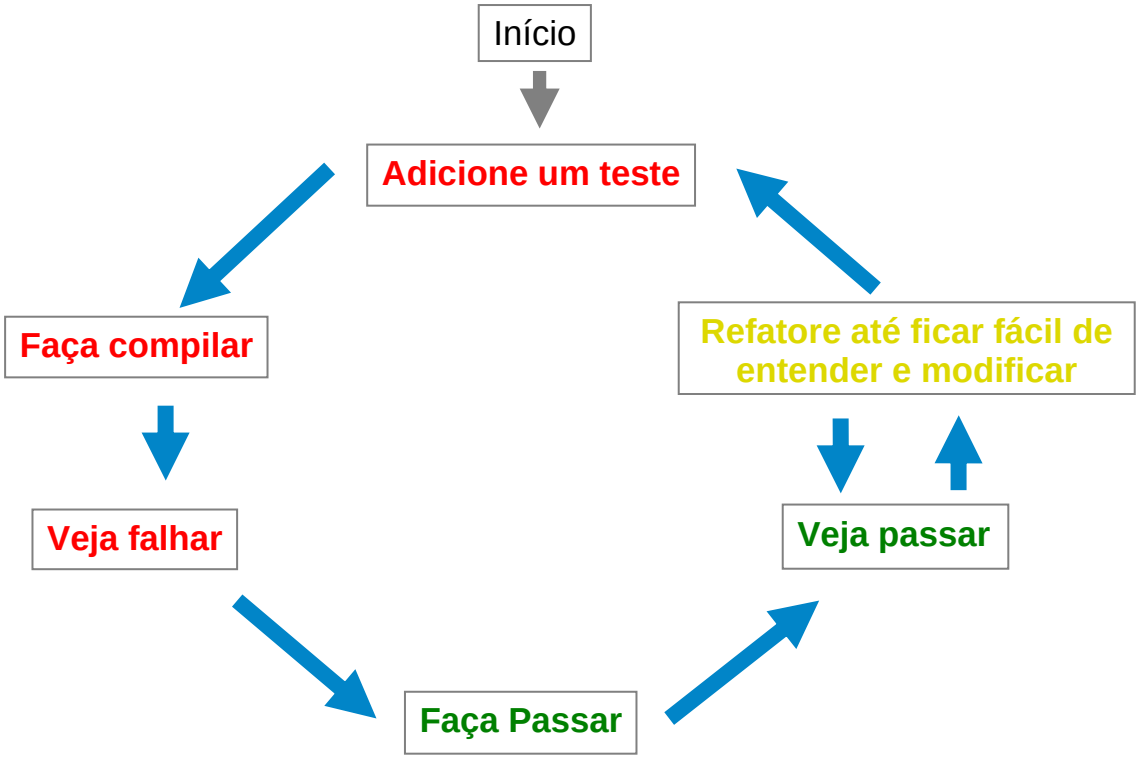


**Figura 5.1 – Ciclo do TDD**

## 5.1. Vantagens do TDD

A seguir, explicamos brevemente os benefícios trazidos pela aplicação do TDD em projetos de desenvolvimento de software.

## 5.2. Design

O objetivo mais importante do desenvolvimento guiado por testes é a melhoria contínua do *design*. Com TDD o projeto evolui junto com os testes, ou seja, em passos pequenos e de forma a atender apenas aos requisitos das funcionalidades que estão sendo implementadas.

Cada teste representa, de forma extremamente pragmática, o reflexo de algo que necessariamente deve ser feito para atingir um objetivo bem definido, e só deve passar (ficar verde) se este objetivo for alcançado. Em função disso, o desenvolvedor é induzido a primeiro focar-se nos objetivos e refleti-los nos testes, e só depois pensar no código que alcançará estes objetivos. Isto traz grandes benefícios para o *design* do sistema e é bastante eficiente, pois reduz a possibilidade de se escrever código desnecessário.

"Escrevendo os testes primeiro, você se coloca no papel do usuário do seu código, ao invés de desenvolvedor. A partir desta perspectiva, você normalmente consegue obter um sentimento muito melhor sobre como uma interface será realmente utilizada e pode ver oportunidades para melhorar seu *design*" [HUNT, THOMAS, 2000, p.115; TELES, 2005, p.105].

Quando está difícil de escrever um teste automatizado, é sinal que existe um problema no *design* e não no teste. Isto acontece porque quanto pior o *design*, mais difícil será escrever os testes. Em contrapartida, quanto menor o acoplamento e maior a coesão do código, mais fácil será testá-lo, ou seja, mais fácil será o trabalho do desenvolvedor. Por isso, o simples compromisso em criar testes automatizados induz o desenvolvedor a preocupar-se continuamente em criar código simples e com o melhor *design* possível [BECK, 2005; SATO; TELES, 2005].

## 5.3. Documentação

Os testes automatizados implementados antes das funcionalidades ajudam a equipe de desenvolvimento a documentar o código, pois são excelentes para expressar intenções de forma clara, além de como os elementos devem se comunicar e acoplar.

Ao usar TDD, o desenvolvedor é estimulado a pensar no código (classes, métodos, etc.) que implementa as funcionalidades, como se ele já estivesse pronto. Desta forma, o único desafio passa a ser documentar no próprio código, através de classes de testes, todos os objetivos identificados.

Além disso, ao contrário da documentação escrita, o teste não perde o sincronismo com o código, a não ser, naturalmente, que se pare de executá-lo [HUNT, THOMAS, 2000; TELES, 2005]. Mesmo a documentação escrita no próprio código na forma de comentários, costuma ficar desatualizada muito facilmente, pois é comum o desenvolvedor prestar atenção apenas no código na hora de modificá-lo. Geralmente os comentários só recebem algum tipo de atenção quando existe algum tipo de dificuldade com o código.

“Não chega a ser surpresa que seja difícil, se não impossível, manter a documentação precisa sobre como o software foi construído. (...) Entretanto, se um sistema possui uma base de testes abrangente que contenha tanto testes dos desenvolvedores, quanto testes dos clientes, estes testes serão, de fato, um reflexo preciso de como o sistema foi construído. Se os testes forem claros e bem organizados, eles serão um recurso valioso para compreender como o sistema funciona a partir do ponto de vista do desenvolvedor e do cliente” [POPPENDIECK, 2003, p.148-149; TELES, 2005].

Pelos motivos explicados, os testes são uma excelente alternativa que pode substituir com vantagens a maior parte da documentação textual, e também uma parcela significativa dos artefatos gráficos usados para documentar um sistema. Com a ressalva de que alguns documentos, como por exemplo esboços UML [FOWLER UML, 2004, pg. 26] continuam sendo bastante úteis, pois comunicam muito rapidamente, são simples e bastante flexíveis por não terem que ser fiéis às regras da UML, e são ágeis por serem criados com ferramentas de desenho leves.

## 5.4. Verificação e Prevenção

A existência de erros e suas reincidências constituem historicamente, um dos problemas mais tradicionais do desenvolvimento de um software. Além disso, desenvolver software de forma verdadeiramente iterativa e incremental, como pregam as metodologias ágeis, gera o risco adicional de se introduzir falhas em algo que vinha funcionando corretamente. A adoção do desenvolvimento guiado por testes minimiza a insegurança em relação à existência ou não de erros no software [BECK, 2003; TELES, 2005].

Os testes automatizados procuram comprovar que as solicitações dos usuários estão sendo atendidas de forma correta. Além disso, os testes verificam se o código está fazendo o que deveria ao longo do tempo, pois é preciso assegurar que o código está fazendo o que se quer, o tempo todo [HUNT, THOMAS, 2003; TELES, 2005].

TDD leva os desenvolvedores a criar uma base de testes automatizados que devem ser executados toda vez que um novo fragmento de código é adicionado ao sistema. Embora isso não impeça a ocorrência de erros, representa um instrumento útil para detectá-los rapidamente, o que agiliza a correção e evita que eventuais erros se acumulem ao longo do tempo [TELES, 2005].

## 5.5. Custos

As fases de desenvolvimento e depuração ocupam a maior parte do tempo de um projeto, sendo que a depuração geralmente só é feita quando o software apresenta um ou mais erros (*Bugs*).

O que torna a depuração muito custosa é o tempo entre a inserção do erro e o momento em que este é detectado. Quanto maior o tempo, maior o custo de depuração, porque para corrigir um problema, o desenvolvedor precisa recuperar o contexto em que este foi inserido, entender uma série de coisas relacionadas, além de descobrir o que pode estar gerando o erro. Ou seja, é preciso re-aprender sobre a funcionalidade e sua implementação para poder corrigir o erro.

O desenvolvimento guiado por testes segue o caminho da prevenção. Para isso, é preciso incorporar hábitos que resultem numa menor probabilidade de ocorrência de erros.

Mesmo assim, é possível que erros ocorram. Neste caso, os testes fazem com que a correção do erro seja mais barata por dois motivos:

1. O teste expõe o erro assim que ele entra no sistema, o que evita a repetição do erro em outras partes do sistema, reduzindo a perda de tempo depurações demoradas e com múltiplas correções.
2. Caso um erro seja introduzido em uma parte do sistema diferente daquela que se está trabalhando no momento, os testes expõem o local exato do erro, permitindo que este seja encontrado e corrigido muito mais rapidamente.

## 5.6. Aprendizado

Para um melhor aprendizado e redução da probabilidade de propagação de erros, é fundamental que o desenvolvedor tenha respostas rápidas para pequenas mudanças. TDD expõe o erro assim que ele é inserido, garantindo uma forma de retorno rápido sobre mudanças efetuadas. Esta detecção rápida do erro permite que o desenvolvedor identifique rapidamente suas causas e conseqüentemente encontre uma solução mais facilmente. Neste caso, a resposta rápida, além de permitir um melhor aprendizado, torna menos provável que o desenvolvedor cometa o mesmo erro no futuro [TELES, 2004].

Os testes representam uma série de exemplos de utilizações do código e, portanto, podem ser bastante úteis para ajudar no aprendizado. “Exemplos não é mais uma maneira de ensinar, é a única maneira de se ensinar” (Albert Einstein).

## 5.7. Produtividade

Devido a grande complexidade envolvida no processo de desenvolvimento de software, é comum ocorrerem erros dos mais variados tipos. É impossível evitar que estes erros ocorram ao longo de um projeto, entretanto é possível fazer alguma coisa em relação à quando estes defeitos são detectados e qual o impacto que causarão no cronograma do projeto.

Observando-se o trabalho de um desenvolvedor, nota-se que boa parte do seu tempo é dedicada à depuração. Trata-se de uma atividade freqüentemente demorada que se repete inúmeras vezes à medida que o desenvolvedor produz mais código para o projeto.

Sempre que um teste detecta uma falha rapidamente, evitam-se longas sessões de depuração que costumam tomar boa parte do tempo dos desenvolvedores. Com mais tempo disponível e melhores oportunidades de aprendizado, os desenvolvedores codificam mais rapidamente e com maior qualidade, ou seja, aumenta-se a produtividade e reduz-se a incidência de defeitos. “Descobri que escrever bons testes acelera enormemente a minha programação(...). Isso foi uma surpresa para mim e é contra-intuitivo para a maioria dos programadores” [FOWLER, 2000, p.89; TELES, 2005, p.109].

## 5.8. Testes

Um teste é um trecho de código com a característica especial de ser escrito para executar outro trecho de código, e determinar se este outro trecho de código está se comportando como esperado ou não.

No TDD, a simples verificação de erros de implementação assume um caráter secundário, pois o objetivo principal dos testes é documentar e verificar as funcionalidades, e não somente a implementação.

Como já foi explicado, quanto mais tempo se passar entre o momento em que o erro é introduzido e o momento em que é identificado, maior tende a ser o tempo de depuração. É o retorno gerado pelos testes que ajuda a reduzir este tempo e, portanto, a acelerar a depuração. Mas, isso só é possível se os testes forem automatizados.

