1. Introdução

O mercado de desenvolvimento de software vem passando por diversas transformações ao longo das últimas décadas. Software mais confiável, mais barato e menores prazos são, e sempre serão, os objetivos buscados e motivadores destas mudanças. Nesta última década observamos o fenômeno do Desenvolvimento Ágil ocupar cada vez mais espaço no mercado com a proposta de oferecer uma abordagem mais dinâmica e iterativa ao desenvolvimento de sistemas. Essa nova abordagem surgiu como alternativa ao tradicional modelo cascata e baseia-se em quatro valores principais: Indivíduos e interações são mais importantes que processos e ferramentas; Software em funcionamento é mais importante que uma documentação abrangente; Colaboração com o cliente é mais importante que negociação de contratos; Responder a mudanças é mais importante que seguir um plano.

Esses valores foram propostos através do Manifesto Ágil [Agile Manifesto, 2001], produto de uma série de reuniões realizadas por pesquisadores e profissionais renomados da área, em 2001, nos Estados Unidos, e servem como diretriz para as equipes de desenvolvimento que se propõem a produzir software desta maneira.

Diante deste novo paradigma, surgiram as chamadas metodologias ágeis de desenvolvimento de software. Uma metodologia ágil é um conjunto estruturado de processos e técnicas que buscam traduzir os valores propostos pelo Manifesto Ágil através de práticas aplicáveis no cotidiano de um projeto de software. *Scrum* [Schwaber, 2004], *Extreme Programming* [Beck, 2004] e *Lean* [Poppendieck, 2003] são exemplos de metodologias ágeis.

Desenvolver software de maneira mais flexível, ágil e iterativa oferece benefícios tanto para os clientes, quanto para as empresas de desenvolvimento, mas também exige mais capacidade de auto-organização das equipes e, naturalmente, mais proficiência dos profissionais envolvidos [Schwaber, 2004]. Diante desta maior exigência, observamos o surgimento de diversas técnicas de apoio ao desenvolvimento. *Pair-Programming* [Beck, 2004], *Continuous Integration* [Beck, 2004], Análise Estática [Beck, 2004] e *Test-Driven Development* [Beck, 2002] são, por exemplo, algumas das técnicas propostas por *Extreme Programming*, como mostra a figura abaixo, e mais utilizadas no mercado. Destacamos, nesse contexto, *Test-Driven Development*, que vem surgindo

como uma técnica crítica [Nagappan, 2008] para a adoção de metodologias ágeis e é tema principal deste trabalho.

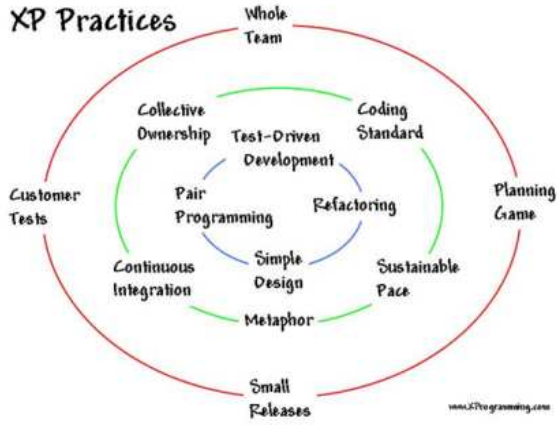

Figura 1: Práticas de Extreme Programming. [XPROGRAMING]

1.1 Test-Driven Development

Test-Driven Development [Beck, 2002] é uma técnica de desenvolvimento de software baseada em pequenos ciclos que alternam entre a escrita de testes e a implementação da solução necessária para que os testes sejam aprovados. A técnica se fundamenta em oferecer um *feedback* rápido para que o trabalho possa progredir com confiança e, desta maneira, nos levar a trabalhar em um nível próximo ao nosso potencial.

O desenvolvimento orientado a testes defende duas regras básicas que, embora muito simples, alteram drasticamente o modo de se trabalhar. São elas:

- Escreva um teste automatizado antes de escrever qualquer código.
- Remova toda duplicidade de código.

