NELSON SENNA DO AMARAL

Minimizando as falhas na concepção dos sistemas com o Desenvolvimento Guiado por Testes

São Paulo

2013

NELSON SENNA DO AMARAL

Minimizando as falhas na concepção dos sistemas com o Desenvolvimento Guiado por Testes

Monografia submetida como exigência
parcial para a obtenção do Grau de
Tecnólogo em Processamento de Dados
Orientador: Prof. Dr. Carlos Hideo Arima

São Paulo
2013

DEDICATÓRIA

Dedico este trabalho à minha família, base de toda a minha existência e valor. Sem seus ensinamentos e mais importante, sem seu carinho nunca teria chegado até aqui.

AGRADECIMENTOS

Primeiramente agradecer aos meus pais por tudo que eles representam para mim e o quanto este trabalho representa para eles. Sem o incentivo e os puxões de orelha nunca poderia homenageá-los aqui. É preciso agradecer também ao meu amigo, Diogo Baeder, responsável por me apresentar à prática, tema deste trabalho e seus diversos conselhos sobre desenvolvimento de software e afins. As próximas pessoas que gostaria de agradecer são meus amigos Dairton Bassi, gerente que acreditou na minha capacidade, Rafael Valeira e Eriksen Costa pelos inúmeros conselhos e paciência comigo. Agradecer ao meu irmão que por diversas vezes desempenhou papel fundamental na pesquisa indicando como executar a mesma. Um agradecimento especial à minha namorada Caroline pelo incentivo constante e, por último, mas, nunca menos importante meu orientador Carlos Hideo Arima, por acreditar e direcionar a pesquisa.

RESUMO

A falta de processo adequado de testes custa, segundo levantamento do National Institute of Standards and Technology (NIST), quase sessenta bilhões de dólares à indústria de sistemas nos Estados Unidos. Dado este cenário o Test Driven Development (TDD) emerge como uma prática que une os testes ao ciclo de desenvolvimento, auxiliando o desenvolvedor a produzir sistemas com maior qualidade e menos falhas.

Este trabalho discute o TDD ou desenvolvimento guiado por testes como alternativa para mitigar as falhas em sistemas de informação. São apresentados os conceitos de falha, testes, vantagens e desvantagens da prática e uma análise da sua aplicação em ambientes reais de desenvolvimento a fim de verificar se através de seu uso é possível reduzir falhas.

Palavras-chave: Test Driven Development, testes, falhas, desenvolvimento, qualidade.

ABSTRACT

The lack of adequate process testing costs, according to a survey by the National Institute of Standards and Technology (NIST), nearly sixty billion dollar industry systems in the United States. Given this scenario Test Driven Development (TDD) emerges as a practice that unites testing techiniques within development cycle, helping the developer to produce systems with higher quality and fewer failures.

This paper discusses TDD or test driven development as an alternative to mitigate failures in information systems. Here are presented concepts of failure, testing, advantages and disadvantages of the practice and an analysis of their application in real development environments in order to check that through it usage can reduce crashes.

Keywords: Test Driven Development, tests, failures, development, quality.

Lista de tabelas

Sumário

1 INTRODUÇÃO

A presença dos sistemas nas nossas vidas é cada vez mais evidente. Eles são encontrados desde em equipamentos cotidianos como geladeiras até sistemas de defesa contra mísseis que o utilizam. São utilizados em tarefas simples ou controlando a complexidade envolvida numa transação de bilhões de dólares, realizada por um banco. Embora grandes empresas invistam na indústria de software uma característica não pode ser deixada de lado, eles são produzidos por seres humanos e, estão sujeitos às falhas inerentes aos seres humanos. Os motivos para tais falhas são as mais variadas possíveis e, portanto, de uma maneira simplista é possível dizer que não existe um software que não apresente falhas também. Se, o fator humano não é o único a causar falhas no ciclo de vida do software, ele é um dos únicos fatores presente em todo o ciclo e o mais capacitado a assegurar que o que foi produzido não irá apresentar defeitos.

Um estudo feito pelo National Institute of Standards and Technology (NIST) mostra que erros de software custam anualmente 59.5 bilhões de dólares à economia norte americana e mostra também que um terço deste custo, 22.2 bilhões aproximadamente, poderiam ser eliminados se os softwares em questão fossem testados dentro de uma infra-estrutura que possibilitasse a identificação e correção desses erros rapidamente. Seguindo a linha de prejuízos causados por falhas em software, destaca-se o caso do navio US Vicennes que em 1988 derrubou um Airbus 320, avião comercial para transporte de passageiros, acarretando a morte de 290 pessoas devido a um defeito no software de reconhecimento do navio, que confundiu o Airbus com um avião de combate F-14.

Apesar dos dados alarmantes citados acima não é válido dizer que as empresas, na sua totalidade, não se preocupam com a qualidade do que produzem. Vários fatores que contribuem para que essas falhas ocorram. Fatores que incluem estratégias de marketing, responsabilidade limitada assumida pelas empresas de software além de baixo investimento e tempo cada vez menores em testes e depuração. Intrínseco a estes fatores está a dificuldade em definir e medir a qualidade do software.

Qualidade é um termo subjetivo e pode ser definido por inúmeros atributos. Usuários podem dar pesos diferentes para um mesmo atributo de um determinado

produto, um exemplo simples pode ser como usuários que utilizam um modelo de celular, e diversos aplicativos talvez dêem maior importância ou valor para um aparelho que tenha mais espaço de armazenamento, enquanto, outros que apenas utilizam prioritariamente os serviços de internet dêem maior valor pelo melhor acesso. No caso do software a definição destes atributos foi feita em 1991 quando a International Organization for Standardization (ISO) adotou a norma ISO 9126 como padrão. Este padrão compreende funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade como atributos de qualidade de software. A definição de como calcular valores para os atributos é vaga e, mesmo com os esforços de publicações da Institute of Electrical and Eletronics Engineers (IEEE) tanto em 1988 quanto em 1996 para apresentar alternativas para mensurar estes atributos não se foi capaz de determinar um atributo de qualidade em particular para que as comparações feitas tivessem precisão aceitável ou não causassem ambiguidade entre os resultados. Mesmo sendo difíceis de serem mensurados, os atributos, confiabilidade e manutenibilidade, podem ser usados para avaliar a qualidade geral de qualquer produto de software final (NIST, 2002).

Um dos problemas para assegurar qualidade de sistemas é sua complexidade. Computadores por si só são complexos por possuírem um grande número de estados diferentes e isto já os torna difíceis de conceber, descrever e testar. Por sua vez, sistemas possuem mais estados que um computador em ordens de magnitude (Brooks, 1986).

A complexidade apesar de necessária para o crescimento e manutenção dos sistemas pode levar a uma série de problemas se não controlada.

> “Muitos dos problemas clássicos de desenvolvimento de sistemas vem da complexidade e seu crescimento não linear. A partir da complexidade vem a dificuldade de comunicação entre os membros da equipe, o que leva a falhas de produtos, custos excessivos, atrasos nas entregas. A partir da complexidade vem a dificuldade de enumerar, muito menos compreensão de, todos os estados possíveis do programa e, a partir disso vem a insegurança. A partir da complexidade das funções vem a dificuldade de invocar as mesmas, o que torna o programa difícil de usar. Da complexidade da estrutura vem a dificuldade de abrir os programas para novas funcionalidades sem a criação de efeitos colaterais. A partir da complexidade da estrutura, vem o estado inesperado que causa falhas de segurança. Não só problemas técnicos, mas problemas de gestão, também vem da complexidade. Esta complexidade dificulta a visão geral, impedindo, assim, a integridade conceitual. Torna-se difícil de encontrar e controlar todas as pontas soltas. Ela cria uma enorme fardo no processo de aprendizado e compreensão tornando a rotatividade de pessoal um desastre.” (Brooks, 1986)

O modelo de desenvolvimento em cascata, metodologia de desenvolvimento de sistemas amplamente conhecida e usada, coloca os testes nas suas fases finais,

quando o produto já está completamente desenvolvido. Desta forma um defeito nas fases inicias, como levantamento de requisitos, só será encontrado no final de todo o ciclo de desenvolvimento e, depois que toda a complexidade das regras de negócio já tiverem também sido aplicadas. Um estudo mostrou que uma falha encontrada na fase de especificação custa em média US$139 para ser corrigida, enquanto na fase de desenvolvimento, custa US$7.000, e para descobrir e corrigir uma falha depois da implantação do sistema o custo sobe para US$14.000 (Viega e McManus, 2000). Com a popularização das metodologias ágeis uma nova técnica de desenvolvimento emergiu e tem como principal característica mudar o ciclo de desenvolvimento e passar a desenvolver os testes antes do código. Esta técnica atende pelo nome de TDD, acrônimo para Test Driven Development, e já tem mostrado resultados positivos em pesquisas feitas tanto na indústria quanto na academia, nas quais seu uso de forma consistente resultou na redução de defeitos (George and Williams, 2004), (Bhat and Nagappan, 2006). Para a aplicação consistente desta técnica é necessário que os conceitos de teste e suas variações, as ferramentas necessárias, quais são as vantagens e desvantagens que a técnica pode trazer se adotada.

1.1 Problema

É possível mitigar falhas nos sistemas de informação no seu processo de desenvolvimento?

1.2 Objetivos

1.2.1 Objetivo geral

O objetivo desta pesquisa é analisar o Desenvolvimento Guiado por Testes como alternativa para redução de falhas nos sistemas de informação.

1.2.2 Objetivos específicos

Os seguintes objetivos devem ser alcançados para que o objetivo geral seja atingido:

- Apresentar os diversos tipos de teste;
- Apresentar os conceitos do Desenvolvimento Guiado por testes;
- Analisar resultados das aplicações práticas do Desenvolvimento guiado por testes em empresas que desenvolvem sistemas de grande e médio porte;

1.3 Justificativa

Falhas nos sistemas de informação causam perdas, como já foi dito na introdução. Além dos custos e esforço para a sua correção, as falhas implicam em falta de credibilidade e confiança por parte do cliente, gerando mais perdas, sejam elas nos acessos – caso o serviço seja virtual – ou nas vendas de novas licenças.

A estrutura do Desenvolvimento Guiado por Testes torna possível agregar novas funcionalidades que sejam coerentes as suas definições além da correção de falhas já existentes através de testes de regressão. Além disso, a técnica promove o aumento da segurança do desenvolvedor que pode a qualquer momento executar os testes e ter certeza que nenhuma nova falha será adicionada.

Apesar destas características, o Desenvolvimento Guiado por Testes não é utilizado em larga escala. Além de ser relativamente novo, o Desenvolvimento Guiado por Testes não segue nenhum modelo conhecido ou tradicional de desenvolvimento, o que torna sua adoção mais difícil por grandes corporações.