A figura 5.2 mostra a estrutura básica de uma classe de testes do SAM (APÊNDICE C). Neste sistema foi convencionado que a cláusula *using* (*import* em Java) e o nome da classe de teste devem seguir um padrão para que informações sobre a classe testada estejam disponíveis de forma clara e fácil. Segundo o padrão de nomenclatura, a classe de teste da figura 5.2 tem o objetivo de testar a classe CarrinhoMaterial que está num *namespace* (*package* em Java) do sistema chamado Apresentacao.

```
using NUnit.Framework;
namespace SAM.Teste.Apresentacao
{
    [TestFixture]
    public class CarrinhoMaterial_Teste
    {

    }
}
```

**Figura 5.2 - Estrutura de uma classe de teste vazia**

Os dois tipos básicos de testes, no TDD, são os testes de unidade e os testes de aceitação.

## 5.9. Testes de Unidade

Os testes de unidade documentam e monitoram automaticamente as funcionalidades que o desenvolvedor deseja para trechos de código do sistema.

Para conferir se o código está se comportando como esperado, geralmente usa-se uma afirmação, ou seja, uma chamada de método simples que verifica se algo é verdadeiro ou não. Por exemplo, o método *IsTrue* verifica se uma determinada condição *booleana* é verdadeira.

Um teste unitário deve preocupar-se unicamente com o trecho, geralmente um método, de código que está sendo testado. Não deve depender, ou se preocupar, com outras partes do sistema.

Um teste unitário não é bom quando se comunica com o banco de dados, faz algum tipo de comunicação na rede, comunica-se com o sistema de arquivos, não pode ser executado ao mesmo tempo em que outros testes (dependência entre testes) ou é preciso fazer alterações especiais no ambiente para poder executá-lo [FEATHERS, 2004].

## 5.10. Testes de Aceitação

Os testes de aceitação documentam num nível mais alto as funcionalidades (histórias) desejadas para o sistema. Eles funcionam como uma ferramenta de monitoramento automático que alerta sempre que algum requisito deixar de ser atendido.

Devem ser especificados pelos desenvolvedores em conjunto com os clientes para refletir o que os clientes desejam, pois são eles que conhecem o negócio e, portanto, os objetivos que devem ser alcançados. Os testes de aceitação falam numa linguagem que o cliente entende.

Além de documentarem os requisitos, os testes de aceitação promovem o desenvolvimento guiado pelos objetivos a serem alcançados, melhorando o *design*. Além disso, verificam o funcionamento integrado das classes do sistema.

Ao contrário do testes de unidade, os testes de aceitação devem ser executados nas mesmas condições em que a funcionalidade testada será executada dentro do sistema, com todas as dependências que forem necessárias, como por exemplo, do banco de dados ou da rede. Os testes de aceitação “são escritos para assegurar que o sistema como um todo funciona. (...) Eles tipicamente tratam o sistema inteiro como uma caixa preta tanto quanto possível” [FOWLER, 2000, p.97; TELES, 2005, p. 103].

## 5.11. Estudo de Caso

Para refletir em código de teste o comportamento esperado, o desenvolvedor tem que se colocar no papel de usuário do código que será implementado. Como explicado na seção 5.1.2, isto gera reflexos muito positivos no *design*.

O teste da figura 5.3 verifica o comportamento esperado de uma funcionalidade do sistema SAM, descrito no apêndice C deste trabalho. Este comportamento consiste em adicionar um novo material a uma entidade da camada de apresentação chamada carrinho de materiais. Esta entidade será representada no código por uma classe chamada de *CarrinhoMaterial*.

Para este teste (figura 5.3), o carrinho de materiais ainda está vazio, e adicionar um item equivale a chamar um método da classe *CarrinhoMaterial* que acrescenta os dados de um item a uma estrutura interna que armazena itens. O resultado esperado é que a classe

*CarrinhoMaterial*, antes sem nenhum item, passe a ter um item em sua coleção de itens. Tudo isto é refletido através do nome do método de teste e de suas quatro linhas.

```
[Test]
public void AdicionaItem_carrinhoVazio()
{
    ICarrinhoMaterial objCarrinhoMaterial = new CarrinhoMaterial();

    Assert.AreEqual(0, objCarrinhoMaterial.NumeroDeItens);
    Assert.IsTrue(objCarrinhoMaterial.AdicionaItem(_item1));
    Assert.AreEqual(1, objCarrinhoMaterial.NumeroDeItens);
}
```

**Figura 5.3 – Teste guiando a implementação de uma funcionalidade**

As palavras destacadas em vermelho pelo ambiente de desenvolvimento indicam classes, interfaces e métodos que ainda não existem. Neste ponto é obvio que a classe de teste ainda nem compilará.

Partindo da premissa que fazer o teste passar é implementar corretamente a funcionalidade que o teste verifica, fica muito claro pela figura 5.3 o que realmente precisa ser codificado.

O primeiro passo é fazer o teste compilar, para isso é preciso implementar os métodos e classes referenciados no teste, mas que ainda não existem. Neste caso, as palavras destacadas em vermelho pelo ambiente de desenvolvimento indicam claramente o trabalho a ser feito. Ou seja, criar a interface ***ICarrinhoMaterial***; criar a classe ***CarrinhoMaterial***; criar o método ***AdicionaItem*** que recebe um item como parâmetro e o armazena na instância da classe e retorna *true* (verdadeiro) em caso de sucesso; criar a propriedade ***NumeroDeItens*** que informa quantos itens foram adicionados.

Tudo estará correto quando não existirem erros de compilação, ou seja, quando o necessário estiver implementado, e quando os três *Asserts* no final do teste estiverem corretos, ou seja, quando o que foi implementado estiver funcionando como esperado.

A figura 5.4 mostra o código que implementa o comportamento esperado de acordo com o que está descrito no teste da figura 5.4. Não há ainda nenhuma validação, porque o objetivo é fazer o teste passar, nada mais.

```
namespace SAM.Apresentacao.Interfaces
{
    public interface ICarrinhoMaterial
    {
        bool AdicionaItem(ItemCarrinhoVO item);
        int NumeroDeItens { get;}
    }
}
```

**Figura 5.4 – Interface de acordo com o teste da figura 5.3**

```
public class CarrinhoMaterial : ICarrinhoMaterial
{
    private IList<ItemCarrinhoVO>
        _itens = new List<ItemCarrinhoVO>();
    public bool AdicionaItem(ItemCarrinhoVO item)
    {
        _itens.Add(item);
        return true;
    }
    public int NumeroDeItens
    {
        get { return _itens.Count; }
    }
}
```

**Figura 5.5 – Classe de acordo com o teste da figura 5.3**

Para esta mesma funcionalidade existem outras possibilidades de uso que precisam de verificação e por isso serão refletidas em testes, que depois servirão de documentação do sistema. Um teste representa algo concreto que pode ocorrer durante a interação do cliente com a funcionalidade, ou seja, um teste indica precisamente algo que com certeza deve-se implementar. Com TDD diminuí-se drasticamente a possibilidade de codificar algo que não tem utilidade alguma.

Uma das possibilidades de uso para a funcionalidade descrita na figura 5.3, é adicionar um item duplicado. Neste caso, pelas regras do negócio, o item não deve ser armazenado. A figura 5.6 mostra o código do teste que verifica isto.

```
[Test]
public void AdicionaItem_duplicado()
{
    Inicializa Carrinho com 3 itens
    Assert.AreEqual(3, objCarrinhoMaterial.NumeroDeItens);
    Assert.IsFalse(objCarrinhoMaterial.AdicionaItem(_item3_repetido));
    Assert.AreEqual(3, objCarrinhoMaterial.NumeroDeItens);
}
```

**Figura 5.6 – Teste adiciona item duplicado, ainda não implementado**

Para o teste da figura 5.6 não é preciso remover erros de compilação, pois estes não existem. Porém, como já foi dito, eliminar os erros de compilação seria apenas o primeiro passo.

Um novo teste deve sempre falhar, e é o que ocorre com este. Esta é uma regra importante do desenvolvimento guiado por testes. Um teste que passa logo após ser escrito é totalmente desnecessário, pois se o comportamento já existe, então ele foi acrescentado e validado num teste anterior, afinal os testes sempre são codificados antes das funcionalidades. Ou seja, ao criar um teste, sempre deve existir a necessidade de alterar a implementação da funcionalidade para fazer o teste passar.

A figura 5.7 mostra a nova implementação da funcionalidade para atender também a este teste. Neste caso já há alguma validação, mas apenas as exigidas para fazer o teste da figura 5.6 passar, pois o objetivo é sempre fazer o teste atual passar, nada mais. O princípio Passos de Bebê (Capítulo 3, seção 3.3.13) da XP prega exatamente isto.

```
public bool AdicionaItem(ItemCarrinhoVO item)
{
    if (ItemDuplicado(_itens, item)) return false;

    _itens.Add(item);
    return true;
}
```

**Figura 5.7 - Adiciona item de acordo com o teste da figura 5.6**

Algumas das vantagens do desenvolvimento dirigido por testes que podem ser visualizadas nestes dois pequenos exemplos de teste unitário são:

- Melhor direcionamento dos esforços de desenvolvimento, já que o teste diz exatamente o que precisa ser feito, nem mais, nem menos;
- Documentação das funcionalidades, já que é possível visualizar no próprio código de teste, tudo que a funcionalidade faz, ou seja, tudo que foi explicado sobre as figura 5.3 e 5.6;
- Melhoria do *design*, já que construir um teste é planejar focado no objetivo, e isto ajuda a tornar o *design* mais efetivo. Além do mais, *design* ruim dificulta enormemente, quando não inviabiliza, a escrita dos testes;
- Código mais limpo e simples, por ser implementado de forma disciplinada e em pequenos passos, e por atender apenas ao desejado. Nos exemplos anteriores foram dois passos, figura 5.4 e 5.6;
- Maior produtividade, já que um conjunto amplo e bem escrito de testes automatizados informa exatamente onde estão e o porque dos erros, evitando longos períodos de depuração;
- Maior segurança, já que os testes automatizados são como um painel que mostra o que está funcionando e o que não está a qualquer momento que se deseje;

## 5.12. xUnits

Existem diversos frameworks para testes unitários que derivam de uma mesma arquitetura para testes conhecida como xUnit. Os principais xUnits da atualidade são: NUnit (www.nunit.org) para .NET e JUnit (www.junit.org) para JAVA.

É possível escrever testes sem a ajuda de tais frameworks, mas é praticamente impossível desenvolver usando TDD sem eles, principalmente devido a perda de agilidade que isso representa.

Entre as principais funcionalidades oferecidas pelos NUnit e JUnit estão: um conjunto amplo de asserções para testar resultados esperados; interface gráfica e textual para execução de testes; integração com os principais ambientes de desenvolvimento (IDEs); grande comunidade de usuários.

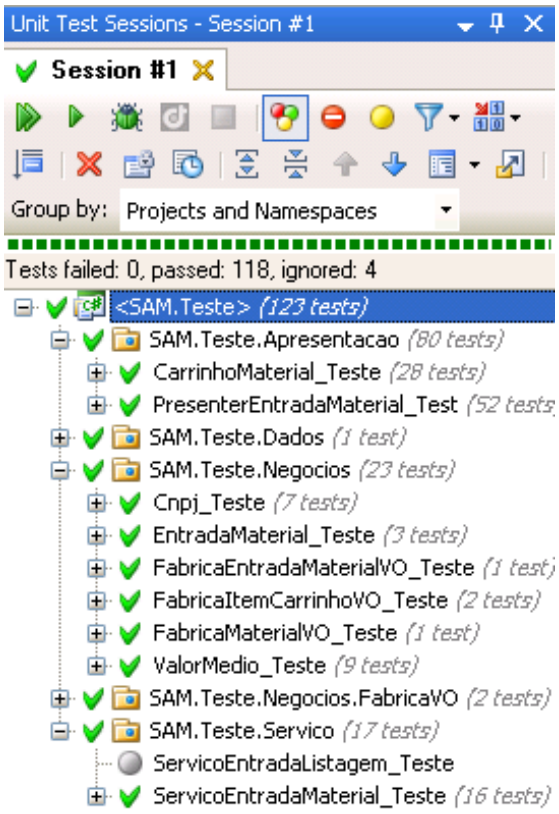


**Figura 5.8 - NUnit integrado a IDE Visual Studio 2005 exibe os resultados dos testes**

## *5.13. Mocks e Stubs*

São objetos usados para testar unidades individuais de código sem testar suas dependências. São sinônimo de objetos falsos que simulam objetos reais para testes.

