UNIVERSIDADE PARANAENSE UNIPAR
SISTEMAS DE INFORMAÇÃO – UNIDADE PARANAVAÍ

ROBSON LUÍS DO NASCIMENTO

TESTE DE SOFTWARES, UMA ABORDAGEM SOBRE AUTOMAÇÃO DE TESTES E PROCESSOS DE SOTWARES

PARANAVAÍ - PR
2013

ROBSON LUÍS DO NASCIMENTO

TESTE DE SOFTWARES, UMA ABORDAGEM SOBRE AUTOMAÇÃO DE TESTES E PROCESSOS DE SOTWARES

Trabalho de Conclusão de Curso apresentado à banca examinadora do curso de Sistemas de Informação da Universidade Paranaense – UNIPAR, como exigência para obtenção do grau de Bacharel em Sistemas de Informação, sob orientação da Profa: M. Sc. Késsia Rita da Costa Marchi

PARANAVAÍ

2012

TERMO DE APROVAÇÃO

ROBSON LUÍS DO NASCIMENTO

TESTE DE SOFTWARES, UMA ABORDAGEM SOBRE AUTOMAÇÃO DE TESTES E PROCESSOS DE SOTWARES

Trabalho de conclusão aprovado como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação da Universidade Paranaense – UNIPAR, pela seguinte banca examinadora:

_____

Profa. M.Sc. Claudete Werner
Mestre em Ciência da Computação

_____

Prof. Esp. Willian Barbosa Magalhães
Especialista em Desenvolvimento de Sistemas para Internet

_____

M. Sc. Késsia Rita da Costa Marchi
Mestre em Ciência da Computação

Paranavaí, 05 de Novembro de 2013

DEDICATÓRIA

É uma grande satisfação finalizar mais uma etapa de minha vida, o curso de Sistemas de Informação. A partir de agora começa o curso de minha vida profissional, não encontrarei facilidade no caminho apenas degraus que irei agarrar-me a subir com o conhecimento adquirido durante esta longa caminhada, no qual muitas pessoas fizeram parte. Não poderia deixar de agradecer a Deus que me iluminou todos estes anos, ajudando a conquistar o meu sonho e de meus pais Maria Rosangela e Luís Lúcio, no qual não mediram esforços para ter três filhos formados. As minhas irmãs Michele, e Mecila que sempre estiveram por perto para aconselhar e ajudar a manter o foco nos estudos. Essa família linda que nos momentos de minha ausência dedicados ao estudo do ensino superior, sempre fizeram entender que o futuro, é feito a partir da constante dedicação no presente.

Não poderia deixar de citar aqueles que se dedicaram constantemente, que fazem com amor dedicação e coragem ao passar o pouco que sabem, pois a palavra mestre, nunca fará justiça aos professores dedicados, aos quais, terão meu eterno agradecimento.

Aos amigos que aqui fiz e será cultivado para uma vida inteira, uma segunda família que sem dúvidas nos mantem de pé nos momentos difíceis e que estão juntos na vitória e na derrota.

AGRADECIMENTOS

Á Deus que é a base de tudo, por ter me dado saúde e sabedoria para realizar este projeto, a minha família pelo incentivo no dia a dia e indicando o melhor caminho a seguir, e a orientadora que não mediram esforços e compreensão para o desenvolvimento do estágio e também para os amigos que me ajudaram de alguma maneira atenciosamente. Em especial a professora mestre Késsia Rita Costa March, pois o velho ditado já dizia, “Quem não senta para aprender, já mais ficará de pé para ensinar.” Muito obrigado por acreditar em mim e nesta pesquisa.

RESUMO

Um software computacional quando desenvolvido, independentemente de sua área de atuação, deve no mínimo atender às necessidades para o qual foi desenvolvido. Necessidades estas que podem variar dependendo do cliente e da área de atuação destes. Garantir que um software computacional funcione de forma adequada para satisfazer a solicitação do cliente, não é uma tarefa fácil. Por isso, desenvolver softwares requer pessoas que saibam trabalhar com as expectativas do cliente, também requer uma equipe desenvolvimento madura que acompanhe essas solicitações. Para isso, foram criados processos conhecidos no ramo, como metodologias, que auxiliam gerentes e a equipe de desenvolvimento, possibilitando apresentar ao cliente o andamento da produção de suas solicitações. Como todo produto colocado no mercado, espera-se do mesmo uma qualidade de funcionamento "perfeita". Para garantir que este produto funcione no cliente, da forma como foi solicitado, é necessário, a realização de testes, porém testes manuais são demorados e não cobrem toda funcionalidade do mesmo. Para ajudar neste processo foram criados os testes automatizados que dão ao projeto um maior *feedback*, das funcionalidades do software, ajudando de maneira significativa equipes de desenvolvimento a encontrar problemas durante a fase de desenvolvimento do produto, podendo assim ser corrigido antes de ir para o cliente. Para obter uma maior percepção desta realidade, esse manuscrito realizou uma pesquisa seguindo um estudo sobre os testes automatizados e apresentou uma abordagem de desenvolvimento guiado por testes, técnica está, conhecida como TDD que pode ser acoplado em metodologias ágeis. O mesmo, também possui um estudo de caso, apresentando na prática a técnica abordada durante todo o manuscrito.