Portanto, é necessário que o Desenvolvimento Guiado por Testes seja analisado como um meio de minimizar falhas nos sistemas da maneira correta, evitando que o mesmo se torne mais oneroso que a metodologia de desenvolvimento já praticada e que traga os benefícios anunciados pela técnica à empresa que o adote.

1.4 Metodologia

A metodologia adotada para este trabalho consiste em uma abordagem indutiva. Esta metodologia foi escolhida pois, a proposta para resolução do problema é uma alternativa, já que ela não pode ser considerada a única abordagem para o problema apresentado, portanto, não constitui uma premissa maior ou, generalização, sendo impossível adotar abordagens dedutivas. Segundo Gil (1991) o método indutivo parte do particular e tem como produto a generalização a partir da observação de casos concretos suficientemente confirmadores desta.

> “Parte-se primeiro para observação de fatos e fenômenos cujas causas se deseja conhecer. A seguir, procura-se compará-los com a finalidade de descobrir as relações existentes entre eles. Por fim, procede-se à generalização, com base na relação verificada entre os fatos ou fenômenos” (Gil, 1991).

O caráter dessa pesquisa é teórico e qualitativo. Através do levantamento bibliográfico são expostos os conceitos presentes na técnica apresentada como solução do problema. Primeiro define-se o que é sistema e como obter qualidade através de métricas estabelecidas pelo padrão ISO/IEC 9126. Depois se define o que é um defeito e, exemplos de suas causas são apresentados, além de estratégias para lidar com os mesmos. O próximo capítulo trata de testes em sistemas e suas variações apontando estes como maneira de prevenir defeitos e assegurar a qualidade do sistema. Explica-se o desenvolvimento iterativo e incremental no capítulo seguinte como uma maneira de combater a complexidade dos sistemas de informação. Com a base teórica definida o capítulo 6 aborda o desenvolvimento guiado por testes. Explicam seus conceitos, seu funcionamento, vantagens e desvantagens e, por fim aplicações da técnica em empresas para avaliar seus resultados e usá-los para analisar se o emprego do desenvolvimento guiado por testes é capaz de resolver o problema apresentado e suas causas.

1.5 Estrutura do trabalho

Para alcançar os objetivos desta pesquisa, o trabalho foi dividido em sete capítulos.

O capítulo 2 apresenta a definição de *software* e o modelo proposto pelo consórcio ISO/IEC para qualidade de *software*. São abordadas as categorias de características e subcaracterísticas de qualidade e algumas métricas utilizadas para a avaliação de um produto de *software*.

O capítulo 3 estuda as causas, conseqüências e também estratégias para lidar dos defeitos de *software.*

O capítulo 4 aborda testes de *software*. Primeiro define-se o que é teste no contexto de um sistema de computador, depois são mostradas as vantagens que os testes trazem para este. Após as definições são explorados os métodos e níveis de teste mais utilizados e difundidos.

O capítulo 5 faz referência ao desenvolvimento incremental e iterativo. Esse método é apontado como possível solução para a complexidade encontrada nos sistemas de computadores atuais. Além de apresentar suas definições e exemplos, o capítulo também aborda como a combinação das duas técnicas pode evitar que defeitos sejam encontrados apenas na fase final do produto, aumentando assim sua qualidade.

O capítulo 6 estuda a técnica do desenvolvimento guiado por testes ou TDD. São abordadas aqui as características, vantagens e desvantagens do uso da mesma para o desenvolvimento de sistemas.

O capítulo 7 apresenta os estudos feitos com o desenvolvimento guiado por testes em indústrias pelo mundo. São abordados os resultados desses estudos e a partir destes tenta-se comprovar o aumento de qualidade do produto de *software*, através das métricas propostas pelo consórcio ISO/IEC, com o emprego do desenvolvimento guiado por testes.

Por fim, o capítulo 8 conclui a pesquisa apresenta seus resultados, restrições e contribuições para a continuação ou desenvolvimento de novas pesquisas relacionadas a esta.

2 SOFTWARE E QUALIDADE DE SOFTWARE

2.1 O que é software?

Software é qualquer conjunto de instruções que podem ser lidas por máquinas (em forma de um programa de computador na maior parte das vezes), que direciona o processador do computador a executar operações específicas. O termo é usado em contraste ao hardware, que são as partes físicas as quais carregam as instruções. Ainda, segundo a IEEE, softwares são programas de computador, procedimentos, documentação e dados pertencentes à operação de um sistema computacional.

Os computadores modernos possuem uma variedade de softwares para as mais diferentes tarefas. Eles podem ser classificados em três categorias diferentes:

- Software de aplicação: Software concebido para atender às necessidades de um usuário; por exemplo, software para processamento de textos ou para navegação na internet;
- Software de sistema: Software construído para facilitar a operação e manutenção de um sistema computacional e seus programas associados; por exemplo, sistemas operacionais como Linux Ubuntu e Microsoft Windows;
- Software de suporte: Software que ajuda no desenvolvimento ou manutenção de outro software, por exemplo, compiladores e ambientes de desenvolvimento integrados, IDEs, como Eclipse.

2.2 Qualidade de software

Um dos grandes problemas com o qual a engenharia de software se defronta é medir qualidade. A ISO/IEC 9126 (NBR 13596) fornece um modelo que define seis amplas categorias de características de qualidade de software que possuem subcaracterísticas conforme mostrado na Tabela 1.

Tabela 1: Características e subcaracterísticas definidas pelo ISO 9126

Características	Subcaracterísticas
Funcionalidade	Adequação
	Acurácia
	Interoperabilidade
	Segurança de acesso
Confiabilidade	Conformidade
	Maturidade
	Tolerância à falhas
Usabilidade	Recuperabilidade
	Inteligibilidade
	Apreensibilidade
Eficiência	Operacionalidade
	Comportamento em relação ao tempo
	Comportamento em relação aos recursos
Manutenibilidade	Analisabilidade
	Modificabilidade
	Estabilidade
	Testabilidade
Portabilidade	Adaptabilidade
	Capacidade para ser instalado
	Capacidade para substituir
	Conformidade

Fonte: ISO/IEC 9126-1: 2000. Software engineering– Software product quality- Part 1: Quality Model.

A seguir as características e subcaracterísticas são detalhadas. As definições das subcaracterísticas foram extraídas diretamente da norma.

A funcionalidade compreende o conjunto de funções que satisfazem as necessidades explícitas e implícitas para a finalidade a que se destina o produto e suas subcaracterísticas são:

- Adequação: “Atributos do software que evidenciam a presença de um conjunto de funções e sua apropriação para as tarefas específicas”;
- Acurácia: “Atributos do software que evidenciam a geração dos resultados ou efeitos corretos ou conforme acordados”;
- Interoperabilidade: “Atributos do software que evidenciam sua capacidade de interagir com sistemas específicos”;
- Conformidade: “Atributos do software que fazem com que ele esteja de

acordo com as normas, convenções ou regulamentações previstas em leis e descrições similares, relacionadas à aplicação";

- Segurança de acesso: "Atributos do software que evidenciam sua capacidade de evitar o acesso não autorizado, acidental ou deliberado a programas e dados".

A confiabilidade é o conjunto de atributos que evidenciam que o desempenho do software se mantém ao longo do tempo e em condições estabelecidas. Suas subcaracterísticas são:

- Maturidade: "Atributos do software que evidenciam a frequência de falhas ou defeitos no software";
- Tolerância à falhas: "Atributos do software que evidenciam sua capacidade em manter um nível de desempenho especificado nos casos de falhas no software ou de violação nas interfaces especificadas";
- Recuperabilidade: "Atributos do software que evidenciam sua capacidade de restabelecer seu nível de desempenho e recuperar os dados diretamente afetados, em caso de falha, e no tempo e esforço necessários para tal".

A usabilidade compreende os atributos que evidenciam o esforço para utilização do software pelo usuário, ou seja, quão fácil é utilizar este software. As subcaracterísticas são:

- Inteligibilidade: "Atributos do software que evidenciam o esforço do usuário para reconhecer o conceito lógico e sua aplicabilidade";
- Apreensibilidade: "Atributos do software que evidenciam o esforço do usuário para aprender sua aplicação (por exemplo: controle de operações, entradas, saídas)";
- Operacionalidade: "Atributos do software que evidenciam o esforço do usuário para sua operação e controle da operação".

A característica da eficiência é constituída por um conjunto de atributos que verifica o relacionamento entre o nível de desempenho de software e a quantidade de recursos usados, mediante condições estabelecidas. As subcaracterísticas são:

- Comportamento em relação ao tempo: "Atributos do software que evidenciam seu tempo de resposta, tempo de processamento e velocidade na execução de suas funções";
- Comportamento em relação aos recursos: "Atributos do software que

evidenciam a quantidade de recursos usados e a duração de seu uso na execução de suas funções".

A manutenibilidade compreende os atributos que evidenciam o esforço necessário para fazer modificações especificadas no software. Suas subcaracterísticas são:

- Analisabilidade: "Atributos do software que evidenciam o esforço necessário para diagnosticar deficiências ou causas de falhas, ou para identificar partes a serem modificadas";
- Modificabilidade: "Atributos do software que evidenciam o esforço necessário para modificá-lo, remover seus defeitos ou adaptá-lo a mudanças ambientais";
- Estabilidade: "Atributos do software que evidenciam o risco de efeitos inesperados, ocasionados por modificações";
- Testabilidade: "Atributos do software que evidenciam o esforço necessário para validar o software modificado".

A portabilidade é a capacidade do software de ser transferido de um ambiente para o outro. Suas subcaracterísticas são:

- Adaptabilidade: "Atributos do software que evidenciam sua capacidade de ser adaptado a ambientes diferentes especificados, sem a necessidade de aplicação de outras ações ou meios além daqueles fornecidos para essa finalidade pelo software considerado";
- Capacidade para ser instalado: "Atributos do software que o tornam consoante com padrões ou convenções relacionadas à portabilidade";
- Capacidade para substituir: "Atributos do software que evidenciam sua capacidade e esforço necessário para substituir um outro software, no ambiente estabelecido para este outro software".

Assim como as características principais, as subcaracterísticas não podem ser mensuradas diretamente, portanto elas são divididas em atributos mensuráveis.

Para a aplicação da norma devem ser levadas em consideração as características e subcaracterísticas que tenham maior importância para o cliente e dar pesos diferentes às mesmas.

Para avaliar a qualidade de software é necessário medi-la. Para determinar o valor de uma característica ou subcaracterística é necessária a utilização de

métricas para que seja possível avaliar a qualidade de um software. As métricas utilizadas na norma ISO/IEC 9126 são classificadas em três tipos: externas, internas e de qualidade de uso. Para os fins deste trabalho só serão definidas e exemplificadas as métricas externas.

2.2.1 Métricas externas

As métricas externas são usadas para avaliar o produto de software através de medições baseadas nas necessidades do usuário. Essas métricas usam medidas de um produto de software derivadas de medidas do comportamento do sistema do qual faz parte. A tabela 2 mostra exemplos de métricas externas descritas na norma ISO/IEC 9126.