O fluxo de trabalho usualmente adotado pelas equipes de desenvolvimento é composto por uma etapa de codificação seguida por uma etapa de testes. A primeira regra propõe uma inversão no fluxo natural de trabalho, onde passamos a ter uma etapa inicial de construção de testes automatizados e, em seguida, escrevemos o código necessário para que os testes sejam executados sem que consigam encontrar falhas.

A idéia por trás da alteração do fluxo natural de trabalho é de que, ao alterarmos a ordem das etapas, não alteramos somente a ordem em que as tarefas são realizadas, mas alteramos também as decisões que são tomadas durante sua realização e, assim, produzimos um melhor resultado.

A segunda regra procura atender ao princípio de que manter cada comportamento do sistema explícito apenas uma única vez em todo o código é a essência de um bom design, desta maneira devemos remover toda duplicidade existente.

Combinando essas duas regras, chegamos ao ciclo de desenvolvimento proposto pela técnica e representado na figura abaixo.

Figura 2: Ciclo de Test-Driven Development

Conforme mostra a figura, o ciclo proposto é composto por três pequenas etapas, descritas a seguir.

A primeira etapa, como propõe a primeira regra, contempla a construção de um teste unitário referente à funcionalidade prestes a ser implementada. Quando o teste é construído, o código necessário para gerar os resultados esperados ainda não existe e, portanto, o teste recém criado naturalmente falhará.

Durante a segunda etapa é realizada a implementação mínima necessária para que o teste recém criado seja aprovado e, ao final desta etapa, o teste, assim como todo o conjunto de testes já existentes, deve ser executado e aprovado normalmente.

A etapa final é reservada para reavaliarmos a maneira como o código está estruturado, de modo a identificar possíveis melhorias e, como defende a segunda regra, remover qualquer duplicação que eventualmente tenha sido introduzida.

Freqüentemente chamamos cada uma dessas melhorias de refatoração que, segundo Martin Fowler [Fowler, 1999], pode ser definida como "*uma modificação realizada na estrutura interna de um software para que ele seja mais facilmente compreendido e os custos de manutenção reduzidos, sem que o seu comportamento externo seja alterado*".

O ciclo se encerra com todos os testes sendo aprovados, com o código funcionando e todas as refatorações previstas concluídas. É importante notar que as funcionalidades e

o conjunto de testes são construídos incrementalmente, iteração a iteração, através de pequenos ciclos que implementam um caso de teste cada. Realizar pequenos ciclos nos permite utilizar o *feedback* resultante da execução dos testes constantemente e, assim, deixar que o resultado dos testes, de fato, dirija o desenvolvimento.

A nova dinâmica proposta exerce um grande impacto na maneira como desenvolvemos, bem como nos resultados produzidos. Existem alguns pontos, detalhados a seguir, que merecem destaque.

O primeiro ponto que destacamos é a necessidade de, antes de iniciar o desenvolvimento de uma funcionalidade, definir qual comportamento deve ser contemplado pelo sistema. A criação dos testes antes da sua solução volta o foco do desenvolvimento para a formalização do comportamento da funcionalidade e não para detalhes da sua implementação.

Test-Driven Development nos apóia e direciona na tarefa de análise [Beck, 2002], forçando a formalização [Bhat, 2006] deste comportamento, delimitando claramente o escopo de cada funcionalidade e dando maior enfoque para as questões mais relevantes.

Construir os testes antes de produzir a implementação da sua solução nos coloca no papel de clientes da interface sendo implementada e, nesse papel, damos maior enfoque para questões como usabilidade e clareza [Beck, 2002] e produzimos um design com alta coesão e baixo acoplamento [Beck, 2001a].

O impacto de uma técnica na produtividade é sempre muito importante, por isso, no desenvolvimento orientado a testes, devemos minimizar, o tanto quanto possível, o esforço para a construção dos testes. Escrever um teste precisa ser simples e não pode consumir muito tempo, portanto, precisamos de interfaces intuitivas, mais coesas e menos acopladas, pelo simples fato de que elas precisam ser facilmente testadas [Beck, 2002]. Um bom design passa a ser uma necessidade e o código testável por construção.