Um *mock* ou *stub* retorna ou executa apenas o necessário para suprir as dependências de um teste. São criados geralmente a partir de uma interface, seja dinamicamente ou de forma estática. Quando estático, é uma instância de uma classe que implementa a mesma interface que a classe real. Quando dinâmico, são gerados com o auxílio de *frameworks* específicos.

Tanto *Mock Object* quanto *Stubs Objects* podem ser usado para verificar se o resultado esperado está sendo obtido dentro de um teste. A diferença é que os *mocks* vão um pouco além, fornecendo subsídios para verificar se os objetos estão interagindo da forma esperada para atingir determinado resultado. Outra diferença básica é que os *Stubs* normalmente são classes sem comportamento, mas implementadas pelo desenvolvedor. No

caso dos *mocks* não é necessário implementar nenhuma classe extra, pois existem dezenas de *frameworks* que permitem a criação destes objetos dinamicamente e utilizá-los de forma muito rica para os mais variados tipos de testes. Um exemplo de *framework* é o *RhinoMocks* (www.ayende.com/projects/rhino-mocks.aspx).

# 6. Apêndice B – Refatoração

Este capítulo descreve a refatoração, que apesar de não ser explicitamente citada como uma prática, é fundamental para muitas práticas da XP, especialmente para o desenvolvimento guiado por testes. Por este motivo e por ser teoricamente muito rica e interessante, recebeu um destaque especial dentro do trabalho. Apesar do destaque adicional, a refatoração não será descrita em detalhes, com exemplos e estudos de caso, pois o objetivo é a descrição teórica das bases desta prática e de sua relação com a TDD, que é uma prática muito importante dentro da XP.

Refatoração é o processo de alteração de um software de modo que o comportamento observável do código não mude, mas que sua estrutura interna seja melhorada. É uma maneira disciplinada de aperfeiçoar o código que minimiza a chance de introdução de falhas. Em essência, refatorar é melhorar o projeto do código após este ter sido escrito [FOWLER, 2004].

Tornar o código mais fácil de entender e modificar, estes são os objetivos diretos da refatoração. Qualquer alteração no código que não vise estes objetivos não pode ser considerada uma refatoração [FOWLER, p. 53, 2004].

Ao programar, podemos ter código que funciona corretamente, mas que é difícil de entender por uma deficiência qualquer. A refatoração deve necessariamente ajudar a tornar o código mais legível, corrigindo deficiências estruturais, por exemplo.

As técnicas de refatoração incluem, entre outras coisas, metodologias para detectar (“farejar”) possíveis problemas. Martin Fowler e Kent Beck, mostram em [FOWLER, 2004] uma ampla lista de “maus cheiros” e como refatorá-los.

Alguns destes “maus cheiros” são: código duplicado; métodos longos; classes grandes; lista de parâmetros longa; alterações divergentes; cirurgia com rifle; hierarquias paralelas de herança; classe ociosa.

## 6.1. Refatoração e TDD

O processo de Refatoração depende substancialmente da existência de testes de unidade e aceitação. Sem estes testes, fica muito difícil ter a segurança de que a refatoração foi feita corretamente, ou seja, não modificou o comportamento (funcionalidade) do código, nem inseriu qualquer tipo de erro num código que anteriormente estava funcionalmente correto [TELES, 2004; FOWLER, 2004; BECK, 2002; HUNT, THOMAS, 2004].

A Refatoração e a TDD são conceitos intimamente relacionados, ou seja, são dependentes e não funcionam de forma ótima, um sem o outro.

**TDD** pode ser utilizado sem refatoração, mas o código tende a tornar-se difícil de entender e modificar, de forma proporcional à complexidade da funcionalidade que se está implementando. Isto acontece porque com TDD codifica-se primeiro algo muito simples, apenas o suficiente para passar nos testes previamente escritos. Depois disso, melhora-se o código até que este fique fácil de entender e modificar, e a melhor maneira de melhorar um código que está funcionando é via refatoração.

**Refatoração** pode ser utilizada sem TDD, mas neste caso há uma grande possibilidade de ocorrerem problemas. Os testes expõem claramente quais as funcionalidades, facilitando o trabalho de mudar a implementação sem alterar o comportamento. Os testes também funcionam como um sistema de monitoramento automático e bastante eficiente, que alerta tanto sobre erros quanto sobre alterações nas funcionalidades.

Mesmo quando existe a intenção de escrever os testes, mas não como no TDD, ou seja, sem seguir um processo disciplinado, e geralmente depois de implementar as funcionalidades, não se está agindo da melhor forma. Primeiro porque é muito comum esquecer de escrever algum dos testes, segundo porque também é comum neste caso, escrever vários testes desnecessários [TELES, 2004; FOWLER, 2004; BECK, 2002; HUNT, THOMAS, 2004].

Existe o argumento de que essa fortíssima dependência da refatoração em relação aos testes não existe, pois a refatoração foi formalmente definida antes que o conceito de TDD fosse formalizado.

Realmente, a refatoração é mais antiga que a TDD. O primeiro trabalho detalhado e formal sobre refatoração foi a tese de doutorado de William Opdyke [OPDYKE, 1993], publicada em 1993. Já a TDD só alcança um patamar mais formal em meados de 2000.

A grande falha do argumento acima se dá porque TDD é uma forma disciplina de desenvolver software claro e que funciona, baseada fundamentalmente em testes automatizados. Porém os testes sempre existiram, não surgiram com a computação e muito menos com a TDD.

Aplicar refatoração de forma disciplinada, em pequenos passos, seguindo corretamente os bons catálogos, garante altos ganhos e pequenas possibilidades de erros. Porém essas pequenas possibilidades de erros, que são menores que as do desenvolvimento ou manutenção sem refatoração, existem e não devem ser ignoradas. Por isso a dependência em relação aos testes automatizados e conseqüentemente, caso deseje-se algo mais robusto, em relação a TDD [TELES, 2004; FOWLER, 2004; BECK, 2002; HUNT, THOMAS, 2004]

A conclusão é que, do ponto de vista da refatoração, usar TDD em um projeto, além de totalmente compatível, torna o desenvolvimento muito mais fácil, confiável e eficiente. E, como visto no Apêndice A deste trabalho, a recíproca também é verdadeira, ou seja, aplicar TDD auxiliado pela refatoração traz grandes vantagens.

São práticas intimamente relacionadas e complementares, pois ambas objetivam tornar o código mais claro (limpo, fácil de entender e manter) e funcional, sendo que TDD aumenta a chance de funcionalidade contínua e refatoração melhora a clareza do código.

## 6.2. Vantagens Indiretas

A refatoração acaba trazendo outros benefícios, que podem ser considerados indiretos, porque derivam da busca por um código mais fácil de entender e modificar.

### 6.2.1 Melhora do Projeto

**Projetos Implementados:**

- Ao refatorar continuamente um projeto é possível corrigir problemas graves em sistemas já prontos, caso estes existam, levando a uma melhoria significativa em um projeto mal elaborado. Esta característica contraria uma máxima do desenvolvimento tradicional que diz ser praticamente inviável, do ponto do vista do custo / benefício, corrigir problemas de projeto em softwares concluídos.

- A refatoração impede que o projeto se deteriore com as possíveis alterações que tendem a ocorrer ao longo do tempo.

**Projetos em Implementação:**

- A refatoração impede que os problemas presentes no projeto sejam codificados, ou seja, conhecendo-se as técnicas de refatoração, pode-se encontrar e corrigir os possíveis problemas de um projeto durante a própria implementação deste.

É claro que quanto pior elaborado estiver o projeto, maior a quantidade de refatorações necessárias para torná-lo fácil de entender e modificar. Ou seja, um projeto bem planejado só traz benefícios, mas mesmo nos casos em que o projeto é ruim, refatorar continuamente permite detectar e corrigir muitos dos problemas presentes.

### 6.2.2 Aprendizado

Refatorar ajuda a tornar mais fácil a tarefa bastante comum de analisar e entender código legado. É possível através da refatoração, progressivamente ir tornando mais claros e fáceis de entender os trechos de código de um sistema legado, à medida que estes forem sendo analisados e entendidos. Nesse processo, é preciso investir intensamente no aprendizado do que o código realmente faz e aplicar este aprendizado no próprio código, através da própria refatoração.

Ou seja, a refatoração permite documentar de forma incremental e através do próprio código, o aprendizado das funcionalidades de um sistema.

Num nível mais alto, a refatoração permite aprender também sobre a estrutura do projeto, já que à medida que trechos de código vão sendo entendidos, fica muito mais fácil entender o projeto como um todo.

### 6.2.3 Depuração

Refatorar ajuda na melhora e no entendimento do código, conseqüentemente também ajuda no processo de depuração, pois é muito mais rápido e fácil encontrar uma falha num código claro e fácil de entender.

### 6.2.4 Velocidade de desenvolvimento

Quando temos código e projeto mais claros e fáceis de entender, com maior nível de conhecimento pelos desenvolvedores e com menos falhas. Conseqüentemente teremos maior velocidade de desenvolvimento e manutenção, e também menores custos.

# 7. Apêndice C - Sistema de Administração de Materiais

Como explicado no capítulo 1, este trabalho teve como objetivo principal mostrar em detalhes como funciona e quais as vantagens da Programação Extrema ou XP (*Extreme Programming*).

Os conceitos apresentados foram aplicados sempre que possível a um sistema que foi desenvolvido em paralelo ao estudo teórico. O objetivo principal foi visualizar a aplicação da XP durante o desenvolvimento de um sistema integrado de gerenciamento de requisições de materiais e de estoque, batizado de SAM – Sistema de Administração de Materiais – para a Universidade Federal de Sergipe.

O SAM permitirá:

- O cadastro de entradas de materiais, seja via nota fiscal, doação, cessão, empenho, etc., com as facilidades da web;
- Alterações e exclusões, totais ou parciais, nas entradas cadastradas, desde que as restrições de integridades dos dados assim permitam. Neste caso, para que se mantenha a integridade dos dados, é preciso atualizar em cascata todas as entradas do item em questão, realizadas após a entrada deste item;
- O cadastro de requisições de materiais com as facilidades da web, pelos vários setores da Universidade Federal de Sergipe (UFS);
- Alterações e exclusões nas requisições feitas, atendidas ou não. Uma requisição atendida, corresponde a uma saída de material. A mesma lógica de atualização em cascata tem que ser feita para as alterações e exclusões de requisições atendidas;
- Que o Departamento de Requisições de Materiais (DRM) administre as entradas e saídas de materiais de forma ágil, fácil e integrada com os dados do estoque;
- O DRM tenha acesso a estatísticas e relatórios sobre o entradas, saídas e estoque;
- Gerar dados confiáveis para o setor de contabilidade.

A seção 7.1, contextualiza o sistema, enquanto a seção 7.2 mostra sua arquitetura, a seção 7.3 descreve as ferramentas utilizadas.

## 7.1. Contextualização do estudo de caso

A contextualização do estudo de caso será dividida em duas partes. A primeira foca no domínio da aplicação e a segunda nas características técnicas desta.

**:: Contextualização do domínio da aplicação**

Atualmente na UFS, as requisições de materiais feitas ao DRM são realizadas via WEB num sistema implantado a pouco tempo. Este sistema tinha como propósito inicial gerenciar também o estoque, mas acabou não sendo concluído. Por este motivo, o sistema de requisições, apesar de atender bem a um propósito restrito, não possui nenhuma integração com os dados do estoque e não gerencia o atendimento das requisições feitas por ele próprio. Este sistema de requisições, que vamos chamar de SAM-R (Requisições), do ponto de vista deste estudo de caso, é um sistema legado que deverá ser integrado ao SAM.

Para gerenciar o atendimento às requisições e o estoque, atualmente usam-se dois sistemas mais antigos. Em função das limitações técnicas e da falta de integração destes sistemas, o trabalho ainda é fortemente dependente de anotações em papel e da memória dos usuários. Conseqüentemente o controle e a eficiência ficam prejudicados, pois entre outras coisas não há como gerenciar de forma integrada e segura: as saídas de materiais do estoque, ou seja, o atendimento às requisições; as entradas de materiais, ou seja, compras, doações, empenhos, entre outros. Além disso, é comum ocorrerem inconsistências contábeis nos dados destes sistemas. Estas inconsistências, quando detectadas pelo setor de contabilidade, são corrigidas por um funcionário específico, mediante muito trabalho.