Tabela 2: Exemplo de métricas do ISO 9126

Nome da métrica	Propósito da métrica	Medida e fórmula	Interpretação	Tipo de escala	Tipo de medida
Densidade de falhas	Quantos problemas foram detectados?	X = NFAI / SIZE NFAI = Número de falhas encontradas SIZE = Tamanho do produto	0 <= X Nos testes finais, quanto mais próximo a zero melhor.	Taxa	NFAI = Contagem SIZE = Tamanho X = Contagem / Tamanho
Cobertura de implementação funcional	Quantas funções estão de acordo com a especificação?	X = A / B A = Número de funções corretamente implementadas, confirmadas através da execução de testes B = Número de funções descritas na especificação	0 <= X <= 1 Quanto mais próximo de 1 melhor.	Absoluta	A = Quantidade B = Quantidade X = Quantidade / Quantidade
Tempo gasto	O mantenedor	Tempo médio:	0 < Tempo	Taxa	Tm, TS, TI =

Nome da métrica	Propósito da métrica	Medida e fórmula	Interpretação	Tipo de escala	Tipo de medida
para implementar uma mudança pelo mantenedor	consegue mudar o software facilmente para resolver uma falha?	Soma(Tm) / N Tm = TS – TI TS = Tempo que a falha foi removida do software TI = Tempo que a causa da falha foi encontrada N = Número de falhas registradas e removidas do software	médio Quanto mais próximo de zero melhor.		Tempo

Fonte: ISO/IEC 9126-2: 2000. *Software engineering– Software product quality*- Part 2: External Metrics.

3 DEFEITOS DE SOFTWARE

O ser humano está sujeito a cometer enganos que podem gerar erros seja no código, na especificação, na arquitetura ou em qualquer outro artefato de um sistema (ISTQB). Este erro humano, a menos que seja descoberto, se propagará como um defeito até o código executável. Quando o pedaço de código defeituoso é executado o sistema entrará em um estado de erro, que poderá ser visto como uma anomalia ou falha quando ele chegar ao cliente final (Maximilien and Willians, 2003). É importante ressaltar que nem todos os defeitos implicam necessariamente em uma ou mais falhas. Por exemplo, um trecho de código não utilizado pelo sistema pode conter um defeito, mas, a falha originária desse defeito só se manifestará quando o trecho de código for executado.

Além da natureza humana, que o faz passível de erro, existe fatores externos que também influenciam na causa de novos defeitos são eles:

- Pressão no prazo;
- Alta complexidade do código;
- Alta complexidade na infra-estrutura;
- Mudanças de tecnologia.

Além dos fatores citados acima, que podem ser considerados internos se usarmos como ponto de vista a empresa ou desenvolvedor responsáveis pela concepção do sistema, existem ainda fatores ambientais ou externos que também podem influenciar na causa de novos defeitos, como radiação, magnetismo, campos eletrônicos e poluição, que podem causar falhas em sistemas embarcados (*firmware*) ou influenciar a execução de software pela mudança das condições de *hardware* (ISTQB).

As consequências dos defeitos dependem tanto do sistema, seu tamanho, complexidade e aplicação quanto da característica do defeito. Os impactos dos defeitos variam e podem causar de problemas pequenos como o travamento de um computador até eventos catastróficos que causam mortes. Do ponto de vista de negócios os defeitos podem gerar perda de clientes, custos maiores de manutenção ou vendas reduzidas. Devido a estes fatores foram desenvolvidas estratégias para lidar com defeitos, tendo como objetivo causar o menor impacto ao usuário final.

3.1 Estratégias para lidar com defeitos

Existem basicamente três estratégias para lidar com defeitos:

- Prevenção: É atingida através de atividades de especificação, implementação, arquitetura e manutenção apropriadas com o principal objetivo de evitar defeitos. Esta estratégia faz uso de métodos avançados de construção de sistema, métodos formais e a reutilização de blocos de sistema que são confiáveis, e o suporte ativo do conhecimento do domínio do sistema;
- Remoção: Esta estratégia faz uso de técnicas como revisão, análise e testes para verificar e validar uma implementação e, assim remover as falhas que foram expostas devido a erros durante a especificação, arquitetura e na própria implementação;
- Tolerância: Faz uso de técnicas que permitam o sistema continuar em operação em um nível de performance e segurança aceitável depois que uma falha foi descoberta. Esta estratégia pode ser usada como uma camada adicional de proteção já que não existem técnicas que garantam um sistema complexo livre de defeitos.

A estratégia padrão para lidar com defeitos é a prevenção, pois, através dela é possível remover a maior parte das falhas dependentes de um estado específico do sistema. Apesar de ser a estratégia padrão a prevenção não é necessariamente econômica ou factível (Maximilien and Williams, 2003). Sendo assim, a melhor coisa a se fazer depois da prevenção é tentar remover os defeitos assim que ocorrerem evitando que se propaguem até o usuário final. Uma implementação confiável de ciclos de remoção de falhas, especialmente aqueles que não resultam de uma falha que precisa ser corrigida, estão geralmente associados a níveis altos do CMM (Capability Maturity Model) (Maximilien and Williams, 2003). Além disso, quanto mais cedo um defeito for corrigido mais barato ele custará.

> “Se um problema for pego na fase de levantamento de requisitos, este tem um custo de aproximadamente 139 dólares para ser consertado. Quando o desenvolvimento é iniciado, o custo cresce para aproximadamente 1000 dólares por falha. Se a falha não for encontrada até o projeto ser completado, o custo cresce de maneira significante. Por exemplo, muitas empresas possuem times para testes os quais tem como trabalho explorar um produto

> exaustivamente depois da fase de desenvolvimento ser completada. Para essas pessoas encontrarem falhas, e para estas falhas serem corrigidas, o custo médio passa os 7000 dólares. Se as falhas não forem encontradas e consertadas até o produto atingir o usuário final, o custo sobe para mais de 14 mil dólares por falha - mais de 100 vezes mais dinheiro por falha se usarmos para comparação o custo de uma falha encontrada na fase inicial do desenvolvimento.” (Viega e McManus, 2000)

Uma das maneiras de assegurar que o sistema atende as expectativas do usuário final apresentando o menor número de defeitos possível é executar testes para verificar e validar seu funcionamento.

4 TESTES DE SOFTWARE

Teste de software é uma investigação conduzida para prover às partes interessadas informações sobre a qualidade de um produto ou serviço que está em teste ou ainda “Teste de software é qualquer atividade que tem como objetivo avaliar um atributo ou capacidade de um programa ou um sistema e determinar que ele obtém os resultados esperados” (Hetzel, 1988)

O teste, ao contrário do que se imagina não é apenas, o processo de executar um sistema com a intenção de encontrar falhas (Myers, 1979). Testes podem possuir objetivos diferentes como, por exemplo, prover informações para tomada de decisão. O teste é parte integrante do ciclo de desenvolvimento de sistemas e pode ser aplicado em qualquer fase do ciclo de desenvolvimento (Pan, 1999). Além do objetivo citado como exemplo os principais objetivos do teste de software são:

- Aumento de qualidade, pois, as falhas de software podem causar grandes perdas já que sistemas são usados nas mais diversas áreas. Qualidade significa que o sistema irá se comportar conforme o requisito estabelecido. Um sistema que roda corretamente, o requisito mínimo de qualidade, significa que este irá se comportar conforme o esperado nas circunstâncias previstas (Pan, 1999);
- Uso na verificação e validação do sistema gerando métricas que possam ser usadas para comprovar se o sistema funciona em determinadas situações ou não;
- Comprovar se o sistema é confiável, ou seja, são feitos testes de acordo com um perfil operacional e são medidas as quantidades de falhas de um sistema em um determinado período de tempo dado um ambiente específico, por exemplo, testes que envolvam muitos acessos simultâneos a uma mesma funcionalidade por um dado período de tempo.

É importante ressaltar que é impossível testar um sistema inteiro e garantir que ele não possui nenhum defeito. Sistemas são geralmente muito complexos e, sua complexidade não pode ser controlada já que os clientes sempre querem que uma nova funcionalidade seja implementada deixando-o ainda mais complexo.

Existem diversas técnicas e métodos de teste para as mais variadas etapas

do ciclo de desenvolvimento de sistemas. Para guiar os testes são necessários oráculos, que nada mais são que métodos para checar se o sistema que está sendo testado se comportou corretamente durante uma execução em particular (Baresi and Young, 2011). Os testes podem ser classificados por propósito e divididos em: testes de correção, testes de performance, testes de disponibilidade e testes de segurança. Podem também ser classificados por escopo, conhecidos como níveis de teste, e divididos em: testes de unidade, testes de componente, testes de integração e testes de sistema (Pan, 1999). Os métodos mais populares de teste derivam dos testes de correção já que não é sempre que existem métricas ou especificações para testes de segurança, disponibilidade e performance.

4.1 Métodos de teste

Dentro dos testes de correção se encontram os métodos de teste mais usados e difundidos. As estratégias de caixa, como são mais conhecidas, compreendem os requisitos mínimos para que o sistema funcione, que é o propósito essencial para os testes (Pan, 1999). A pessoa responsável pelo teste não precisa necessariamente conhecer os detalhes internos do sistema em teste, portanto podem ser adotados tanto os pontos de vista de caixa branca quanto o ponto de vista de caixa preta para executar um teste (Pan, 1999).

4.1.1 Testes de caixa branca

Também chamado de teste de caixa de vidro, teste guiado por lógica (Myers, 1979) ou teste baseado em design (Hetzel, 1988) este método consiste em construir casos de teste de acordo com detalhes de implementação do sistema, como linguagem de programação, lógica e estilos (Pan, 1999). Este método não é capaz de detectar partes do sistema que não foram implementadas ainda já que baseia-se exclusivamente no código existente. Através desse método é possível testar exaustivamente o sistema, pois é possível criar casos de teste para cada tomada de decisão do sistema, por exemplo, estruturas de controle aplicadas a uma dada condição.

4.1.2 Testes de caixa preta

Também conhecida como testes orientados a dados (Myers, 1979) ou testes baseados em requisitos (Hetzel, 1988) este método se preocupa apenas com o aspecto funcional do sistema sem se preocupar com a estrutura interna do mesmo. Neste método a funcionalidade é determinada observando as saídas de acordo com as entradas providas, baseando-se nos resultados esperados na especificação (Pan, 1999).

Ao contrário dos testes de caixa branca, este método não requer nenhum conhecimento de linguagem de programação. Para se obter maior cobertura e determinar quão robusto é o sistema em teste é necessário criar um grande conjunto de entradas com a maior variedade possível. O grande problema deste método é que nunca é possível saber quando a especificação, critério usado para criar um caso de teste, está correto, em função das restrições da linguagem utilizada para escrevê-la (geralmente linguagem natural) a ambiguidade é frequentemente inevitável além de ser responsável por aproximadamente 30% de todas as falhas do sistema (Pan, 1999).