Podemos notar uma grande diferença na produtividade, a curto e longo prazo, quando adotamos o desenvolvimento orientado a testes ou quando utilizamos o modelo tradicional de desenvolvimento, realizando apenas uma única etapa de testes ao final. A tendência é que a produtividade se mantenha constante quando utilizamos *Test-Driven Development*, já que o esforço de testes está incluso a cada iteração do ciclo. Quando adotamos o modelo tradicional, é natural que haja uma maior produtividade a curto prazo, quando não há esforço algum de testes e, quando a etapa de testes for realizada,

haja uma queda acentuada na produtividade. Essa queda brusca é atribuída ao esforço necessário para correção dos defeitos identificados e para a alteração, também, de todos os pontos do sistema que dependessem dos trechos onde os defeitos residiam, o que pode gerar um grande retrabalho.

Podemos visualizar a diferença do comportamento da produtividade, ao longo do tempo, para os dois contextos, na figura abaixo [Hunt, 2003].

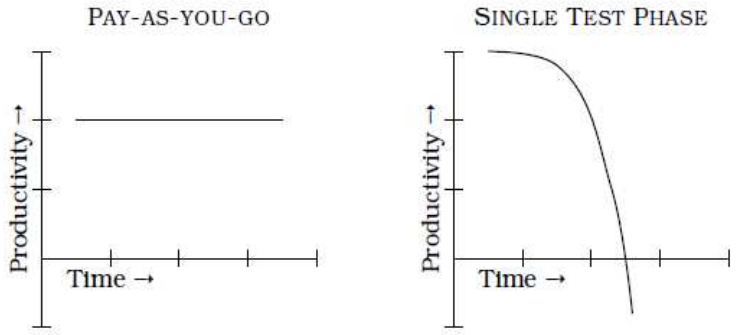

Figura 3: Impacto da adoção de Test-Driven Development na produtivdade

Devemos destacar, também, que a inversão no fluxo de trabalho proposto pela técnica, realizando a construção dos testes antes da implementação da sua solução, implica na construção de testes automatizados para todo o código criado, o que nos leva, inevitavelmente, a um alto nível de cobertura [Beck, 2002].

O conjunto abrangente de testes criados, usualmente chamado de suíte de testes, atua como uma espécie de “armadura de testes” [Feathers, 2004a], protegendo o sistema contra introdução de defeitos durante a manutenção e, desta forma, permitindo que as modificações sejam realizadas com confiança.

Conforme dissemos, uma questão fundamental para a introdução de qualquer técnica é o seu impacto na produtividade da equipe. Quando analisamos *Test-Driven Development*, a idéia básica é que o tempo gasto construindo os testes seja compensado pelo tempo economizado no esforço de manutenção, seja corretiva ou evolutiva. Um estudo dirigido [Nagappan, 2008] na Microsoft e na IBM apresenta uma melhoria entre 40% e 90% na qualidade contra uma pequena queda de produtividade, variando entre 15% e 35%.

Todas as questões discutidas até aqui se aplicam à construção de novos sistemas, mas quando avaliamos a técnica, sob a ótica dos muitos sistemas legados existentes nas organizações, identificamos algumas questões que merecem uma análise mais detalhada.

1.2 Sistemas Legados

Durante o ciclo de vida de um sistema de software existe uma tendência natural de que a sua estrutura interna decaia através das repetidas manutenções, evolutivas ou corretivas, realizadas ao longo do tempo [Eick, 1999].

Ao mesmo tempo, é natural que exista uma constante necessidade de evolução do sistema [Fowler, 1999]. Por exemplo, novas funcionalidades podem ser solicitadas, regras de negócio podem mudar, defeitos podem ser encontrados ou melhorias de desempenho podem se tornar necessárias.