O SAM facilitará o gerenciamento da entrada e saída de materiais do estoque. O objetivo principal é tornar mais rápido, simples e seguro o trabalho do órgão que gerencia o estoque na UFS.

**:: Contextualização técnica da aplicação**

O sistema utiliza TDD como a prática central da implementação, mas também utiliza refatoração, cartões com histórias, ciclos curtos, feedback constante do cliente, local de trabalho informativo e outras práticas da XP.

A arquitetura é dividida nas seguintes camadas: Apresentação; Serviço; Negócio; Persistência.

## 7.2. Arquitetura

A divisão em camadas é uma das melhores técnicas para, através da separação de responsabilidades, quebrar a complexidade e aumentar a flexibilidade de um sistema de software. Seguindo este raciocínio, o sistema SAM possui quatro camadas que agrupam componentes com responsabilidades em comum (figura 7. 1).

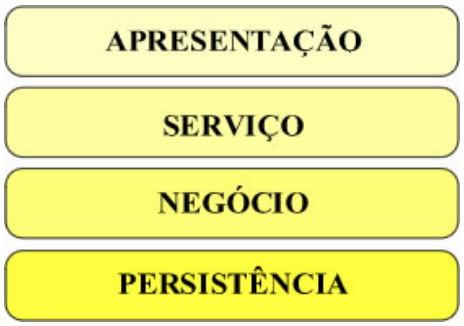


**Figura 7.1 - Camadas**

## 7.3. Camada de apresentação

Nesta camada fica a parte do sistema que interage diretamente com o usuário. A estrutura da camada de apresentação coloca em locais distintos, o comportamento e a apresentação propriamente dita. Esta separação é feita através de dois componentes, a *view*, que interage diretamente com o usuário, e o *presenter*, que cuida do estado do sistema e das respostas às solicitações do usuário [FOWLER, SITE].

O tipo de separação descrita acima foi chamada por Fowler [FOWLER, SITE] de MVP (*Model View Presenter*), e é uma adaptação natural do padrão MVC [POSA, 1996] para ambientes onde a *view* possui pouco poder, como é o caso do ambiente WEB. No MVC, entre outras coisas, a *view* cuida dos eventos e acessa o modelo. No MVP a *view* continua a cuidar dos eventos, mas imediatamente repassa estes ao *presenter*, que é quem tem a responsabilidade de tratá-los e manter consistente o estado do sistema.

O modelo neste projeto engloba as camadas de serviço e negócios, e representa as entidades do mundo real, suas regras e possíveis estados.

No MVP, a *view* torna-se muito mais simples, porque passa a cuidar quase que somente da interação direta com o usuário. Já o *presenter*, segue o padrão *Mediator*

[GAMMA, 1995], o que no caso do MVP significa que cada *presenter* encapsula as interações entre uma *view* e o modelo, ajudando a manter um acoplamento fraco entre eles. Este acoplamento fraco e separação de responsabilidades entre a *view* e o modelo, permite que o comportamento na *view* seja testado muito mais facilmente e também que a aplicação seja portada para outros ambientes visuais sem mudanças no modelo.

***Supervising Controller***

Muitos *frameworks* para *interfaces* com o usuário fornecem a habilidade de mapear facilmente e de forma declarativa a *view* com o modelo, freqüentemente usando algum tipo de *Data Binding* (mecanismo que busca assegurar que as modificações na *view* sejam refletidas no modelo e vice-versa). Como o MVP padrão não prevê uma comunicação direta entre a *view* e o modelo, para aproveitar melhor as facilidades oferecidas por estes ambientes foi necessária uma pequena adaptação. O MVP adaptado para este tipo de ambiente foi chamado por Fowler de *Supervising Controller* [FOWLER, SITE]. Existe ainda outra adaptação do MVP, que Fowler chamou de *Passive View* [FOWLER, SITE].

O *Supervising Controller* será mais detalhado, por ter sido muito utilizado no SAM. A grande diferença deste, em relação ao que foi explicado para o MVP, é que a *view* passa a cuidar também das interações **simples** com o modelo, e o *presenter* cuida somente das interações mais complexas (Figura 7.2).

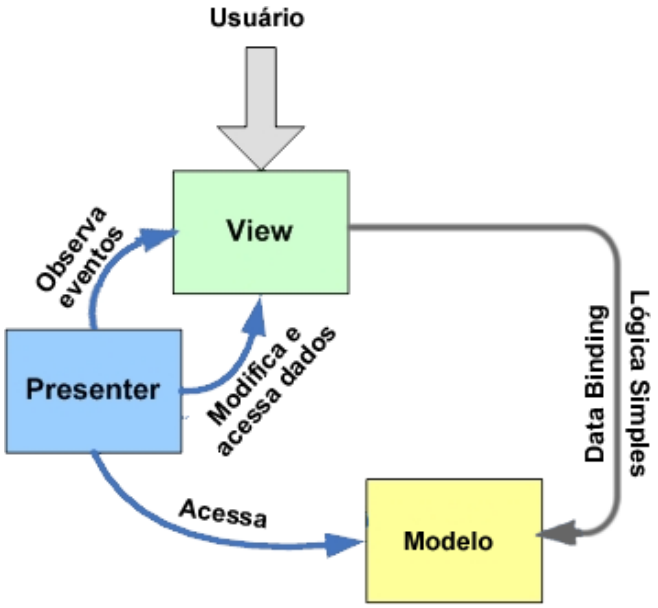


**Figura 7.2 –** ***Supervising Controller***

O código na *view* é muito difícil de ser testado sem a execução manual do aplicativo ou sem a manutenção de scripts que automatizam a execução de componentes da interface do usuário. O padrão *Supervising Controller* não separa totalmente a *view* do modelo, como o MVP, mas separa toda a lógica mais complexa, que corresponde justamente a todo e qualquer código que precise das garantias oferecidas pelos testes automatizados.

O diagrama de classe da figura 7.3 mostra um exemplo de relacionamento dentro do SAM de uma *view*(TelaEntraMaterial), seu *presenter* (PresenterEntradaMaterial) e o modelo (ServicoEntraMaterial). O modelo é representado por uma classe de serviço, porque como é explicado na seção 4.2.2, no SAM todo o acesso ao modelo se dá através de classes de serviço. Através desta figura é possível ter uma visão geral da arquitetura usada no SAM com base no padrão *Supervising Controller*.

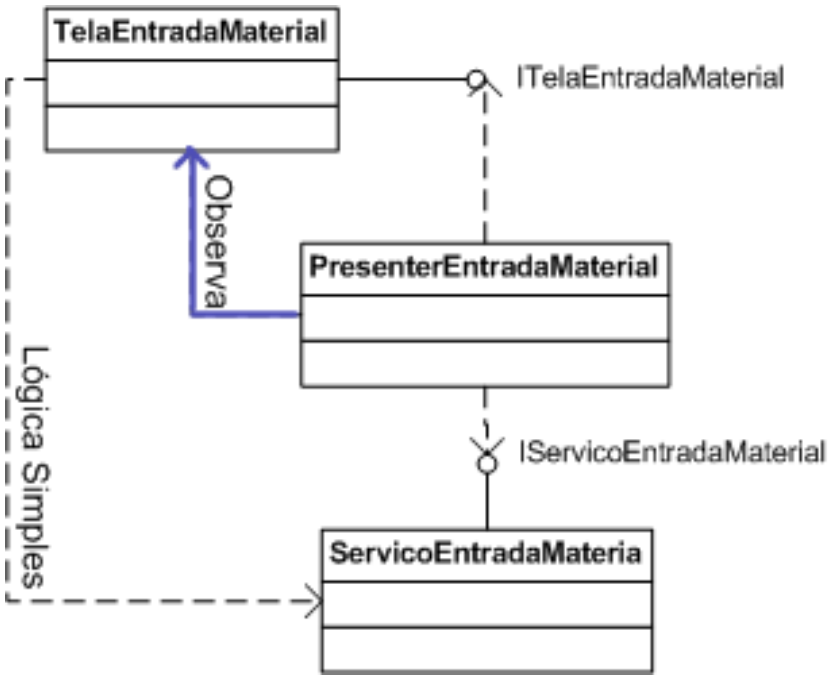


**Figura 7.3 - Diagrama de classes *Supervising Controller* no SAM**

O objetivo da adoção do *Supervising Controller* neste projeto foi aproveitar os recursos do *framework .Net 2.0*, já que este permite que os componentes visuais sejam configurados para acessar o modelo de forma fácil e declarativa, através da IDE (ambiente de desenvolvimento integrado) *Visual Studio 2005*.

### 7.3.1 Supervising Controller para comportamento simples

Esta sub-seção mostra os passos necessários para fazer a *view* acessar o modelo diretamente, ou seja, mostra na prática o *Supervising Controller* para um comportamento simples. O comportamento é exibir num *DropDownList* todos os fornecedores cadastrados.

Logo após uma instância de um componente que pode ser preenchido dinamicamente ser arrastado para a tela, a IDE exibe uma caixa (veja figura 7.4) que dá a possibilidade de selecionar ou criar uma fonte de dados para o componente.

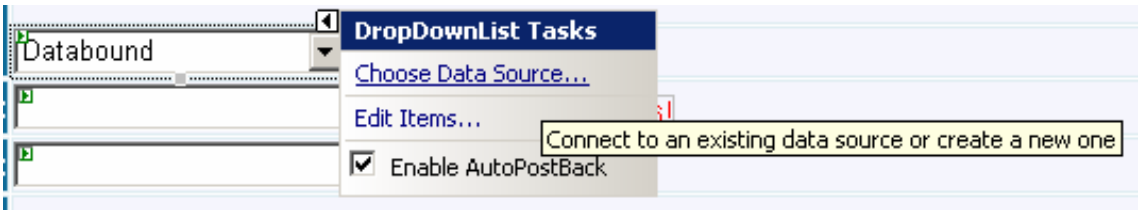


**Figura 7.4 – Componente ao ser arrastado para a tela**

Para detalhar melhor o processo, foi selecionada a opção criar uma nova fonte de dados. Esta opção abre um assistente de configuração com as opções mostradas na figura 7.5. A figura mostra que a IDE *VisualStudio 2005* oferece cinco tipos de fonte de dados. Como o objetivo neste exemplo é obter os dados através do método de uma classe, foi escolhida a opção *Objetct*.

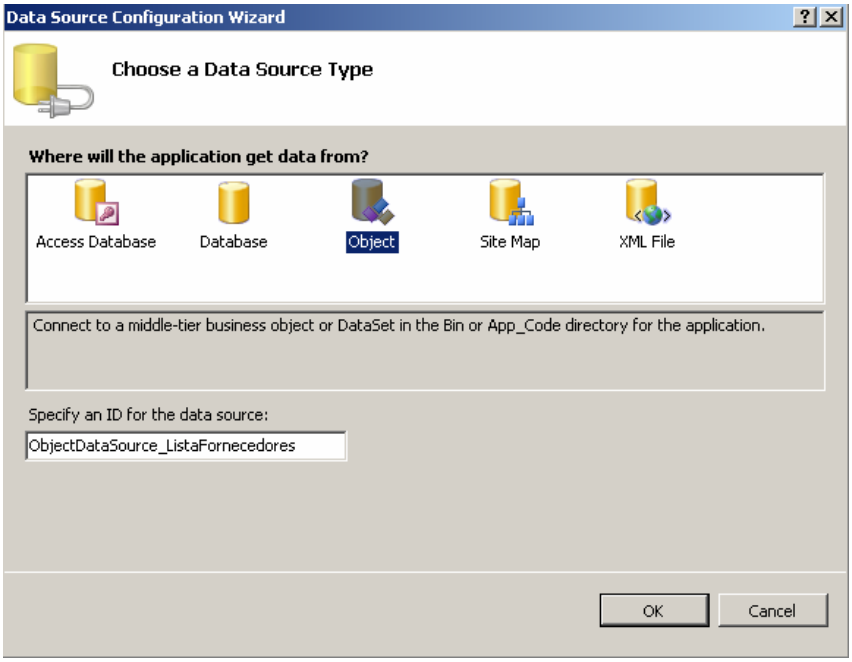


**Figura 7.5 - Escolhendo o tipo da fonte de dados**

O próximo passo é escolher a classe onde está o método que retornará os dados. Isto é mostrado na figura 7.6. Neste caso foi selecionada uma classe da camada de serviço, pois como explicado na seção 7.4, no SAM todo o acesso ao modelo é feito via camada de serviço.