4.2 Níveis de teste

Testes são feitos geralmente em níveis diferentes ao longo do processo de desenvolvimento e manutenção do sistema. Os alvos dos testes podem variar: um módulo, um grupo do mesmo tipo de módulos (relacionados por propósito, uso, comportamento ou estrutura) ou o sistema inteiro. Neste processo três grandes estágios podem ser conceitualmente distinguidos sem que nenhum tenha maior importância do que o outro são eles: Testes de unidade, testes de integração e testes de sistema (SWEBOK).

4.2.1 Testes de unidade

Em 1986 a IEEE definiu testes de unidade como:

> “Um conjunto de uma ou mais unidades de programas com sua lógica de controle de dados (por exemplo, tabelas), procedimentos de uso e operações que satisfaz as seguintes condições: (1) Todas as unidades são de um mesmo programa (2) Pelo menos uma das unidades novas ou modificadas ainda não foi testada por testes de unidade (3) O conjunto de unidades com sua lógica de dados e procedimentos é o único objetivo do processo de teste”.

Apesar do relatório técnico afirmar que testes de unidade podem ser tanto um módulo quanto um sistema inteiro, é mais fácil observar um dado comportamento em unidades menores e por este motivo as definições sobre testes de unidade pode variar na literatura.

Segundo Hunt and Thomas (2003) um teste de unidade é um pedaço de código escrito por um desenvolvedor e que exercita uma funcionalidade pequena e específica no código sendo testado. Mas, mais importante que adotar a uma definição específica para testes de unidade é entender quais são seus objetivos (Ammann, 2008):

- Identificar falhas dentro de escopos pequenos tal que seja fácil para o desenvolvedor encontrá-las;
- Remover o máximo de falhas possíveis antes do teste de integração;
- Dar um retorno ao programador da relevância da unidade programada em relação às especificações.

Testes de unidade podem incluir também características específicas não funcionais tais como comportamento dos recursos, como falta de memória, e testes de robustez, além de testes estruturais como cobertura de código.

Tipicamente, testes de unidade ocorrem com acesso ao código que está sendo testado e com a ajuda de ferramentas de testes e de depuração. Na prática, envolve o programador do código.

4.2.2 Testes de integração

Testes de integração são utilizados para verificar a interação entre as unidades de um sistema. A integração pode ser feita em diferentes estágios do sistema para validar se o que se está sendo desenvolvido funcionará corretamente

quando todas as “peças” do sistema estiverem juntas. Neste nível também são validados acessos a outras aplicações, como um banco de dados, as quais a empresa ou desenvolvedor não tem acesso ao código.

4.2.3 Testes de sistema

Testes de sistema têm como objetivo verificar e validar um sistema inteiro. Neste nível de teste a grande maioria da falhas funcionais já foram identificadas nos níveis anteriores (teste de unidade e de integração). Este nível de teste é considerado apropriado para validar requisitos não funcionais como segurança, velocidade e disponibilidade. Toda e qualquer variável de ambiente como requisitos físicos, por exemplo, memória, sistema operacional, etc. são validados neste nível de teste.

5 DESENVOLVIMENTO ITERATIVO E INCREMENTAL

Com o crescimento das metodologias ágeis de desenvolvimento algumas práticas antigas ganharam maior visibilidade. Tanto o desenvolvimento incremental como o iterativo são mais antigos que as metodologias ágeis. O primeiro projeto de sistemas que utilizou o IID, acrônimo para Iterative and Incremental Development, foi o Mercury da NASA em 1960 (Basili and Larman, 2003).

As duas práticas estão presentes no ciclo do TDD e por isso entender seus propósitos, diferenças e como usá-las em conjunto é importante.

5.1 Desenvolvimento iterativo

No desenvolvimento iterativo existem intervalos de tempo para melhorar o que já está desenvolvido no sistema em questão (Cockburn, 2001). A diferença entre essa prática e as demais é que ao invés de entregar o que foi desenvolvido no final de um ciclo de desenvolvimento são examinados alguns pontos: O que foi desenvolvido está correto? Os clientes gostam da maneira que o que foi desenvolvido funciona? O que foi desenvolvido é rápido o suficiente?

Existem duas estratégias para a aplicar a prática, uma considerada otimista e outra pessimista:

- A estratégia otimista baseia-se em desenvolver o sistema da melhor maneira possível tendo em mente que se o mesmo foi desenvolvido suficientemente bem, as alterações serão relativamente pequenas e poderão ser incorporadas rapidamente;
- Na estratégia pessimista desenvolve-se o mínimo possível antes do sistema entrar na fase de validação, pressupondo-se que menos trabalho será perdido quando novas informações forem obtidas.

Figura 1: Exemplo de desenvolvimento iterativo.

Fonte: Adaptado de <http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=13178>

5.2 Desenvolvimento incremental

No desenvolvimento incremental, o trabalho é quebrado em pedaços menores e desenvolvidos separadamente e integrados quando completados (Cockburn, 2001). A idéia do desenvolvimento incremental é possibilitar a entrega de sistemas com um requisito funcional antes do prazo esperado e assim obter resultados e opiniões sobre o produto antes de todas as demais funcionalidades serem agregadas e, possíveis erros serem sanados antes do produto final ser construído. Um erro comum nesta prática é não compreender a tempo o que não foi entendido nas definições iniciais do projeto e o que precisa ser melhorado no design do sistema. Este erro leva a um erro maior e mais antigo, o de entregar o que o cliente não quer.

Figura 2: Exemplo de desenvolvimento incremental.

Fonte: Adaptado de <http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=13178>

5.3 Combinando as duas práticas

O desenvolvimento incremental dá oportunidades de melhorar tanto o

processo de desenvolvimento quanto fazer ajustes aos requisitos que mudam em função do tempo. O desenvolvimento incremental dá oportunidades de melhorar a qualidade do produto final (Cockburn, 2001). A combinação das duas práticas, portanto, evita que falhas e defeitos sejam encontrados apenas antes do produto entrar em produção e, torna mais simples a tarefa de os encontrar e corrigir. Se a cada ciclo de incremento no sistema for feito um ciclo de iteração é possível corrigir problemas de performance, ajustar o sistema para que o mesmo funciona da maneira que o cliente deseja e certificar-se que o sistema comporta-se da maneira como foi planejado.

6 TEST DRIVEN DEVELOPMENT

TDD ou Test Driven Development é uma técnica de desenvolvimento de sistemas que tem sido usada esporadicamente por décadas (D. Gelperin and W. Hetzel, 1987); a referência mais antiga do seu uso data de 1960 no Projeto Mercury da NASA (C. Larman and V. Basili, 2003). A técnica é uma das principais práticas do Extreme Programming (XP), porém, nem todos os desenvolvedores concordarem com todas as práticas do XP, fazendo com que o desenvolvimento guiado por testes começasse a ganhar vida própria (Maximilien and Williams, 2003). Apesar do fato de possuir a palavra “teste” no nome e envolver a criação e a execução de casos de teste, não é possível dizer que a técnica é apenas sobre testes. O ciclo do TDD afeta todos os envolvidos no desenvolvimento de software desde o desenvolvedor até o cliente final.

A técnica utiliza a estratégia “teste primeiro” na qual os casos de testes são escritos um a um antes do código que os implementa, e em seguida o código para que o teste tenha sucesso é gerado. O desenvolvimento de software se torna uma série de pequenas iterações nas quais cada caso de teste guia a criação de software e por último no design do programa (Schroeder and Rothe, 2005).

Estudos realizados na IBM mostram que após a adoção do TDD o número de defeitos diminuiu 40% em relação a um projeto que não utiliza a técnica. O mesmo estudo ainda demonstra que a quantidade de tempo para o desenvolvimento de um novo requisito com o emprego do TDD subiu de 15 a 20% (Nagappan, 2008). Por esses e outros motivos é necessário que a técnica seja bem definida e explicada para que assim seja possível entender quais são seus reais riscos e benefícios de implantação.

6.1 Conceito

TDD é uma técnica de desenvolvimento de software que propõe que os testes, para uma dada funcionalidade, sejam escritos antes da implementação da mesma e, que novos trechos de código só sejam escritos com objetivo de fazer com que algum teste resulte em sucesso (Koskela, 2007). Apesar de estar relacionado

com testes o TDD não necessariamente implica em validação. Para alguns autores o objetivo da técnica é tornar a especificação e a estrutura da aplicação mais simples e coesa.

"TDD não é sobre testes, é sobre como usar testes para criar software de maneira simples e incremental. Além de melhorar a qualidade e o projeto de software, ele também simplifica o processo de desenvolvimento" (Gold et all. 2004)

> "No ciclo de desenvolvimento do TDD escrevemos os testes antes de escrever o código. Ao invés de verificar todo seu trabalho depois de pronto, TDD faz com o que testar seja uma atividade de arquitetura. Nós usamos testes para deixar nossas idéias mais claras sobre o que queremos que o código faça." (Freeman et all., 2004)

O grande diferencial do TDD é seu ciclo curto. Esta técnica vai ao caminho contrário da maneira que a grande maioria dos desenvolvedores está acostumada a trabalhar. Num ciclo de desenvolvimento normal o primeiro passo é projetar, logo depois implementar o projeto e por último testar a implementação do projeto de alguma forma (Koskela, 2007). No modelo tradicional de desenvolvimento de software o desenvolvedor só descobrirá alguma falha na última etapa do ciclo, ou seja, depois de já ter implementado tudo que o projeto contemplava, a quantidade de código já é grande tornando a tarefa de corrigir a falha mais difícil e ainda não garante que o desenvolvedor não introduzirá novas falhas na tentativa de corrigir a primeira falha encontrada.

"A granularidade do ciclo teste-então-codifique fornece um retorno contínuo ao desenvolvedor. Com TDD, falhas e/ou outros defeitos são identificados muito cedo e rapidamente assim que código novo é adicionado ao sistema, e a fonte do problema é mais fácil de ser determinada" (Maximilien and Wiliams, 2003)

O ciclo do TDD repete pequenos ciclos de IID (Desenvolvimento iterativo e incremental). O desenvolvimento incremental constrói um sistema requisito por requisito, ao invés de construir todas as camadas e componentes e os integrando ao final do desenvolvimento (Freeman et all, 2004). Dessa forma é possível manter o sistema sempre integrado sem ter que esperar até seu final para que as integrações sejam postas à prova. O desenvolvimento iterativo faz com que a implementação do requisito seja melhorada de maneira progressiva até que a mesma esteja boa o bastante (Freeman et all, 2004). Por ter essas duas características a metodologia

possibilita que o sistema cresça de maneira segura e que seja possível fazer ajustes necessários sem onerar o ciclo de vida do sistema em desenvolvimento. Para que isso seja possível devem-se testar constantemente as funcionalidades existentes, evitando assim que novas funcionalidades implementadas façam com que as antigas parem de funcionar, além de manter o código o mais simples possível tornando-o manutenível (Freeman et all, 2004).