Sistemas legados são sistemas que, embora ainda forneçam serviços relevantes para os seus usuários, atingiram um alto nível de desordem interna e convivem com grandes conseqüências desta desorganização [Feathers, 2004a].

A principal característica de um sistema legado é a sua baixa manutenibilidade. O excesso de dependências, o alto nível de complexidade, o baixo nível de modularização, clareza e legibilidade do código legado são, geralmente, as causas da grande dificuldade de manutenção. Essas indesejáveis características são, de maneira geral, resultado da violação dos princípios de design estabelecidos inicialmente, da imprecisão dos requisitos, das pressões exercidas para entregas e da falta de conhecimento dos desenvolvedores envolvidos [Eick, 1999].

Podemos, também, citar, como características de sistemas legados, a ausência dos desenvolvedores que participaram da sua criação, a inexistência ou desatualização da documentação e das especificações, de modo que a única fonte de informações confiável sobre o sistema é o próprio sistema.

Uma característica muito comum desses sistemas é a inexistência de testes automatizados e, freqüentemente, os testes são somente realizados manualmente. Michael Feathers [Feathers, 2004a], inclusive, define sistema legado como um sistema que não possui uma suíte de testes automatizados.

Quando trabalhamos em sistemas que não possuem testes automatizados, constantemente encontramos dificuldades para determinar o impacto de uma modificação realizada. Diversos pontos podem depender de um comportamento alterado e identificar exatamente todos esses pontos pode ser uma tarefa extremamente dispendiosa. Esse problema se potencializa quando trabalhamos em sistemas legados, visto que, nestes sistemas, a presença excessiva de dependências é muito comum.

A dificuldade de prever o impacto das modificações torna a manutenção do sistema muito propensa a erros, já que a alteração de um comportamento pode ser adequada para uma parte do sistema, mas pode representar um defeito em outra que seja dependente do comportamento alterado.

A alta propensão a erros, neste contexto, estimula uma postura excessivamente conservadora na realização de modificações e refatorações por parte da equipe de desenvolvimento. Essa postura, que reflete uma espécie de temor sobre o impacto das modificações, é bastante prejudicial à evolução do sistema. O temor pela realização de modificações incentiva a criação de duplicidade de código e torna o design do sistema pobre, o que reduz a sua manutenibilidade ao longo do tempo.

A existência de duplicidades no código eleva o nível de retrabalho na realização de modificações, bem como potencializa a inserção de defeitos. Quando possuímos um comportamento duplicado no sistema e desejamos alterá-lo, precisamos realizar a modificação em todos os trechos que o duplicam e, caso algum trecho não seja modificado, o sistema se tornará inconsistente.

Todos esses fatores contribuem para que a manutenção de sistemas legados seja extremamente improdutiva, especialmente pela propensão a erros e pela realização constante de retrabalho.

Embora existam diversas técnicas de prevenção e correção da degradação dos sistemas, como, por exemplo, *Test-Driven Development* e *Refactoring*, o fato é que existem inúmeros sistemas legados, nas condições descritas acima, sendo utilizados no mercado e oferecendo serviços valiosos para seus usuários.

Estudos recentes indicam que cerca de 90% dos gastos com software são voltados para manutenção e evolução de sistemas já existentes [Erlikh, 2000]. Existem, também, estudos que revelam que, em média, as 100 maiores empresas dos Estados Unidos mantêm cerca de 35 milhões de linhas de código cada e a expectativa é que esse número dobre a cada sete anos [Muller, 1994].

É importante destacar que, do ponto de vista do usuário, não podemos considerar um sistema legado algo ruim, afinal ser legado indica que o sistema foi, e continua sendo, útil para os seus clientes [Fraser, 2005].

Diante deste cenário, manter um sistema legado pode se tornar uma necessidade, embora essa tarefa nem sempre seja trivial. Devido às condições encontradas na

organização interna de sistemas legados, como descrevemos anteriormente, realizar a manutenção de um sistema legado pode ser extremamente custoso. Muitas vezes é difícil prever o impacto das modificações realizadas e ter certeza de que o resultado esperado será, de fato, produzido.