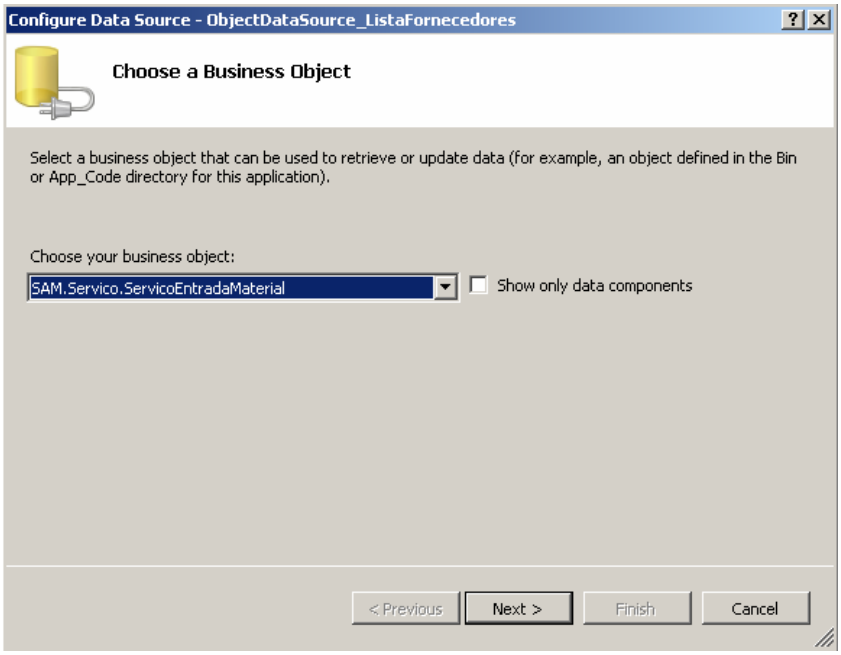


**Figura 7.6 - Escolhendo a classe fonte dos dados**

A figura 7.7 mostra parte da lista de métodos disponíveis na classe ServicoEntradaMaterial, que foi selecionada no passo anterior (figura 7.6).

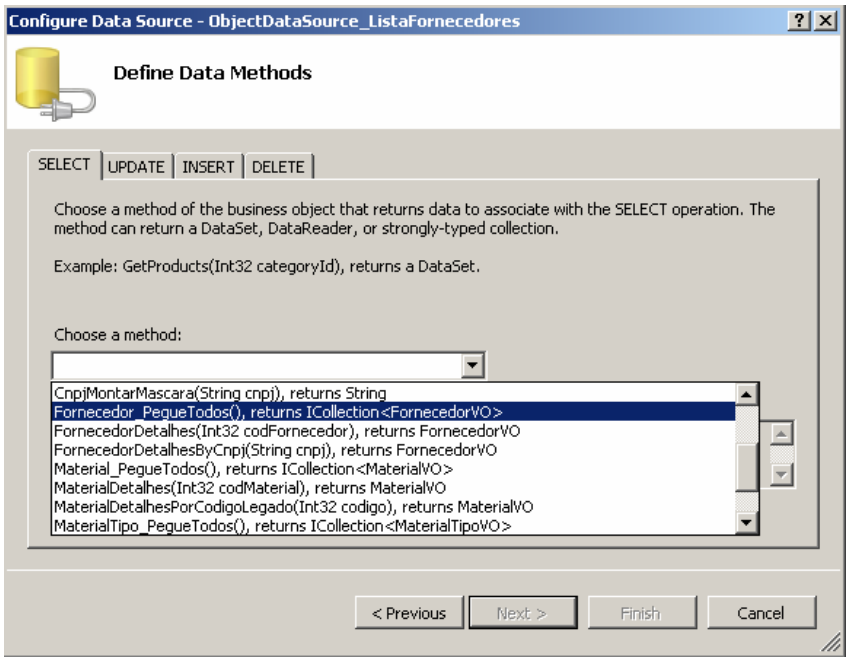


**Figura 7.7 - Parte dos métodos da classe ServicoEntradaMaterial**

A figura 7.8 mostra o método da classe de serviço selecionado. Neste caso foi o método Material_PegueTodos() que retorna uma coleção de objetos MaterialVO. Este é o último passo do assistente para criação da fonte de dados.

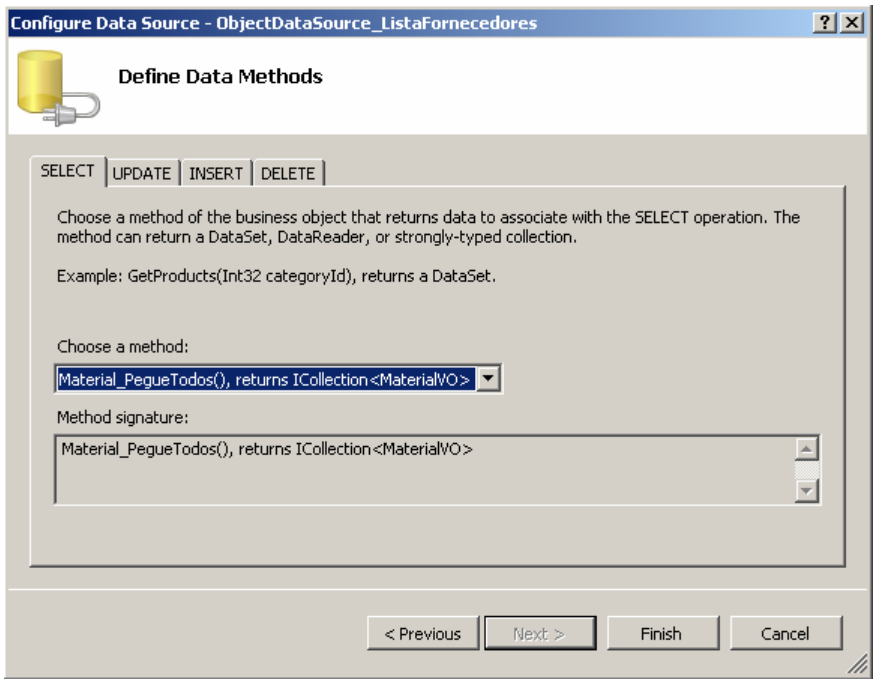


**Figura 7.8 - Método que retorna os dados**

Com a fonte de dados criada, o controle escolhido pode utilizá-la. Neste caso, o assistente mostra esta (figura 7.9) tela pré-configurada para a fonte de dados que acabou de ser criada, pois o assistente foi iniciado logo após o controle ser incluído na tela (figura 7.4).

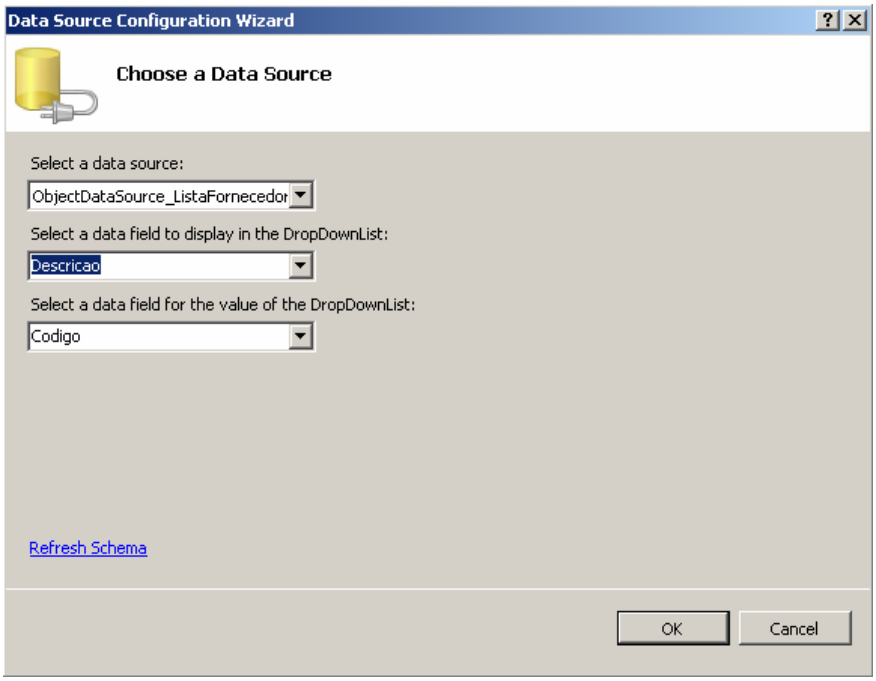


**Figura 7.9 - Configuração do controle *DropDownList***

A fonte de dados ficará disponível para qualquer outro controle na mesma página e é representada pela marca ASP.NET mostrada na figura 7.10.

```
<asp:ObjectDataSource ID="ObjectDataSource_Fornecedores" runat="server"
    SelectMethod="Fornecedor_PegueTodos"
    TypeName="SAM.Servico.ServicoEntradaMaterial">
</asp:ObjectDataSource>
```

**Figura 7.10 - Marca ASP.NET para uma fonte de dados**

A figura 7.11 mostra o controle em execução exibindo dados de alguns fornecedores. Estes dados foram trazidos do banco de dados de desenvolvimento através do método Fornecedor_PegueTodos, como descrito nas figuras desta sub-seção.

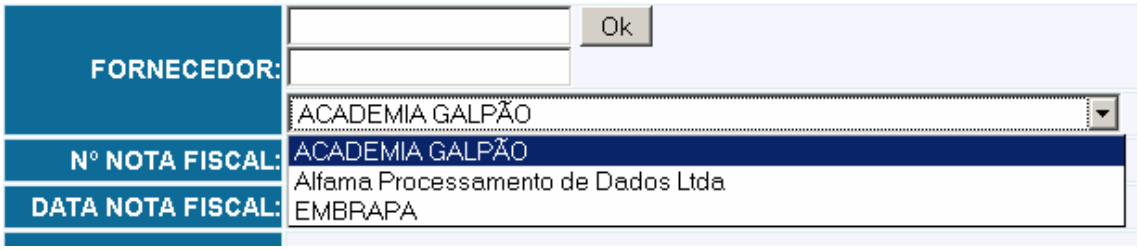


**Figura 7.11 - Fornecedores exibidos na *DropDownList***

É mais vantajoso adotar uma arquitetura que prevê duas formas de relacionamentos entre a camada de apresentação e o modelo, porque as IDEs modernas através de seus assistentes, como mostrado nas figuras acima, tornam a implementação destas funcionalidades simples e intuitivas. Se a separação entre a *view* e o modelo fosse total, este processo seria mais trabalhoso e bem menos intuitivo, pois o comportamento teria que ser codificado através no *presenter*.

A desvantagem é que fazendo desta forma não é possível testar. Por isto, este tipo de facilidade só deve ser utilizada quando o comportamento é simples e pouco crítico para o sistema.

### 7.3.2 Supervising Controller para comportamento não simples

O uso do *Supervising Controller* não significa abrir mão das vantagens do MVP, como a separação de responsabilidades, já que todo o comportamento não trivial continua sendo movido para fora da *view*. Principalmente em aplicações Web, onde o código ligado puramente a *interface* visual pode tornar-se muito complexo, esta separação, além de tornar viável o uso de testes automatizados, deixa o sistema muito mais fácil de entender e modificar.

Esta sub-seção mostra os passos necessários para implementar uma funcionalidade mais complexa seguindo o padrão *Supervising Controller*. Neste caso, a *view* tem um papel passivo, pois sua interação com o modelo é mediada pelo *presenter*.

A funcionalidade desejada é exibir os detalhes de um fornecedor depois que o usuário fornece o código deste.

A figura 7.12 mostra os controles que serão acessados pelo *presenter*. Eles estão na *view*, que no caso do .NET é uma única classe dividida em dois arquivos com o mesmo nome, mas com extensões diferentes.