Astels (2003) resume o que foi discorrido sobre o conceito da técnica em tópicos:

- Mantém-se um conjunto exaustivo de testes que além de guiar o desenvolvimento asseguram as funcionalidades pré-existentes;
- Nenhum código entra em produção sem um teste associado a ele;
- Testes são escritos antes da implementação da funcionalidade;
- Testes determinam a quantidade e como o código da implementação será escrito.

6.2 O ciclo do TDD

O ciclo de desenvolvimento do TDD é constituído de iterações. Cada iteração corresponde à implementação de uma nova funcionalidade ou correção de alguma falha e é composta dos seguintes passos (Beck, 2002):

- Escrever rapidamente o teste para a funcionalidade;
- Rode todos os testes e veja o novo teste falhando;
- Faça uma pequena alteração;
- Execute todos os testes e veja todos terem sucesso;
- Mude o código para remover duplicação.

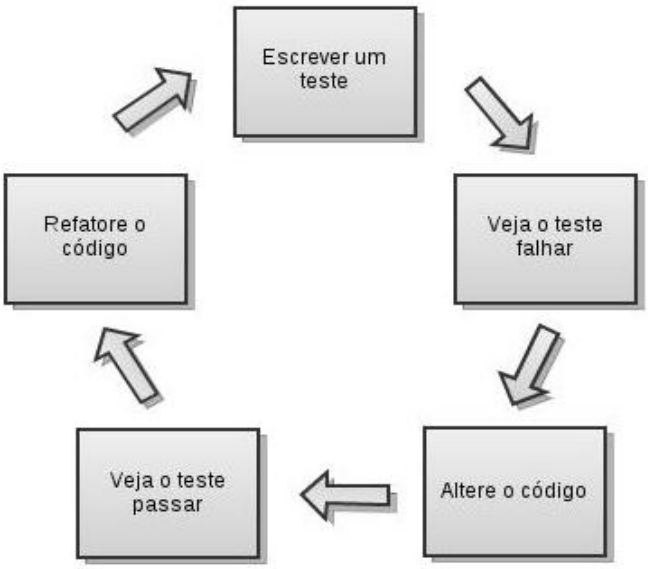

Figura 3: Ciclo do TDD.
Fonte: Elaborado pelo autor.

Primeiro escreve-se um teste, depois escreve-se código para que o teste funcione e na última etapa remove-se a duplicação existente tanto no código de teste quando no código que fez o teste funcionar.

Esse pequeno ciclo permite ao desenvolvedor construir um software de maneira incremental, o que reduz riscos porque a quantidade de trabalho não terminado continua pequena e garante que no prazo final de entrega do software, o desenvolvedor tenha algo funcional mesmo que ele não contenha todas as funcionalidades exigidas pelo cliente. (Koskela, 2007)

Então, além de garantir a funcionalidade do software através dos testes, o ciclo do TDD possibilita que o software seja construído em pequenas partes sempre baseadas nas especificações de funcionalidade proposta pelo projeto e, permite que o desenvolvedor nunca se afaste da especificação completa do projeto e consiga controlar, com maior facilidade, estimativas para a conclusão do mesmo.

6.2.1 Escrever o teste

Cada requisito, caso de uso ou estória de usuário é decomposto em um conjunto de comportamentos que são necessários para que o requisito seja atendido

por completo (Gold et all, 2004). Para cada comportamento do sistema deve-se escrever um teste de unidade, dessa forma é possível ter um critério de aceitação para cada um dos comportamentos que formam o requisito.

Uma característica importante é que o teste deve conseguir ser executado em isolamento, ou seja, que o sucesso ou falha dele deve ser irrelevante para outros testes (Beck, 2002). Outra característica importante é a que o teste deve ser executado de forma rápida, pois se a cada vez que for necessário rodar os testes muito tempo for perdido o desenvolvedor deixará de executá-los com frequência e depois de algum tempo os abandonará. Uma ótima maneira de tornar os testes rápidos é automatizando os mesmos, para tal podemos fazer uso das variações da família xUnit, que é composta por frameworks que abrem um leque de opções ao desenvolvedor para escrever testes e verificar seus resultados com informações de quais testes passaram, quais falharam e até mesmo a linha da falha.

6.2.2 Ver o novo teste falhando

Uma das únicas formas de saber se o teste é válido é vendo o mesmo falhar. No processo de escrita é possível esquecer de algum detalhe que o torne inválido sintaticamente como utilizar uma nomenclatura inválida para alguma variável. Além disso, a falha no teste mostra que a asserção feita realmente só irá ser válida quando implementarmos o código que realmente a satisfaça e que o teste não depende de nenhum outro resultado para que seja válido. A falha, neste caso, representa um tipo de progresso, pois agora há uma visão concreta do que deve ser feito para que o objetivo seja alcançado e, não mais uma vaga idéia do que possa estar errado.

6.2.3 Fazer uma pequena alteração

O código gerado nessa etapa deve ser o necessário para fazer o teste ter sucesso. Nenhuma outra mudança em qualquer parte do sistema deve ser feita a não ser que seja para cumprir o objetivo acima. Nesse estágio o desenvolvedor não deve se preocupar com modularidade, boas práticas de desenvolvimento ou qualquer outro fator de qualidade.

"Você provavelmente não vai gostar da solução, mas o objetivo agora não é obter a resposta perfeita, mas, fazer o teste passar. Nós vamos fazer nosso sacrifício no altar da verdade e da beleza mais tarde" (Beck, 2002)

6.2.4 Executar todos os testes e ver todos terem sucesso

Depois de ter feito a alteração mais simples possível para que o teste obtivesse sucesso, no passo anterior, todos os outros testes devem ser executados. Essa é a garantia que nenhuma alteração feita pelo desenvolvedor causou a falha de outros testes, obedecendo assim à característica de isolamento do primeiro passo do ciclo.

6.2.5 Mudar o código para remover duplicação

É neste passo que o desenvolvedor olhará para o código produzido até agora e, aplicará técnicas para a remoção da duplicação inserida pelo teste e é também uma das partes mais importantes do ciclo todo.

> "O jeito mais comum que já vi para estragar o TDD é negligenciando o último passo. Refatorar o código para mantê-lo claro é a mais importante parte do processo, caso contrário você acaba com uma agregação bagunçada de fragmentos de código (Pelo menos eles terão testes, então é um resultado menos doloroso que a maioria das falhas de design)" (Fowler, 2005)

A esta etapa também é dado o nome de refatoração e é uma das bases do TDD.

> "Refatoração é o processo de mudar código com o objetivo único de melhorar sua estrutura interna, sem mudar seu funcionamento. Refatoração é basicamente melhoria de mau código. A maioria dos desenvolvedores refatora código diariamente usando o bom senso, contudo um catálogo de métodos de refatoração foi definido e explicado em Refactoring: Improving the Design of Existing Code, de Martin Fowler et al. (Addison-Wesley, 1999). Como refatoração é uma parte importante de TDD é preciso desenvolver um entendimento desses métodos para que se possa rapidamente identificar padrões de mau código e os refatorar" (Gold et al., 2004)

6.3 Vantagens

A prática de TDD traz com ela muitos benefícios, além dos já citados, como desenvolvimento incremental e iterativo, destaca-se entre eles: processo de

desenvolvimento simplificado, constantes testes de regressão, melhora da comunicação, melhor entendimento dos requisitos de software, centralização do conhecimento, melhora no design do sistema e aumento da confiança dos desenvolvedores na aplicação.

O processo de desenvolvimento torna-se mais simples, pois o desenvolvedor não precisa em primeiro lugar se preocupar com sua produtividade porque eles são mais focados. A atenção é toda voltada para uma pequena parte do código, aquela que você está testando. Então, o desenvolvedor que utiliza a técnica só precisa completar um ciclo de TDD e seguir para a próxima tarefa. Desta forma o desenvolvedor deixa de se preocupar com as várias decisões que devem ser tomadas e só vai tomando algumas delas conforme ele vai desenvolvendo o que é mais fácil (Gold et all., 2004).

Testes de regressão são importantes para evitar que mudanças em um módulo tenham consequências desconhecidas em todo projeto (Gold et all., 2004). Com TDD todos os testes devem ser rodados a cada mudança feita no código evitando assim que “surpresas” venham à tona enquanto trabalhamos. Isso quer dizer que qualquer mudança que cause algum efeito indesejado pode ser identificada e corrigida rapidamente.

A melhora da comunicação é possível desde que possamos usar os testes para explicar como um pedaço do sistema irá se comportar em um dado cenário. Ao invés de usarmos documentos ou qualquer outro artefato de software que pode não acompanhar a evolução do mesmo, com TDD passamos a utilizar algo que muda juntamente com o código.

A melhor compreensão dos requisitos do sistema é obtida, pois, à medida que o desenvolvedor passa a escrever os testes ele define quais são os critérios de sucesso e de falha que um dado comportamento deve ter num dado contexto. Quanto mais testes são adicionados por causa de novos requisitos ou falhas, o conjunto de testes se torna a representação dos comportamentos necessários daquele sistema. Por mais que um módulo esteja bem documentado e seu código claro, algumas vezes é difícil entender o porquê de algumas decisões tomadas na hora do desenvolvimento. E, como módulos são geralmente construídos por um único indivíduo o conhecimento dos requisitos e de como o tal módulo funciona fica concentrado em seu criador. Com TDD, os testes constituem um repositório que provê uma lista de requisitos para o módulo. O código implementado provê a

solução para esta lista (Gold et all., 2004);

Quando o código é melhorado ou mantido, rodar os testes de maneira automatizada pode ser usado para identificação de novos defeitos, servindo como testes de regressão para o sistema (Maximilien and Williams, 2003);

TDD evita que a manutenção de software por meio de depuração seja utilizada. As correções feitas utilizando este método estão 40 vezes mais propensas a erros que o desenvolvimento de uma nova funcionalidade (Humphrey, 1989). Utilizando TDD, sabem-se exatamente quais linhas foram adicionadas e que fizeram o teste falhar e é possível chegar à fonte do problema de imediato, evitando assim longas sessões de depuração (Koskela, 2007);

O desenvolvedor ganha confiança, pois, sabe que ao fazer uma mudança ele precisa apenas executar os testes para saber se o que ele fez compromete algo e pode rapidamente mudar até que os testes estejam todos funcionando novamente.

Um dos benefícios menos óbvios do TDD é como a técnica pode ajudar a melhorar o design do sistema como:

- Melhora no encapsulamento e na modularidade do sistema;
- Relacionamentos entre classes mais simples;
- Complexidade do design reduzida.