Uma excelente alternativa para este contexto seria a construção de testes automatizados que garantissem que uma modificação produzisse exatamente o resultado esperado e que o impacto das modificações ficasse restrito somente à área desejada. Talvez, se fossemos mais além, poderíamos utilizar *Test-Driven Development* na manutenção de sistemas legados.

1.3 Test-Driven Development e Sistemas Legados

Test-Driven Development adota, como princípio básico, a construção de testes automatizados antes das respectivas soluções. Muitas das vantagens obtidas ao se utilizar a técnica são derivadas não somente da extensa presença de testes automatizados, mas também dessa inversão do fluxo de trabalho.

Introduzir *Test-Driven Development* em sistemas legados significa adotar a técnica em sistemas que já possuem milhares de linhas de código funcionando e nenhum teste automatizado, o que, por definição, torna incompatível a sua utilização da técnica, em sua concepção original, neste contexto.

Quando tentamos correlacionar *Test-Driven Development* e sistemas legados com objetivo de, através da construção de suítes de testes, conquistar previsibilidade e confiança na manutenção e evolução de sistemas, observamos que não há um grande interesse acadêmico nesta área.

Realizamos uma série de pesquisas em busca de artigos que discutam aspectos da introdução de *Test-Driven Development* em sistemas legados. Os termos "*legacy system quality control*", "*legacy system test*", "*legacy system test driven*", "*legacy test driven*", "*legacy maintenance*", "*refactoring test*" e "*testability legacy system*" foram selecionados para a realização das buscas. Os mecanismos de busca e bases de artigos acessados foram: Google, Google Acadêmico, Academic Search Premier, ACM Digital Library, Aerospace & High Technology Database, ANTE: Abstracts in New Technologies and Engineering, Applied Science & Technology, Compendex, Computer and Information Systems Abstracts, Ebrary, Guide to Computing Literature, HighWire

Press, IEEE Electronic Library Online, Oxford Journals Online, ProQuest Dissertation, Safari Technical Books, SciELO, ScienceDirect Journals, Scopus e SpringerLink Contemporary.

Infelizmente não encontramos nenhum artigo publicado que abordasse a relação entre *Test-Driven Development* e Sistemas Legados. Encontramos, porém, um livro e dois textos que merecem destaque: "Working Effectively with Legacy Code" de Michael Feathers [Feathers, 2004a], "*Testing Legacy Code*" de Elliotte Rusty Harold [Harold, 2006] e "*Test-Last Development: A primer for unit testing of legacy code*" de Chris Aloia [Aloia, 2009]. O livro "*Test-Driven Development by Example*" [Beck, 2002] também dedica metade de uma página ao assunto.

Recentemente, em julho de 2010, foi publicado o artigo "*Prioritizing Unit Test Creation for Test-Driven Maintenance of Legacy Systems*" [Shihab et al, 2010]**,** que cita o termo *Test-Driven Maintenance*, mas não se aprofunda na discussão de aspectos da introdução de *Test-Driven Development* em sistemas legados, limitando-se a apresentar estratégias para priorização de módulos para introdução de testes unitários.

O livro [Feathers, 2004a] aborda estratégias para introdução de testes unitários em sistemas legados, diante das dificuldades de manutenção encontradas, apresentando um catálogo de técnicas de refatoração e de alternativas para desfazer dependências de código.

Segundo o autor do livro, introduzir testes unitários em código legado nem sempre é uma tarefa trivial. Código criado sem os respectivos testes tende a ter uma interface de difícil testabilidade, acumulando muitas responsabilidades e possuindo muitas dependências.

O autor defende a realização de sucessivas refatorações, de modo a tornar o design existente gradativamente testável. Entretanto, como observa o autor, quando refatoramos código sem que existam os respectivos testes para garantir sua corretude, nos expomos ao risco de inserir novos defeitos. Michael Feathers precisamente observa essa situação e a denomina de Dilema da Refatoração, que ocorre quando precisamos refatorar para tornar o código testável e, então, introduzir os testes, mas para realizarmos as refatorações com segurança precisamos dos testes rodando.