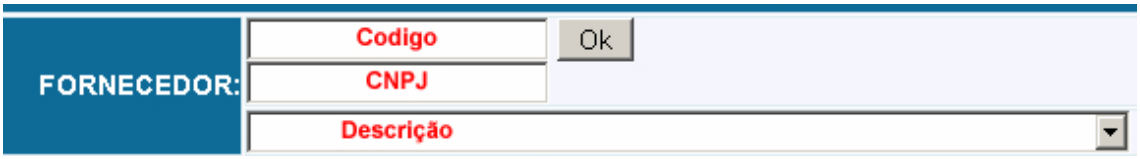


**Figura 7.12 - Controles na tela da aplicação**

Num arquivo com extensão “aspx” ficam os elementos visuais e as marcas ASP.NET, e em outro arquivo com extensão “cs” fica o código fonte em uma das linguagens suportadas pelo *framework*. O *presenter* precisa de assessores (*get* e *set*) declarados numa interface e implementados na *view* para ter acesso aos controles. As figuras 7.13 e 7.14 mostram respectivamente, a assinatura dos assessores na interface ITelaEntradaMaterial e suas implementações na classe TelaEntradaMaterial (*view*).

```
public interface ITelaEntradaMaterial
{
    string FornecedorCodigo { get; set; }

    string FornecedorCnpj { get; set; }

    string FornecedorComboBoxItemValueSelecionado { get; set; }
```

**Figura 7.13 - Assinatura dos assessores**

```
public partial class TelaEntradaMaterial : Page, ITelaEntradaMaterial
{
    public string FornecedorCodigo
    {
        get { return txtFornecedorCodigo.Text; }
        set { txtFornecedorCodigo.Text = value; }
    }

    public string FornecedorCnpj
    {
        get { return txtFornecedorCnpj.Text; }
        set { txtFornecedorCnpj.Text = value; }
    }

    public string FornecedorComboBoxItemValueSelecionado
    {
        get { return cmbFornecedor.SelectedValue; }
        set
        {
            if (value == "" || value == "0")
                cmbFornecedor.SelectedIndex = 0;
            else
                cmbFornecedor.SelectedValue = value;
        }
    }
```

**Figura 7.14 - Implementação dos assessores na *View***

Como mostrado nas figura 7.2 e 7.3, o *presenter* observa os eventos da *view*. Para isto é preciso registrar na *view* o *presenter* como *observer* [GAMMA, 1995]. A figura 7.15 mostra como isto é feito no código.

```
public partial class TelaEntradaMaterial : Page, ITelaEntradaMaterial
{
    ...
    public TelaEntradaMaterial()
    {
       new PresenterEntradaMaterial(this);
    }
    ...
```

**Figura 7.15 - A *view* registra o *presenter* como *observer***

O *presenter* observa a *view* recebida em seu construtor. A figura 7.16 mostra o construtor do presente e um método que associa os eventos desejados na *view* a métodos no *presenter*. Esta associação significa que sempre que o evento ocorrer o método será chamado, ou seja, o comportamento definido no método será executado.

```
public class PresenterEntradaMaterial
{   ...
    public PresenterEntradaMaterial(ITelaEntradaMaterial telaEntraMat)
    {
        _tela = telaEntraMat;
        _servico = ContainerIoC.PegImplementacao<IServicoEntradaMaterial>();
        InscrevaTratadoresParaEventosDaTela();
    }
    private void InscrevaTratadoresParaEventosDaTela()
    {
        _tela.EventoFornecedorCodigoMestreDetalhes +=
                                    OnEventoFornecedorCodigoMestreDetalhes;
    ...
```

**Figura 7.16 - Construtor e método que associa eventos da *view* a comportamentos**

A figura 7.17 mostra o *presenter* atuando como **mediador** entre a *view* e o modelo para o comportamento buscar detalhes de um fornecedor através do código deste. Como mostrado na figura 7.16, este método é chamado quando um evento na *view* é disparado. Especificamente, quando o evento mostrado indica é disparado significa que o usuário forneceu um código de fornecedor na tela.

É no corpo do método mostrado na figura 7.17 que está definido o comportamento desejado. Este comportamento é composto de três ações principais destacadas na figura 7.17 através de quadros vermelhos. A primeira ação é ir até a *view* e pegar o código fornecido pelo usuário. A segunda é consultar o modelo através de um método da camada de serviços para recuperar os detalhes do fornecedor. A terceira ação basicamente é exibir os detalhes do fornecedor na tela.

```
public class PresenterEntradaMaterial
{
    private void OnEventoFornecedorCodigoMestreDetalhes(object sender, EventArgs e)
    {
        ...
        cod = Convert.ToInt32(_tela.FornecedorCodigo);
        ...
        if (cod > 0)
        {
            FornecedorVO fornecedorVO = _servico.FornecedorDetalhes(cod);
            if (fornecedorVO != null)
            {
                _tela.FornecedorComboBoxItemValueSelecionado = fornecedorVO.Codigo.To
                _tela.FornecedorCnpj = _servico.CnpjMontarMascara(fornecedorVO.Cnpj);
                _tela.FornecedorMsgErroCodigo = "";
                _tela.FornecedorMsgErroCnpj = "";
                _tela.NumNotaSetFocus();
            }
        ...
}
```

**Figura 7.17 - Comportamento do *presenter***

O diagrama de seqüência da figura 7.18 mostra num nível mais alto de abstração a seqüência em que as ações ocorrem na execução da funcionalidade mostrada em detalhes nesta sub-seção. Como o foco é a arquitetura da camada de apresentação e conseqüentemente o *Supervising Controller*, o diagrama de seqüência omite algumas ações sem importância para o entendimento da arquitetura, incluindo toda a seqüência de acontecimentos que ocorre abaixo da camada da camada de serviços.

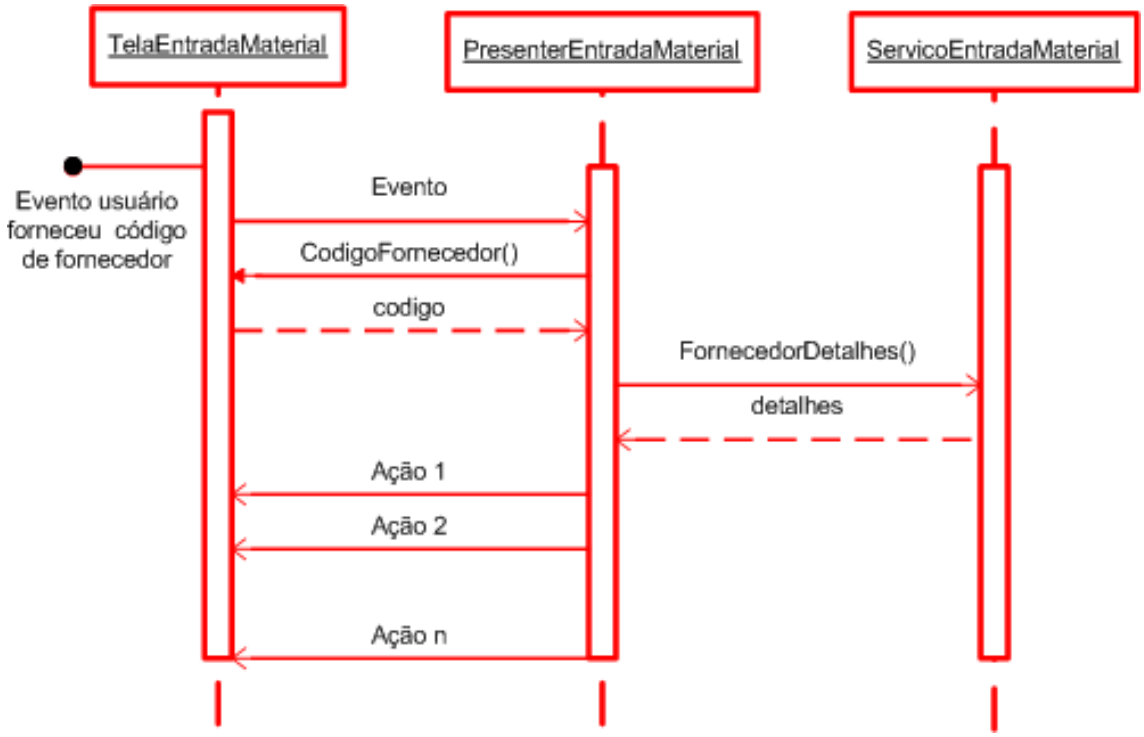


**Figura 7.18 – Seqüência de ações na camada de serviço**

## 7.4. Camada de Serviços

No sistema SAM, a camada de apresentação interage com o modelo unicamente através de uma camada de serviços (Service Layer, [FOWLER, 2003, pág.133]).

A camada de serviço do sistema é muito simples, pois atua apenas como uma fachada (Façade, [GAMMA, 1995, pag.179]) para o modelo, onde todas as classes de negócio estão implementadas. Tudo que a camada de serviço faz é encaminhar pedidos feitos à fachada para objetos de mais baixo nível, atuando essencialmente como uma API (*Application Programming Interface*) simplificada para acesso ao modelo. Este tipo de

acesso é mais eficiente que o acesso direto, porque é totalmente direcionada às necessidades dos níveis mais altos do sistema. Além disso, permite um desacoplamento das classes de negócio, já que estas serão sempre acessadas através da camada de serviços e conseqüentemente não serão criadas e amarradas em função das demandas da camada de apresentação, facilitando em muito a manutenção.

A figura 7.19 mostra uma classe da camada de apresentação usando a a camada de serviço como fachada para acessar funcionalidades implementadas na camada de negócio. O acesso às funcionalidades está destacado na figura através de caixas com borda vermelha.

```
public class PresenterEntradaMaterial
{
    private void OnEventoFornecedorCnpjMestreDetalhes(object sender, EventArgs e)
    {
        _eventoFornecedorCnpjMestreDetalhesDisparado = true;
        string cnpj = _tela.FornecedorCnpj;
        ...
        string cnpjLimpoValido = _servico.CnpjLimpaValida(cnpj);

        if (cnpjLimpoValido != "")
        {
            FornecedorVO fornecedorVO =
                _servico.FornecedorDetalhesByCnpj(cnpjLimpoValido);
        ...
        }
    ...
    }
```

**Figura 7.19 - Classe de serviço sendo usada como fachada**

A figura 7.20 mostra a implementação na camada de serviço dos métodos utilizados na figura 7.19.

```
public class ServicoEntradaMaterial : IServicoEntradaMaterial
{
    public FornecedorVO FornecedorDetalhesByCnpj(string cnpj)
    {
        IFornecedor f = ContainerIoC.PegImplementacao<IFornecedor>();
        ICnpj objCnpj = ContainerIoC.PegImplementacao<ICnpj>();
        string cnpjSoComNumeros = objCnpj.Limpar(cnpj);
        if (objCnpj.Validar(cnpjSoComNumeros))
            return f.PegDetalhesByCnpj(cnpjSoComNumeros);
        else
            return null;
    }
    public string CnpjLimpaValida(string cnpj)
    {
        ICnpj objCnpj = ContainerIoC.PegImplementacao<ICnpj>();
        string cnpjLimpo = objCnpj.Limpar(cnpj);
        if (objCnpj.Validar(cnpjLimpo))
            return cnpjLimpo;
        else
            return "";
    }
```

**Figura 7.20 – Métodos da camada de serviço**

Os dois métodos mostrados na figura 7.20, mesmo sendo muito simples, permitem entender melhor algumas vantagens de se ter na arquitetura uma camada de serviço. O método ***FornecedorDetalhesByCnpj*** utiliza duas classes da camada de negócio chamadas *Fornecedor* e *Cnpj*. Estas classes não tem nada em comum dentro do modelo de negócio, mas precisam trabalhar juntas para suprir esta demanda da *view*. Sem uma camada de serviço, um acoplamento destas classes teria que ser definido em algum lugar do modelo, o que tornaria o modelo mais complexo de implementar e de manter. Já o método ***CnpjLimpaValida*** da classe *ServicoEntradaMaterial*, só existe devido a uma demanda direta da *view TelaEntradaMaterial*. Sem uma camada de serviço, este método teria que ser criado em alguma classe de negócio. Criar métodos em classes de negócios para atender a demandas específicas das *views*, multiplica o número de métodos e torna o modelo mais complexo.