6.4 Desvantagens

Se comparado a formas consagradas de desenvolvimento de software, TDD gera uma base de testes maior. Em um estudo realizado para aferir a aceitação do TDD tanto no meio acadêmico quanto na indústria, foi observado que os grupos que praticaram TDD para cada linha de código gerada, duas linhas de código de teste eram geradas (Dubinsky and Hazzan, 2007). Grandes bases de testes podem trazer problemas desde que as mesmas não evoluam junto com o código que as faz passar. Conforme o sistema evolui é necessário revisitar os testes como é feito com as classes que eles verificam e foram mudadas para adicionar novas funcionalidades ou foram simplificadas através de refatoração (Meszaros, 2007). Não manter a base de testes implica nos seguintes problemas:

- Testes que não fazem mais sentido ou que foram mal escritos levam mais

tempo para serem compreendidos. Quanto mais tempo um teste demorar a ser compreendido menor é a produtividade do desenvolvedor, pois, além do tempo necessário para entender o que o teste deveria fazer o desenvolvedor pode levar muito tempo para ajustar o teste e mante-lo passando corretamente;

- Grandes de base de testes mal escritas podem levar muito tempo para executar o que leva o desenvolvedor a abandonar à prática;
- Quando mudanças simples fazem vários testes falharem, além da desmotivação, o desenvolvedor se torna mais resistente a escrever novos testes (Wasmus and Gross, 2007).

Outros estudos empíricos mostraram um aumento na quantidade de falhas em testes de aceitação. O ganho de confiança do desenvolvedor ao praticar TDD pode ser uma faca de dois gumes. Se por um lado o desenvolvedor se sente mais confortável ao alterar código que não é de sua autoria ou mesmo partes críticas de um sistema, o excesso de confiança o desestimula a realizar testes que não são os de unidade como testes de integração ou de sistema (Siniaalto and Abrahamsson, 2007).

Apesar da literatura indicar que o TDD pode ser aplicado em qualquer parte de um projeto nem sempre o emprego da técnica é uma tarefa fácil. Um exemplo clássico são os testes que envolvem a criação de interfaces para usuários. Algumas vezes a criação de vários testes simplesmente consome muito mais tempo se compararmos o processo à verificação manual (Wasmus and Gross, 2007).

Da mesma forma que nas metodologias convencionais pode haver má interpretação de requisitos ou negligência da documentação gerada pela expectativa do cliente. Como os testes são o reflexo dessa interpretação e ainda que estes estejam todos apontando o bom funcionamento do sistema, nem sempre o produto final estará necessariamente de acordo com os requisitos levantados.

7 APLICAÇÕES DE DESENVOLVIMENTO GUIADO POR TESTES

Por se tratar de uma prática crescente nos dias atuais, mas, relativamente nova em comparação ao processo de desenvolvimento de sistemas, casos de estudo que tentam confirmar as vantagens da adoção do TDD ainda são escassos. Os casos de estudo na indústria mais conhecidos são os executados por Maximilien e Williams na IBM, Bhat e Nagappan na Microsoft. Os mesmos serão apresentados a seguir.

7.1 TDD na IBM

O grupo da IBM desenvolve drivers de dispositivos por mais de uma década. Eles possuem um sistema legado que teve sete versões desde de 1998 até 2002. Em 2002 este mesmo grupo desenvolveu drivers de dispositivos numa nova plataforma. A comparação será feita entre a primeira versão do produto produzido na nova plataforma e a sétima versão do produto legado.

7.1.2 Os times

Todos os participantes da aplicação tem pelo menos bacharelado em ciência da computação ou engenharia da computação. O time que atuava no novo produto nunca havia trabalhado com TDD antes e três integrantes não conheciam a linguagem de programação de forma profunda. Além disso, o mesmo time possuía programadores alocados em locais diferentes, cinco dos integrantes estavam nos Estados Unidos e os outros quatro integrantes estavam no México. Ainda, dois dos membros do time eram novatos nos dispositivos para os quais os drivers seriam desenvolvidos, portanto o conhecimento do negócio precisaria ser construído durante a fase de projeto e na fase de desenvolvimento. A tabela 3 contém um comparativo entre os dois times.

Tabela 3: Comparativo entre os dois times de desenvolvimento envolvidos na aplicação

[redacted]	**Produto legado**	**Produto Novo**
Tamanho do time (Desenvolvedores)	5	9
Experiência do time (Linguagem de programação e conhecimento do negócio)	Experiente	Alguns inexperientes
Alocação	Totalmente alocada	Distribuída
Base de código (KLOC) Nova; Base; Total	6.6;49.9.56.5	64.6;9.0;73.6
Linguagem de programação	Java/C++	Java
Testes unitários	Quando necessário	TDD
Liderança técnica	Recurso compartilhado	Recurso dedicado

Fonte: MAXIMILIEN, E. Michael; WILLIAMS, Laurie. Assessing test-driven development at ibm. In ICSE '03: Proceedings of the 25th International Conference on Software Engineering, páginas 564–569, Washington, DC, USA. IEEE Computer Society, 2003.

7.1.3 Práticas de teste unitário

A prática de testes unitários do time do produto legado foi classificada como "Quando necessário", pois, o processo de teste não era formal e nem disciplinado. Apenas classes consideradas importantes, que colaborariam com outras classes ou que seriam utilizadas por outras classes, eram testadas. As ferramentas responsáveis pelos testes eram criadas pelos próprios desenvolvedores através de scripts ou classes independentes que exercitavam pontos específicos do sistema. A grande maioria dos testes gerados pelo time não foi utilizada na fase de verificação subsequente.

O time que desenvolvia o novo produto adotou o TDD como prática de testes unitários. Para cada classe importante, que colaboraria ou que seria utilizada por outra classes, foi exigido que todo método público e seu comportamento fosse testado utilizando a ferramenta JUnit. Com esta abordagem toda a classe implementada possuía uma classe de teste associada a ela e cada método público desta classe possuía um teste associado à classe de teste. Objetivo da aplicação da técnica era atingir uma cobertura de testes automatizados de 80%. Para garantir que todos os testes criados fossem executados por todos os membros do time, os testes eram executados diariamente, tanto nos Estados Unidos quanto no México, e um e-mail era enviado para todos os membros do time informando quais testes tinham sido executados com sucesso e se algum erro havia ocorrido.

7.1.4 Avaliação dos sistemas

Os dois projetos, tanto o legado quanto o novo, foram enviados um time externo que faria os testes funcionais, depois que a grande maioria do código foi implementada. A equipe de qualidade havia escrito testes de caixa preta de acordo com as especificações dos sistemas. Os defeitos encontrados eram comunicados aos desenvolvedores através de um sistema de controle de defeitos e, classificados por dispositivo. Cada defeito possuía um nível de severidade de acordo com a natureza do mesmo e também pelo número testes bloqueados pelo defeito. Para assegurar que os testes que haviam obtido sucesso previamente não estariam falhando depois das iterações para correções de defeito a equipe de qualidade executou todos os testes novamente em forma de teste de regressão.

7.1.5 Resultados

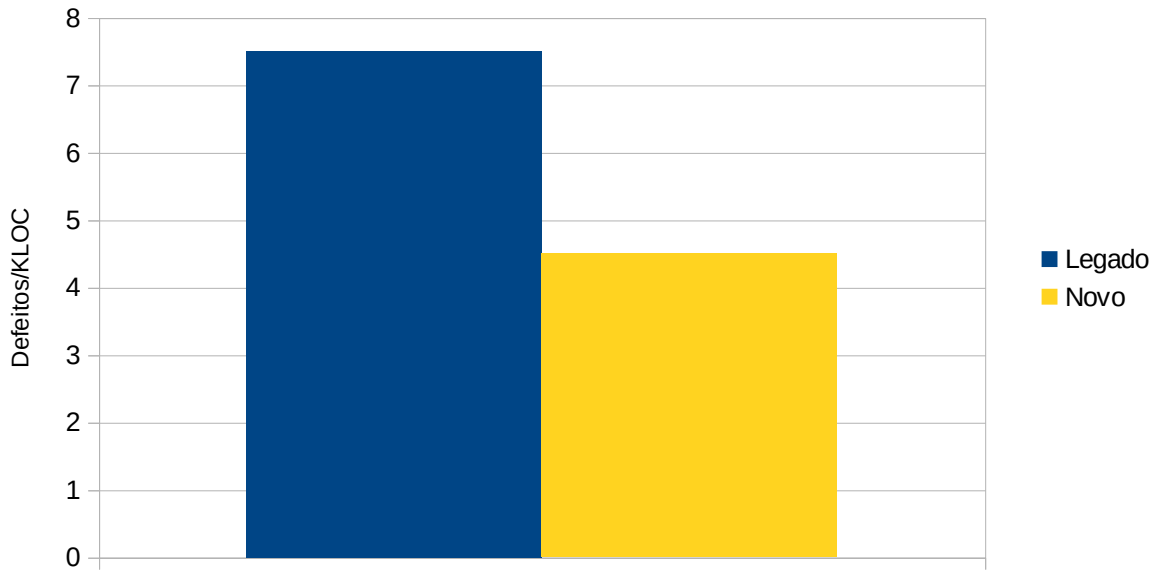

Gráfico 1: Comparação entre o sistema legado e o sistema novo nos testes de regressão feitos pela equipe de qualidade.

O novo projeto teve um desempenho melhor se o compararmos com o legado nos testes de regressão. O gráfico 1 mostra esta comparação de maneira resumida.

A distribuição de severidade entre os defeitos foi basicamente a mesma. A tabela 4 mostra a comparação entre os dois projetos. A escala de severidade varia

de um até quatro onde, um é o mais crítico e 4 representa defeitos não críticos.

Tabela 4: Comparação entre a severidade dos defeitos encontrados pela equipe de qualidade

[redacted]	Produto legado	Produto novo
Severidade 1	3%	2%
Severidade 2	25%	25%
Severidade 3	70%	65%
Severidade 4	2%	8%

Fonte: MAXIMILIEN, E. Michael; WILLIAMS, Laurie. Assessing test-driven development at IBM. In ICSE '03: Proceedings of the 25th International Conference on Software Engineering, páginas 564–569, Washington, DC, USA. IEEE Computer Society, 2003.

O sumário dos resultados é apresentado na tabela 5. O novo produto é menos expansivo do que o legado, ou seja, a abrangência de sistemas operacionais e dispositivos é menor no novo produto se comparado ao legado. Por este motivo muito mais testes foram executados e esforço da equipe de qualidade foi maior para o produto legado. Ainda, este fator faz com que os mesmos testes sejam repetidos para diferentes dispositivos e sistemas operacionais aumentando assim o número de testes por LOC (Line of code) do produto legado e, diminui o número de defeitos por teste também.