Os dois textos [Harold, 2006] [Aloia, 2009], que também destacamos, estão escritos de maneira informal e discutem alguns aspectos da introdução de *Test-Driven Development*

em Sistemas Legados com abordagens semelhantes. São *posts* de blogs, o primeiro publicado no site da IBM e o segundo no blog do autor. Destacamos, nos próximos parágrafos, alguns pontos abordados pelos autores nesses textos.

Um ponto de comum acordo entre eles é que interromper a manutenção do sistema para um longo período de introdução dos testes automatizados quase nunca será uma alternativa justificável. O esforço necessário para a introdução dos testes quase sempre inviabilizará esta tarefa, ora por falta de tempo, ora por falta de recursos.

[Harold, 2006] e [Aloia, 2009] argumentam que, embora não seja possível criarmos testes para todo o sistema, devemos adotar uma visão mais pragmática sabendo que alguma cobertura de código já nos fornece um *feedback* maior do que nenhuma cobertura, alertando-nos para nunca cairmos na armadilha de acreditar que, se não podemos testar todas as linhas de código, não devemos testar nenhuma.

[Harold, 2006] e [Beck, 2002] concordam sobre a maneira como eventuais defeitos identificados devem ser tratados. A existência de um defeito é a prova da falta de um teste que exercite o trecho em que o defeito reside e evidencie a sua presença. Nesta situação, os autores defendem a criação de um teste unitário que reproduza o defeito encontrado e, então, que o defeito seja corrigido.

[Harold, 2006] destaca que existe uma grande possibilidade destas refatorações se tornarem excessivamente extensas, especialmente quando tratamos código com muitas dependências e responsabilidades. Considerando este cenário, o autor ressalta a importância da delimitação do escopo das refatorações sendo realizadas.

Já [Aloia, 2009] acredita que devemos evitar, ao máximo, modificações no código já existente e, sempre que possível, criar os testes adotando a maneira como o código legado está organizado. A idéia do autor é minimizar os eventuais impactos indesejados que possam ser ocasionados pelas modificações.

Os autores destacam a importância de mantermos o foco em atingir a maior cobertura com o menor esforço necessário e, para que isso seja possível, defendem uma adaptação da técnica.

A técnica, originalmente, propõe que os testes verifiquem a menor unidade possível de código, de maneira a facilitar a localização de eventuais falhas. Quando trabalhamos com sistemas legados pode ser interessante, segundo os autores, criar testes que

verifiquem diversas unidades ao mesmo tempo, maximizando, assim, a cobertura atingida pelo teste.

As idéias propostas pelos autores, embora sejam extremamente valiosas, ainda carecem de uma maior maturidade e consolidação – o que fica claro quando observamos o fato de não terem sido publicadas em nenhum evento científico – sendo baseadas na experiência e opinião dos autores.

1.4 Motivação

A realização deste trabalho surgiu como resposta à busca de uma solução para a grande dificuldade encontrada na manutenção no sistema legado que apresentamos no estudo de caso.

Identificamos *Test-Driven Development* como alternativa para o problema encontrado e, a partir daí, iniciamos o trabalho com objetivo de adaptar e avaliar a técnica para o contexto de sistemas legados, de maneira a determinar quais características, presentes na concepção original da técnica, seriam extensíveis a estes sistemas.

1.5 Estrutura da Dissertação

Organizamos a dissertação em seis principais capítulos. Descrevemos, no Capítulo 2, a técnica de *Test-Driven Maintenance*, bem como o processo de adaptação necessário para que chegássemos à forma descrita. Apresentamos, no Capítulo 3, um modelo para avaliação da utilização da técnica. No Capítulo 4, detalhamos o estudo de caso realizado e apresentamos os resultados obtidos. Realizamos, no Capítulo 5, uma análise consolidada da técnica, considerando os aspectos conceituais e os resultados do estudo de caso. Finalmente, no Capítulo 6, apresentamos a conclusão do trabalho.