Além disso, é possível utilizar a camada de serviços para prover funcionalidades extras, como por exemplo controle de transações e segurança, sem torná-la mais complicada [FOWLER, 2003].

A figura 7.21 mostra os relacionamentos entre um *presenter*, uma classe de serviço e duas classes de negócios. São os mesmos métodos e classes mostrados nas figuras 7.19 e 7.20.

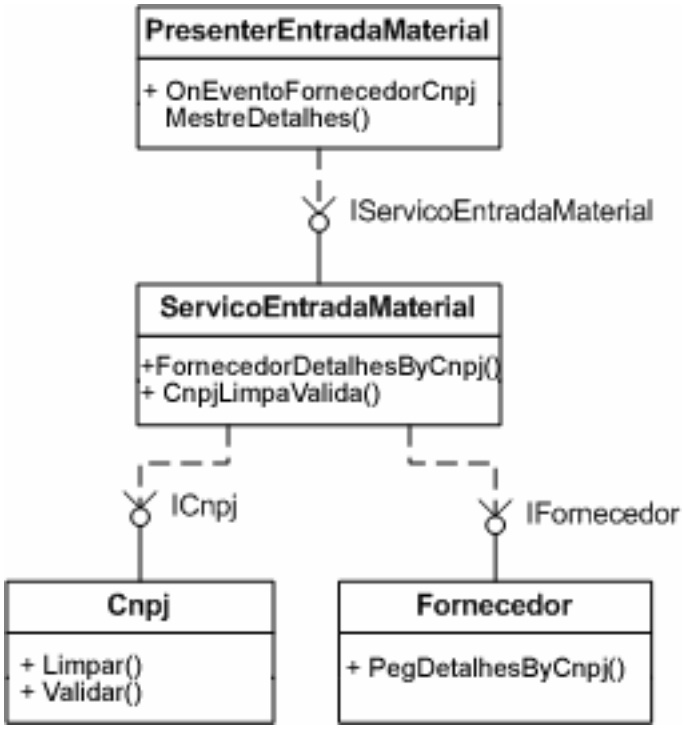


**Figura 7.21 – Diagrama de classe com um das classes da camada de serviço**

## 7.5. Camada de Negócios

A camada de negócios, apesar de simples de usar, ficou conceitualmente um pouco complexa. Esta camada, apesar de usar a biblioteca auxiliar *ActiveRecord* do projeto *Castle* [CASTLE ACTIVERECORD, SITE], não está usando o padrão *Active Record* da forma tradicionalmente descrita nos livros, como por exemplo em [FOWLER, 2003].

Por achar muito interessante e corretas as idéias do DDD (Domain-Driven Design) de Eric Evans [EVANS, 2003], usei alguns de seus conceitos. O principal objetivo foi projetar e implementar a camada de negócios da forma mais simples possível e em função exclusivamente do domínio da aplicação, e com o mínimo de dependência possível da tecnologia ou paradigma usado na persistência. A solução mais elegante e eficiente, sugerida em [EVANS, 2003], é criar repositórios e *agregations roots* para evitar que a complexidade de prover navegação em todos os sentidos inunde o código das classes do

domínio. Porém, devido a pouca experiência e aos prazos, ocorreram algumas adaptações que objetivaram uma maior simplicidade de implementação.

Uma das adaptações foi o uso do padrão *Active Record*, o que deu origem a uma solução mista caracterizada pelo fato de que as classes de negócio não possuem todos os métodos de persistência básicos herdados de uma classe *ActiveRecordBase* [CASTLE ACTIVERECORD, SITE]. Ao invés disso, elas possuem apenas os métodos de negócio que o domínio da aplicação exige, ou seja, não há preocupação direta com a persistência dos dados, pois existe uma outra biblioteca, chamada *Rhino Commons* [RHINO COMMONS, SITE], que entre outras coisas, permite a montagem automática de um repositório(ver [EVANS, 2003]), no topo da hierarquia do NHibernate [NHIBERNATE, SITE], que é automaticamente configurado pelo *Castle ActiveRecord* [CASTLE ACTIVERECORD, SITE] (algo meio complexo a primeira vista, admito).

A escolha do padrão *active record* foi devido à complexidade relativamente pequena da aplicação e a total inexperiência do desenvolvedor com DDD. O *active record* é fácil, prático e bastante eficiente para sistemas pouco complexos. Como o *active record* foi usado com o apoio de um repositório responsável por tornar a persistência transparente, foi possível direcionar as classes de negócio somente para as necessidades do domínio da aplicação, o que não seria possível com o uso do *active record* puro. Isto é vantajoso porque desta forma, as classes de negócio podem ser implementadas da forma mais simples possível e em função exclusivamente do domínio.

Provavelmente não foi a melhor solução, do ponto de vista da simplicidade de entendimento principalmente, mas facilitou muito nos teste automatizados, pois se as classes *active record* implementassem *ActiveRecordBase* ficaria mais complexo de se testar, particularmente de se usar *mocks objects*. Além disso, foi usado um *framework* de inversão de controle que devolve as instâncias dos objetos sempre em função das interfaces. Se as classes de negócio implementassem *ActiveRecordBase*, todos os métodos de persistência que estão nesta classe base ficariam visíveis nos objetos, e não somente os métodos de negócio específicos de cada classe definidos em função do domínio.

A grande vantagem foi sem dúvida o aprendizado, que neste caso, pelo caráter de laboratório do sistema e por se tratar de um estudo de caso, foi muito importante.

## 7.6. Active Record

A principal característica do *Active Record* é que as classes de domínio, além de implementarem a lógica do negócio, realizam sua própria persistência (Figura 7.22). Desta forma, cada classe de negócio fornece os meios para ler ou gravar na estrutura de persistência, os dados que ela própria encapsula.

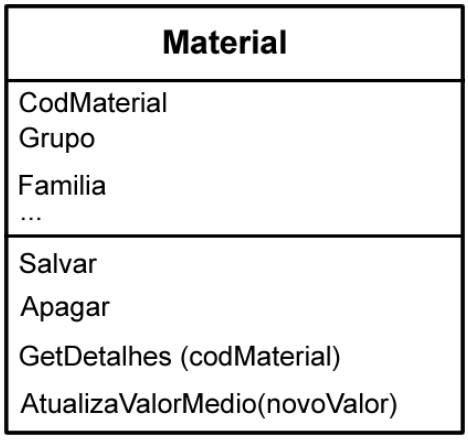


**Figura 7.22 – Classe de negócios Material no padrão *Active Record.***

No SAM, como explicado, as classes do domínio não interagem diretamente com a infra-estrutura de persistência, que é provida por um banco de dados relacional. Para interagirem com o banco de dados, as classes utilizam um repositório, conforme descrito na seção 7.7.

A figura 7.23 mostra a classe *Material* que está na camada de negocio do sistema SAM. Como se pode ver, a classe possui atributos, métodos de negócio com e sem persistência, como *GetDetalhes*, e *AtualizaValorMédio* (obs.: o método *AtualizaValorMédio* é uma mera demonstração para este trabalho).

```
[ActiveRecord]
public partial class Material
{
    private int _codMaterial;
    private MaterialGrupo _grupo;
    private MaterialFamilia _familia;
    private int _numero;
    private MaterialTipo _tipo;
    private string _descricao;
    private MaterialUnidade _unidade;
    private char _restricao;
    private int _estoque;
    private double _valorMedio;
    private int _codigo;

    [PrimaryKey(PrimaryKeyType.Native, ColumnType = "Int32")]
    public virtual int CodMaterial
    {
        get { return _codMaterial; }
        set { _codMaterial = value; }
    }
    [BelongsTo("MaterialGrupo", Cascade = CascadeEnum.All)]
    public MaterialGrupo Grupo
    {
        get { return _grupo; }
        set { _grupo = value; }
    }
```

```
        ...                 ...                 ...
        ...                 ...                 ...
public MaterialVO GetDetalhes(int codMaterial)
{
    Material material = Repository<Material>.Get(codMaterial);

    return FabricaMaterialVO.CriaMaterialVO(material);
}

public bool AtualizaValorMedio(double novoValor)
{
    if (novoValor > 0)
    {
        ValorMedio = (ValorMedio + novoValor)/2;
        return true;
    }
    else return false;
}
```

**Figura 7.23 – Classe Material no padrão *Active Record.***

## 7.7. Camada de Persistência

A função da camada de acesso a dados é comunicar-se com as infra-estruturas externas que sejam necessárias para a aplicação funcionar, e assim isolar as fontes de dados (SGBD, memória, arquivos, *Web Services*, aplicações legadas, etc.) de maneira que, se ocorrerem modificações nas fontes de dados, apenas esta camada precisará ser modificada.

No SAM, a camada de acesso a dados é chamada de **camada de persistência**, por ter como única fonte de dados o SGBD relacional DB2-8 da empresa IBM, que persiste em tabelas os dados do domínio da aplicação.

Como o SAM é uma aplicação orientada a objetos, para minimizar as conseqüências ruins da incompatibilidade conceitual com o mundo dos bancos de dados relacionais, toda a comunicação entre a aplicação e o banco de dados ocorre através de um repositório (descrito em [EVANS, 2003]).

O repositório isola o sistema da infra-estrutura de persistência. Com repositórios, objetos cliente constroem especificações de consultas declarativamente e as submetem ao

repositório. Dados podem ser adicionados e removidos do meio persistente, da mesma maneira como é feito numa coleção de objetos em memória, e o código de mapeamento encapsulado no repositório executará as operações apropriadas de forma transparente. Conceitualmente, um repositório encapsula o conjunto de objetos que representam dados a serem persistidos e as operações executadas sobre eles, dando uma visão orientada a objetos da camada de persistência [FOWLER, 2003, pg. 322]. Além disso, o uso de um repositório reduz o acoplamento em relação a fonte de dados, tornando o desenvolvimento mais simples e o código gerado mais fácil de entender e modificar.

A figura 7.24 mostra um método da classe *Material* que está na camada de negocio do sistema SAM. A linha sublinhada em vermelho está usando o repositório para recuperar um objeto Material persistido no banco.

```
public MaterialVO GetDetalhes(int codMaterial)
{
    Material material = Repository<Material>.Get(codMaterial);

    return FabricaMaterialVO.CriaMaterialVO(material);
}
```

**Figura 7.24 – Código usando o repositório.**

Este exemplo(figura 7.24) dá uma idéia do quanto o uso de repositórios facilita o desenvolvimento, permitindo que o desenvolvedor foque seus esforços na solução dos problemas ligados ao domínio da aplicação, e não em conceitos ligados ao uso de banco de dados relacionais. Além disso é possível notar claramente o quanto o código fica mais enxuto, fácil de entender e manter.

O objetivo do método mostrado na figura 7.24 é retornar os detalhes (dados) de um material. Estão o objeto *Material* retornado pelo repositório é convertido num VO (seção 7.8) para que o cliente do método tenha acesso a esses dados de forma mais leve e prática.

## 7.8. Objetos Valor

Neste projeto, utilizam-se objetos para encapsular os dados transferidos durante as interações entre as camadas (figura 7.25). Estes objetos, chamados de *Value Objects* (VO) são meras estrutura de dados, sem encapsulamento e sem nenhum comportamento associado.

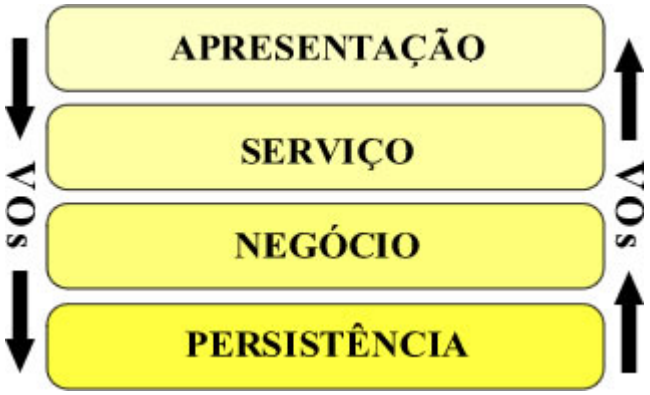


Figura 7.25 – Camadas e VO's

## 7.9. Ferramentas utilizadas

Nesta seção serão apresentadas as ferramentas utilizadas na concepção do SAM.