Tabela 5: Sumário dos resultados obtidos nos testes da equipe de qualidade

[redacted]	Produto legado	Produto novo
Esforço da equipe de qualidade	E	0.49E
Testes executados	TE	0.48TE
Testes/LOC Total	TLT	0.36TLT
Defeitos/Testes	DT	1.8DT
Defeitos/LOC	DL	0.61DL

Fonte: MAXIMILIEN, E. Michael; WILLIAMS, Laurie. Assessing test-driven development at IBM. In ICSE '03: Proceedings of the 25th International Conference on Software Engineering, páginas 564–569, Washington, DC, USA. IEEE Computer Society, 2003.

7.2 TDD na Microsoft

A primeira aplicação da técnica na Microsoft foi feito pelo time responsável pelas aplicações de rede no setor de desenvolvimento do sistema operacional Windows. Eles deveriam desenvolver uma biblioteca comum para as aplicações de rede, que deveria funcionar como um conjunto de módulos reutilizáveis dentro do

departamento de redes.

7.2.1 O time

O time envolvido no projeto era composto de seis desenvolvedores, todos com um nível alto de experiência tanto na linguagem de programação quanto nas regras de negócio. Além disso, todos os desenvolvedores estavam alocados nos Estados Unidos. A tabela 6 contém as informações do time.

Tabela 6: Sumário do time envolvido na aplicação

Métrica		**Valor**
Tamanho do time (apenas desenvolvedores)		6
Localização do time		Estados Unidos
	> 10 anos	0
Nível de Experiência	**6 – 10 anos**	5
	< 5 anos	1
Conhecimento do negócio (Alto, Médio, Baixo)		Alto
Conhecimento da linguagem de programação (Alto, Médio, Baixo)		Alto
Linguagem de programação		C/C++
Conhecimento do gerente de projeto (Alto, Médio, Baixo)		Alto
Alocação do time		Totalmente alocado

Fonte: BHAT, Thirumalesh; NAGAPPAN, Nachiappan. Evaluating the efficacy of test-driven development: industrial case studies. In ISESE '06: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, páginas 356–363, New York, NY, USA, 2006. ACM.

Embora o projeto não seja grande, apenas 17 classes desenvolvidas, ele não pode ser considerado trivial já que é usado por 50 aplicações da área. O projeto terminou com 79% de cobertura levando em consideração apenas testes unitários e excluindo a cobertura do time de testes. A tabela 7 mostra as métricas do produto final.

Tabela 7: Sumário das métricas do produto

Métrica	**Valor**
Base de código (KLOC)	6
Base de testes (KLOC)	4
Porcentagem de cobertura (testes unitários)	79%

Métrica	Valor
Tempo de desenvolvimento (homem/meses)	24
Código legado	Não

Fonte: BHAT, Thirumalesh; NAGAPPAN, Nachiappan. Evaluating the efficacy of test-driven development: industrial case studies. In ISESE '06: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, páginas 356–363, New York, NY, USA, 2006. ACM.

Um outro projeto, que não fazia uso de TDD, foi utilizado para comparar os resultados obtidos na aplicação. O resultado da comparação é que o projeto que não utilizava TDD apresentou mais defeitos (mais que duas vezes e meia mais defeitos), porém o projeto que utilizava TDD apresentou um aumento de 25 a 35 por cento no tempo de desenvolvimento. A tabela 8 mostra mais detalhes da comparação.

Tabela 8: Comparação final entre o projeto usando TDD e um projeto comparável que não utiliza TDD

Métrica	Projeto com TDD	Projeto sem TDD
Base de código (KLOC)	6	4.5
Tempo de desenvolvimento (homem/meses)	24	12
Tamanho do time	6	2
Quantidade de defeitos	X	2.6X

Fonte: BHAT, Thirumalesh; NAGAPPAN, Nachiappan. Evaluating the efficacy of test-driven development: industrial case studies. In ISESE '06: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, páginas 356–363, New York, NY, USA, 2006. ACM.

O segundo estudo conduzido na Microsoft aconteceu no departamento MSN em uma aplicação que fazia uso de *web services*. O projeto contou com até oito desenvolvedores durante seu desenvolvimento. As pessoas envolvidas neste estudo eram menos experientes que os participantes do primeiro se forem levados em conta os critérios de conhecimento da linguagem de programação, conhecimento do negócio e conhecimento do gerente do projeto. A tabela 9 resume os fatores do contexto do projeto.

Tabela 9: Resumo dos fatores de contexto da segunda aplicação da técnica na Microsoft

Métrica		Valor
Tamanho do time (apenas desenvolvedores)		5 - 8
Localização do time		Estados Unidos
Nível de experiência	**> 10 anos**	1
	6-10 anos	7

< 5 anos	0
Conhecimento do negócio (Alto, Médio, Baixo)	Médio
Conhecimento da linguagem de programação (Alto, Médio, Baixo)	Médio
Linguagem de programação	C++/C#
Conhecimento do gerente de projeto (Alto, Médio, Baixo)	Médio
Alocação do time	Totalmente alocado

Fonte: BHAT, Thirumalesh; NAGAPPAN, Nachiappan. Evaluating the efficacy of test-driven development: industrial case studies. In ISESE '06: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, páginas 356–363, New York, NY, USA, 2006. ACM.

O produto usado para a segunda aplicação da técnica era maior do que o primeiro se usarmos como medida a quantidade de linhas de código, seis mil linhas no primeiro estudo contra 26 mil linhas no segundo, porém, isso não implicou em uma cobertura de testes menor já que no segundo caso a cobertura atingiu 88%. A tabela 10 contém as medidas do produto utilizado.

Tabela 10: Métricas do produto utilizado na Microsoft

Métrica	Valor
Base de código (KLOC)	26
Base de teste (KLOC)	23.2
Porcentagem de cobertura (testes unitários)	88%
Tempo de desenvolvimento (homem/meses)	46
Código Legado	Não

Fonte: BHAT, Thirumalesh; NAGAPPAN, Nachiappan. Evaluating the efficacy of test-driven development: industrial case studies. In ISESE '06: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, páginas 356–363, New York, NY, USA, 2006. ACM.

Assim como na primeira aplicação outro produto do mesmo departamento, que não fazia uso de TDD, foi utilizado para comparação das métricas obtidas. O resultado da comparação foi que o projeto que não utilizava TDD apresentou 4.2 vezes mais defeitos do que o que utilizava e novamente o projeto que utilizava TDD teve um aumento no tempo de desenvolvimento, 15% a mais se comparado ao projeto que não utilizava TDD. A tabela 11 apresenta o comparativo entre os dois projetos.

Tabela 11: Comparativo entre o produto utilizado e um outro produto do mesmo departamento

Métrica	Projeto com TDD	Projeto sem TDD
Base de código (KLOC)	26	149
Tempo de desenvolvimento (homem/meses)	46	144
Tamanho do time	5 - 8	12
Quantidade de defeitos	X	4.2X

Fonte: BHAT, Thirumalesh; NAGAPPAN, Nachiappan. Evaluating the efficacy of test-driven development: industrial case studies. In ISESE '06: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, páginas 356–363, New York, NY, USA, 2006. ACM.

7.3 Avaliação dos resultados obtidos

Por motivos de sigilo o número de defeitos dos casos de estudo tanto da Microsoft quanto na IBM não foram revelados. A tabela 12 resume os resultados obtidos.

Tabela 12: Comparativo entre os casos de estudo IBM e Microsoft

Descrição da métrica	IBM	Microsoft Redes	Microsoft MSN
Densidade de defeitos em um projeto comparável sem o uso de TDD	X	Y	Z
Densidade de defeitos do time com uso de TDD	0.61X	0.38Y	0.24Z
Aumento no tempo de desenvolvimento	15-20%	25-25%	15%

Fonte: BHAT, Thirumalesh; NAGAPPAN, Nachiappan. Evaluating the efficacy of test-driven development: industrial case studies. In ISESE '06: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, páginas 356–363, New York, NY, USA, 2006. ACM.

É possível observar que o a densidade de defeitos ou falhas diminuiu nos projetos que fizeram uso de TDD. A densidade de defeitos/falhas é uma das métricas que pode ser utilizada para analisar a maturidade de um sistema de acordo com o ISO 9126. A maturidade de um sistema é, de acordo com o ISO/IEC, "o nível em que um sistema, produto ou componente satisfaz as necessidades de confiabilidade em operação normal". Essa métrica depende da fase de testes que está sendo aplicada. Nas fases iniciais de testes quanto maior seu número melhor, porém, nas fases finais de teste o critério é invertido, quanto menor o valor calculado maior é o nível de maturidade.

Isso leva à conclusão que os projetos que fizeram uso de TDD são mais maduros e confiáveis que os que não fizeram uso da técnica. O aumento no tempo de desenvolvimento não implica em nenhum critério de qualidade do produto. Apesar disso não deve ser desconsiderado. O aumento no tempo de desenvolvimento pode, por exemplo, impactar diretamente no orçamento do projeto e no seu tempo de entrega e, indiretamente nas demais fases do desenvolvimento do produto.

7.4 TDD na China

Este estudo de caso compara dois times de desenvolvimento inexperientes que adotaram TDD contra três times que não utilizam TDD. Como os times trabalhavam em produtos diferentes não foi possível analisar os defeitos de forma justa, porém outro dado relacionado a defeitos foi analisado. Foi comparado quanto tempo os programadores precisariam para corrigir os defeitos relatados por usuários durante os testes de aceitação e depois quando o produto já estivesse em produção. A tabela 13 mostra o número de defeitos encontrados nos testes e durante as operações em produção.

Tabela 13: Quantidade de defeitos reportados por usuários no estudo de caso

Time	Quantidade de defeitos
Utilizando TDD	212
Sem utilizar TDD	643

Fonte: Lui, K. M. and Chan, K.C.C. Test Driven Development and Software Process Improvement in China. *In XP 2004,* Garmisch-Partenkirchen, Germany, 2004.

O gráfico 2 mostra o tempo necessário para correção dos defeitos pelos times. O time que usava TDD corrigiu 97% dos defeitos em apenas um dia enquanto o time que não utilizava corrigiu 73%.

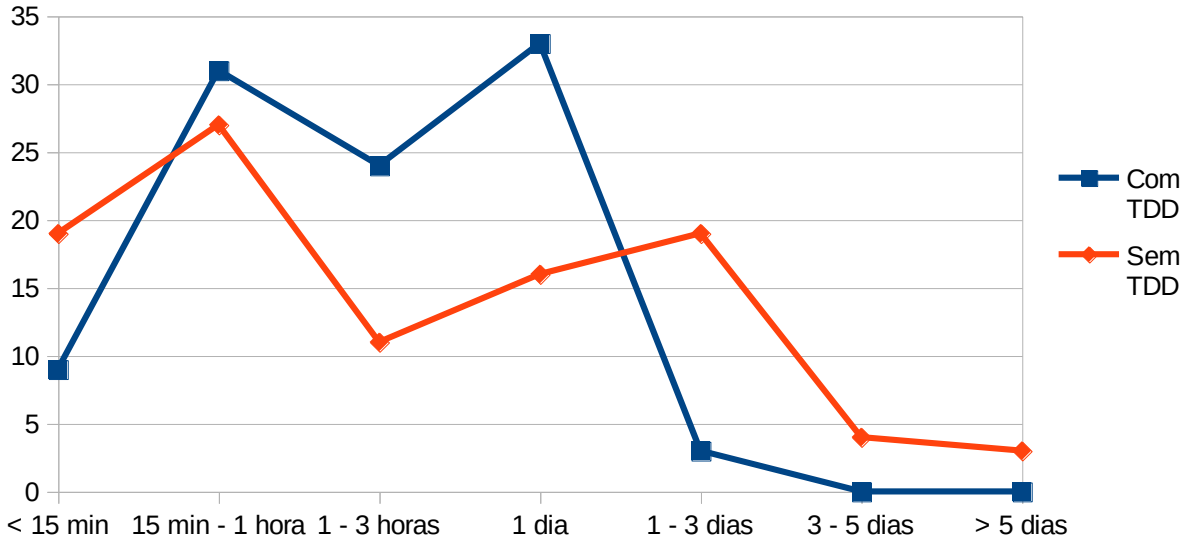

Gráfico 2: Tempo gasto para correção dos defeitos encontrados.

7.4.1 Avaliação dos resultados obtidos

Através dos resultados é possível calcular uma das métricas que podem ser usadas para analisar manutenibilidade. A tabela 14 mostra o cálculo para a métrica "Tempo gasto para implementar uma mudança pelo mantenedor" dado em time/dia.

Tabela 14: Resultado da avaliação dos times utilizando a métrica "Tempo gasto para implementar uma mudança pelo mantenedor"

Time	Valor da métrica
Com TDD	8.58
Sem TDD	19.54

Fonte: Elaborado pelo autor

É possível concluir que os times que utilizaram TDD tornaram os sistemas mais manuteníveis já que esta métrica indica que o melhor resultado é o que tende a zero. Para calcular a métrica foram considerados o número de falhas e o tempo de vinte e quatro horas.

8 CONCLUSÃO E TRABALHOS FUTUROS

8.1 Conclusão

Nesta pesquisa o desenvolvimento guiado por testes, ou TDD, foi abordado como uma possível solução para as falhas nos sistemas de informação. A técnica consiste em um ciclo de cinco etapas: escrever um teste, ver o novo teste falhar, fazer uma pequena alteração para que o teste passe, alterar o código para remover duplicação e rodar todos os testes e vê-los todos passarem. Este ciclo inverte a ordem em que as metodologias de desenvolvimento convencionais executam os testes, os testes são criados e executados primeiro e não mais na etapa final. Esta inversão no ciclo faz com que o novo código seja gerado apenas para que o teste tenha sucesso, desta forma o desenvolvedor consegue tem um controle maior das alterações que estão sendo feitas aumenta a confiança do mesmo já que ele sabe exatamente o precisa ser feito, pois, tem um resultado concreto do que foi desenvolvido apenas executando os testes novamente. Essas e outras vantagens foram encontradas durante a pesquisa, apesar de não serem o foco principal da mesma.

Para que a pergunta problema fosse respondida foi necessário definir os termos: “falha” e “defeito” e quais são as causas mais comuns das falhas em sistemas de informação bem como as estratégias para lidar com as mesmas. Foi possível observar que os testes têm papel fundamental tanto na remoção quanto na prevenção das falhas e, então, os métodos e tipos de testes foram abordados.

Com os conceitos de falha e testes bem definidos foi possível detalhar a prática do desenvolvimento guiado por testes e observar sua relação direta com testes e consequentemente com o aumento de qualidade dos sistemas de informação. Durante a pesquisa ainda foi possível encontrar detalhes sobre os riscos e as desvantagens que a técnica pode proporcionar como: resistência da equipe a mudanças, aumento de custo de manutenção gerado devido a uma base de código maior, entre outros.

Devida a abordagem indutiva da pesquisa foram também expostos resultados de aplicações reais da técnica na indústria de desenvolvimento de sistemas. Por se tratar de uma prática relativamente nova e apesar de seu uso ter se popularizado, ainda faltam mais relatos de sua aplicação na indústria comprovando sua eficácia

para que dessa forma as empresas e os próprios desenvolvedores passem a adotar a prática. Os resultados destas aplicações mostram que o desenvolvimento guiado por testes torna o sistema mais maduro e manutenível do ponto de vista de qualidade proposto pelo consórcio ISO/IEC. Ao adotar o desenvolvimento guiado por testes a quantidade de defeitos foi menor em quatro das cinco aplicações expostas e em uma das aplicações mostrou que a equipe que adotou a técnica conseguiu localizar e remover as falhas de forma mais rápida.

Pode haver limitações quanto à aplicabilidade de TDD em certos domínios de aplicação e contextos, como interfaces gráficas, banco de dados e componentes distribuídos. Os estudos expostos nesta pesquisa também mostram que o desenvolvimento guiado por testes onera o desenvolvimento, aumentando o tempo dedicado a este. Além do tempo o desenvolvimento guiado por testes faz com que a base de código cresça, devido aos testes criados durante o ciclo, e aumente o tempo de manutenção já que conforme o sistema for sendo alterado os testes também deverão ser revisitados para que seja possível utilizá-los para garantir que o sistema se comporta como deveria e a qualidade ser mantida.

Portanto, foi possível verificar que o desenvolvimento guiado por testes é uma alternativa para mitigar as falhas nos sistemas de informação, baseado não só no levantamento bibliográfico, mas, também apoiado nos resultados das aplicações expostos nesta pesquisa.

8.2 Trabalhos futuros

Uma sugestão para trabalhos futuros seria um estudo de caso em uma indústria de software nacional, mas, que contasse com desenvolvedores já habituados com o desenvolvimento guiado por testes para que a comparação entre os resultados finais fosse mais precisa. Um outro estudo de caso interessante seria aplicar a técnica em um projeto já em andamento, com todas as datas de entregas já definidas, para averiguar qual seria o impacto do desenvolvimento guiado por testes não só na qualidade, mas, também no tempo de desenvolvimento. A última sugestão de estudo de caso é destacar as vantagens do desenvolvimento guiado por testes, descritas na literatura, e aplicar métricas as mesmas sendo possível avaliar quais as variáveis que fazem com que a adoção da técnica tenha sucesso ou falhe.

9 REFERÊNCIAS

AMMANN, Paul. **Introduction to Software Testing**. 1.ª ed. Cambridge: Cambridge University Press, 2008.

ASTELS, David. **Test-Driven Development: A Practical Guide (Coad Series)**. 3.ª ed. New Jersey: Prentice Hall PTR, 2003.

BARESI, Luciano; YOUNG, Michal. **Test oracles**. Disponível em: <http://www.cs.uoregon.edu/~michal/pubs/oracles.html>. Acesso em: 20 mar. 2013.

BECK, Kent. **Test Driven Development: By Example**. 1.ª ed. Boston: Addison-Wesley Professional, 2003.

BHAT, Thirumalesh; NAGAPPAN, Nachiappan. **Evaluating the efficacy of test-driven development: industrial case studies**. In ISESE ’06: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, páginas 356–363, New York, NY, USA, 2006. ACM.

BROOKS, Fred. **No Silver Bullet - Essence and Accident in Software Engineering**. Disponível em: <http://www.cs.nott.ac.uk/~cah/G51ISS/Documents/NoSilverBullet.html>. Acesso em: 04 mar. 2013.

DUBINSKY, Yael; HAZZAN, Orit. **Measured test-driven development: Using measures to monitor and control the unit development**. *Journal of Computer Science*, 2007.

FOWLER, Martin, et all. **Refactoring: Improving the Design of Existing Code**. 1.ª ed. Boston: Addison-Wesley Professional, 1999.

GEORGE, Boby; WILLIAMS, Laurie. **A structured experiment of test-driven development**. Information and Software Technology, 46(5):337–342, 2004. *Special issue on Software Engineering, Applications, Practices and Tools from ACM Symposium on Applied Computing*.

GOLD, Russell; HAMMELL, Thomas., and SNYDER, Tom. **Test-Driven Development: A J2EE Example**. 1.ª ed. New York: Apress, 2004.

HETZEL, William C. **The Complete Guide to Software Testing**. 1.ª ed. New York: John Wiley & Sons, 1988.

HUMPHREY, Watts S. **Managing the Software**. 1.ª ed. Addison-Wesley, 1989.

HUNT, Andy; THOMAS, Dave. **Pragmatic Unit Testing in Java with JUnit.** 2.ª ed. Dallas: Raleigh, 2003.

INSTITUTE OF ELECTRICAL AND ELETRONICS ENGINEERS. **IEEE standard for software unit testing**. Technical report, ANSI/IEEE Std 1008-1987.

INTERNATIONAL SOFTWARE TESTING QUALIFICATIONS BOARD. **Certified Tester Foundation Level Management**. Disponível em: <http://www.istqb.org/downloads/finish/16/15.html>. Acesso em: 02 abr. 2013.

ISO/IEC 9126-1: 2000. ***Software engineering– Software product quality***- Part 1: Quality Model.

ISO/IEC 9126-2: 2000. ***Software engineering– Software product quality***- Part 2: External Metrics.

KOSKELA, Lasse. **Test Driven: TDD and Acceptance TDD for Java Developers**. Manning Publications, 2007.

Lui, K. M. and Chan, K.C.C. **Test Driven Development and Software Process Improvement in China**. *In XP 2004,* Garmisch-Partenkirchen, Germany, 2004.

MAXIMILIEN, E. Michael; WILLIAMS, Laurie. **Assessing test-driven development at ibm**. In ICSE ’03: *Proceedings of the 25th International Conference on Software Engineering*, páginas 564–569, Washington, DC, USA, 2003. IEEE Computer Society.

MESZAROS, Gerard. **Xunit Test Patterns: Refactoring Test Code**. 1.ª ed. Westford: Addison-Wesley, 2007.

MYERS, Glenford J. **The art of software testing**. 1.ª ed. New York: John Wiley & Sons, 1979.

PAN, Jiantao. **Software testing.** Disponível em: <http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/>. Acesso em: 04 mai. 2013.

SINIAALTO, Maria; ABRAHAMSSON, Pekka. **A comparative case study on the impact of test-driven development on program design and test coverage**. In ESEM '07: *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, páginas 275-284, Washington, DC, USA. IEEE Computer Society.

SOFTWARE ENGINEERING BODY OF KNOWLEDGE. **2004 SWEBOK GUIDE.** Disponível em: <http://www.computer.org/portal/web/swebok/html/ch5>. Acesso em: 02 mai. 2013.

VIEGA, John; MCMANUS, John. **The importance of software testing**. Disponível em: <http://www.cutter.com/research/2000/crb000111.html> Acesso em: 04 mai. 2013.

WASMUS, Hans; GROSS, Hans G. (2007). **Evaluation of test driven development**. Delft University of Technology.