O sistema foi desenvolvido para o ambiente WEB, utilizando a plataforma .NET 2.0, a ferramenta de desenvolvimento *Visual Studio 2005*, a linguagem de programação C# e o banco de dados DB2 versão 8, pois além de serem, na época da elaboração do sistema, tecnologias comuns para novos projetos no CPD da UFS, elas satisfaziam as condições mínimas desejadas de acordo com as características do projeto.

Além destas ferramentas, foram utilizadas outras que estão mais intimamente relacionadas com o desenvolvimento ágil de software, e particularmente com algumas das práticas da XP. São ferramentas como: *Subversion*, *TortoiseSVN*, *NUnit*, *Rhino Mocks*, *Castle ActiveRecord e Rhino.Commons*.

### 7.9.1 A plataforma *.NET 2.0*

É uma infra-estrutura integrada de desenvolvimento e execução de aplicativos que oferece um ambiente de execução e um *framework* com uma biblioteca de classes comuns a diversas linguagens. Na versão 2.0 o *framework* foi ampliado e bastante melhorado. Todo e qualquer código gerado com *.NET*, pode ser executado em qualquer dispositivo ou plataforma que possua um *.NET Framework*. É uma idéia semelhante à plataforma Java, onde o programador deixa de escrever código para um sistema ou dispositivo específico, e passa a escrever para o *framework .NET*.

A figura abaixo, mostra de forma resumida a arquitetura do *framework .NET*.

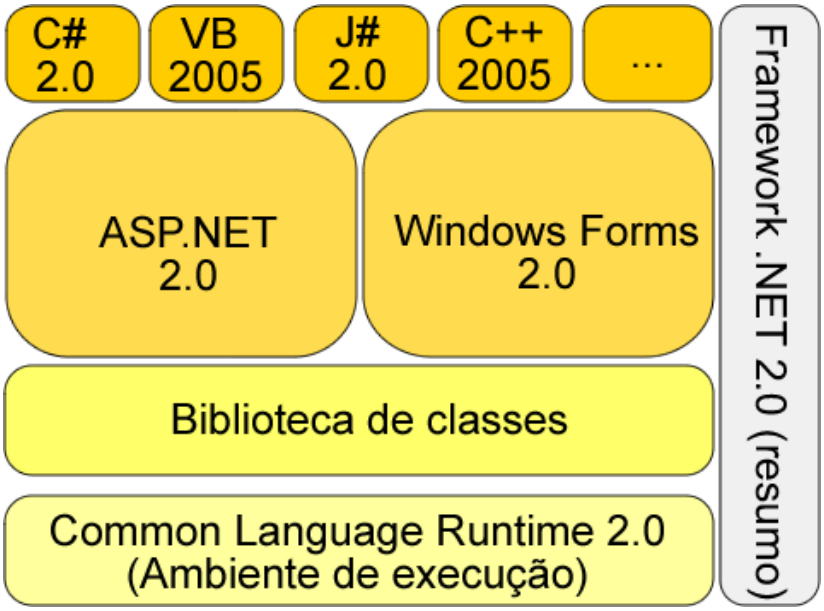


**Figura 7.26 - Resumo da arquitetura .NET**

Na primeira camada estão alguns exemplos de linguagens suportadas nativamente. Neste projeto, foi usada a linguagem C#, que é uma linguagem de programação derivada do C/C++, orientada a objetos, fortemente tipada (*type-safe*) e com sintaxe similar à linguagem Java.

Na segunda camada, estão representados módulos correspondentes ao tipo da aplicação que será desenvolvida. Neste projeto foi usado o *ASP.NET* (*Active Server Pages .NET*), que é o conjunto de classes do *framework .NET* que dá suporte ao desenvolvimento de páginas Web dinâmicas. No *.NET*, aplicações WEB e *Desktop* acessam o mesmo

conjunto básico de classes, herdando todas as suas características, por isso, como qualquer aplicação *.NET*, as aplicações *ASP.NET* podem ser escritas em várias linguagens, como por exemplo, C#, C++, Visual Basic ou J#.

A terceira camada representa a biblioteca básica de classes do *framework*, que é acessível a todas as aplicações desenvolvidas para esta arquitetura. E finalmente, a quarta camada representa o ambiente de execução das aplicações *.NET*, o *CLR*.

A plataforma *.NET* é independente da linguagem de programação porque todos os fontes são compilados para uma mesma linguagem intermediária (IL - *Intermediate Language*) que será compilada em tempo de execução.

A CLR (*Common Language Runtime*) é o ambiente de execução das aplicações .NET. É neste estágio onde o compilador JIT (*Just-in-Time*), que é um componente da CLR, compila a IL e gera a linguagem de máquina na arquitetura do processador.

## 7.10. Histórias do SAM

....

## 7.11. Uma Interação do SAM

...

*7.10 e 7.11 estão no monografia backlog, apesar de já possuírem conteúdo escrito não serão publicados ainda, pois estão incompletos. Além do conteúdo relacionado a estes, existem mais umas 20 páginas que foram cortadas nesta versão por não estarem maduras o suficiente para publicação.*

# *8.* Referências Bibliográficas

AMBLER, Scott. Agile Modeling: Effective Practices for Extreme Programming and the Unified Process. 1. ed. Wiley, 2002, 224 p. ISBN: 0471202827.

AMBLER, Scott; WEB Site; Best Practices for Software Development.
Disponível em: http://www.ambysoft.com/
Última visita em outubro de 2007

ASTELS, David; Miller, Granville; Novak, Mirosla. Extreme Programming - Guia Prático. Campus, 2002.

BECK, Kent; Extreme Programming Explained: Embrace Change. Addison Wesley Professional, US Ed edition (October 5, 1999).

BECK, Kent; FOWLER, Martin; Planning Extreme Programming. Addison Wesley, 2001.

BECK, Kent; Andres, Cynthia. Extreme Programming Explained: Embrace Change, Second Edition. Addison Wesley Professional, 2005.

BECK, Beck. Test Driven Development: By Example (1st edition). Addison-Wesley Professional, novembro, 2002.

BROOKS, Frederick P. The mythical man-month: essays on software engineering, 20th anniversary edition. 2. ed. Reading, MA: Addison-Wesley, 1995. 322 p.

CASTLE ACTIVERECORD, SITE; Projeto de código aberto que implementa o padrão *active Record* [FOWLER, 2003]
Disponível em: www.castleproject.org/activerecord/
Última visita em outubro de 2007

CONSTANTINE, Larry L. The peopleware papers: notes on the human side of software. 1. ed. Upper Saddle River: Prentice Hall PTR, 2001. 346 p.

CARDIM, Ivan Cordeiro. Avaliando o Microsoft Solutions Framework for Agile Software Development em relação ao Extreme Programming. Recife, 2006. Trabalho de graduação. Centro de Informática - Departamento de Sistemas de Computação, Universidade Federal de Pernambuco.
Disponível em: http://www.cin.ufpe.br/~tg/2005-2/icc2.pdf
Última visita em maio de 2007

COCKBURN, Alistair. Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley Professional, 2004, 312 p. ISBN: 0201699478.

DEMARCO, Tom; Lister, Timothy R. Peopleware: productive projects and teams. 1. ed. New York, NY: Dorset House Pub. Co, 1987. 188 p.

DOBBS, Dr.; WEB Site; Revista Dr. Dobbs.
Disponível em: http://www.ddj.com
Última visita em outubro de 2007

DRUCKER, Peter. Desafios gerenciais para o século XXI. 1. ed. São Paulo: Pioneira, 1999. 168 p.

EVANS, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software Addison-Wesley, 2003, 576 p.

FEATHERS, Michael. Working Effectively with Legacy Code (Robert C. Martin Series). Prentice Hall PTR, 2004.

FOWLER, Martin; Kent Beck; John Brant; William Opdyke; Don Roberts. Refatoração - Aperfeiçoando o projeto de código existente; Bookman, 2004.

FOWLER, Martin; UML Essencial, 3ª Edição. Bookman, 2004.

FOWLER, Martin. Patterns of Enterprise Application Architecture, 1ª Edição. Addison-Wesley Professional, 2003.

FOWLER, Martin; SITE pessoal, Supervising Controller.
Disponível em: www.martinfowler.com/eaaDev/SupervisingPresenter.html
Última visita em maio de 2007

GAMMA, Erich; Richard Helm; Ralph Johnson; John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995

HIGHSMITH, Jim. Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. Dorset House Publishing Company, 1999, 392 p. ISBN: 0932633404.

HUNT, Andrew; THOMAS, David. The pragmatic programmer: from journeyman to master. 1. ed. Upper Saddle River: Addison-Wesley, 2000. 321 p.

HUNT, Andrew; THOMAS, David. Pragmatic Unit Testing in C# with NUnit. 1 edition. The Pragmatic Programmers, 2004.

IMPROVE IT, site sobre XP.
Disponível em: http://www.improveit.com.br/xp
Última visita em abril de 2007

JEFFRIES, Ron; ANDERSON, Ann; HENDRICKSON, Chet. Extreme Programming Installed. 1. ed. Boston: Addison-Wesley, 2001. 265 p.

JOHNSON, Jim; "ROI, It's your job". Published Keynote Third International Conference on Extreme Progrmming, Alghero, Itália, 2002.

MACORATTI, Site. Padrões de Projeto: O modelo MVC - Model View Controller.
Disponível em: http://www.macoratti.net/vbn_mvc.htm. Última visita em maio de 2007.

MARTIN, Robert. Site Object Mentor. Agile Software Development, Principles, Patterns, and Practices (Hardcover). Prentice Hall. 2 Sub edition, 2002.
Disponível em: http://www.objectmentor.com/omTeam/martin_r.html
Última visita em junho de 2007

NHIBERNATE, SITE; *Framework* Objeto-Relacional.
Disponível em: http://www.hibernate.org
Última visita em outubro de 2007

OPDYKE, William. Home Page.
Disponível em: http://csc.noctrl.edu/f/opdyke
Última visita em abril de 2007

PALMER, Stephen; FELSING, John. A Practical Guide to Feature-Driven Development. 1. ed. Prentice Hall, 2002, 304 p. ISBN: 0130676152.

POPPENDIECK, Mary e Tom Poppendieck. Lean software development: an agile toolkit. 1. ed. Upper Saddle River, NJ: Addison-Wesley, 2003.

POSA, Buschmann, Frank; Regine Meunier; Hans Rohnert; Peter Sommerlad e Michael Stal. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. West Sussex, Inglaterra: John Wiley & Sons Ltd., 1996.

RHINO COMMONS, Ayende Rahien SITE
Disponível em:
http://www.ayende.com/Wiki/(S(e01wr5zdsus0bu55j1hnr445))/Rhino+Commons.ashx
Última visita em outubro de 2007.

SATO, Danilo. Blog.
Disponível em: http://www.dtsato.com/blog/default/
Última visita em maio de 2007.

SCHWABER, Ken; BEEDLE, Mike. Agile Software Development with SCRUM. 1. ed. Prentice Hall, 2001, 158 páginas. ISBN: 0130676349

STANDISH 2001, The Chaos Report. The Standish Group International, Inc, 2001.
Disponível em:
http://www.standishgroup.com/sample_research/PDFpages/extreme_chaos.pdf.
Última visita em fevereiro de 2007.

TELES, Vinícius Manhães. Um Estudo de Caso da Adoção das Práticas e Valores do Extreme Programming. Rio de Janeiro, 2005. Dissertação (Mestrado em Informática) - Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro.
Disponível em: http://www.improveit.com.br/xp/dissertacaoXP.pdf
Última visita em maio de 2007.

TELES, Vinícius Manhães. Extreme Programming: Aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade. 1. ed. São Paulo: Novatec, 2004. 316 p.

TELES, Vinícius Manhães. Email comunidade XPERS, 01/08/2006.
Disponível em: http://tech.groups.yahoo.com/group/xpers/
Ultima visita em maio de 2007.

WEINBERG, Gerald M. The psychology of computer programming. 1. ed. New York: Van Nostrand Heinhold Company, 1971. 288 p.