Palavras - chave: Testes Automatizados, Testes de Softwares, Desenvolvimento Guiado Por testes, TDD, Scrum, Engenharia de Software, desenvolvimento, Ferramentas de Testes Automatizados.

ABSTRACT

A computer software when developed , regardless of their area of expertise , must at least meet the needs for which it was developed . That these needs may vary depending on the client and the area of operation of these . Ensure that software computational function properly to meet the customer's request , is not an easy task . Therefore , developing software requires people who can work with the client's expectations , also requires a mature development team to monitor those requests . For this, we created processes known in the art , such as methodologies , which help managers and development team , allowing the client to present the progress of the production of their requests . Like every product in the market , we expect the same quality of operation " perfect " . To ensure that this product works on the client , as it was requested , it is necessary to carry out tests , but tests are time consuming and manual do not cover all of the same functionality . To assist in this process were created automated tests that give more feedback to the project , the software features , significantly helping development teams encounter problems during the development phase of the product, and can therefore be corrected before going to the customer . For a greater perception of this reality , this manuscript has conducted research following a study on automated tests and presented an approach to test driven development , technique , known as TDD which can be engaged in agile methodologies . The same also has a case study , showing in practice the technique discussed throughout the manuscript .

Keyword: Automated Testing, Software Testing, Test Driven Development, TDD, Scrum, Software Engineering, Development, Automated Testing Tools.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE SIGLAS

AG - Agile Modeling,

CPF – Cadastro de Pessoa física

MVC – ModeloVisão e Controle

OO - Orientação a Objetos

PO - Product Owner

QAI - Quality Assurance International

TDD - Test Driver Developement.

V&V - Verificação e Validação

XP - Extreme Programming

SUMÁRIO

INTRODUÇÃO

Testes de software, uma abordagem sobre automação de testes e processos de *software*. A crescente evolução do mercado de tecnologia traz um novo paradigma de *softwares* uma vez que há necessidade de estar tudo conectado. Nesta frente, surge a procura por mais qualidade, rapidez e facilidade na utilização do produto final.

É notável a grande mudança na forma como as aplicações são desenvolvidas. Estas formam grandes desafios para engenheiros e desenvolvedores. Metodologias foram criadas para tentar diminuir a quantidade de erros que deixavam o produto final com uma qualidade duvidosa. Foram adotadas metodologias em cascata, espiral, metodologias ágeis, dentre outras, para se chegar a um nível satisfatório de sistema. Todas as metodologias criadas possuem suas qualidades e defeitos, com o único objetivo; de deixar o desenvolvimento mais confiável e com um produto final satisfatório, atingindo todas as exigências do cliente, chegando a qualidade desejada.

Qualidade termo que vem do latim *qualitate*, cujo significado apontado pelo Instituto Nacional de Metrologia, Normalização e Qualidade Industrial (INMETRO), "compreende o grau de atendimento (ou conformidade) de um produto, processo, serviço ou ainda um profissional a requisitos mínimos estabelecidos em normas ou regulamentos técnicos, ao menor custo possível para a sociedade". Neste caso para se chegar a este nível mínimo de qualidade exigido deve se seguir uma documentação estabelecendo os critérios de validação, após esta fase vários testes são realizados em cada produto objetivando a qualidade Fina do mesmo.

O teste de software tornou-se, pouco a pouco, um tema de grande importância, com necessidades de adaptação de métodos práticos que assegurem a qualidade dos produtos finais tornando-os confiáveis e de fácil manutenção. Realizar teste em software é uma forma de verificar se o mesmo atingiu as especificações e funcionou corretamente no ambiente para o qual o mesmo foi projetado chegando a qualidade que se espera dele. (Neto, 2008)

Hoje, muito se fala nas metodologias ágeis em inglês (*Agile Modeling*, ou AG) e nos processos evolutivos de um software. O termo ágil de primeiro momento, nos dá

a impressão de mau planejamento ou falta de controle, porém, ter agilidade em processo de software exige bastante organização, dedicação e controle dos gerentes sobre sua equipe. As metodologias ágeis trouxeram uma maior interação com o cliente fazendo com que os requisitos solicitados não se percam durante a construção do projeto, trazendo o cliente para "dentro da equipe". Desta forma, o software que está sendo desenvolvido tem mais chance de cumprir com as solicitações, evitando que problemas surjam no momento de utilização pelo usuário final deste produto.

1.1 **Proposta**

A proposta deste trabalho é realizar a construção de um projeto científico no qual envolve uma abordagem sobre os testes de softwares destacando a utilização de testes em software. Este também contará com uma abordagem do desenvolvimento guiado por testes o TDD (*Test Driver Developement*). Após esta pesquisa será desenvolvido uma aplicação de exemplo, para validar os conceitos científicos abordados durante a fase de estudo.

O *Test- Driver Developement* (Desenvolvimento guiado por testes) se baseia na construção de software coberto pelos testes, seguindo tecnologias e conceitos de testes de unidade, integração e interface entre as funcionalidades. Contudo, o TDD resume-se em uma técnica de desenvolvimento ágil com um *design* de produção melhor documentado, de fácil entendimento. Portanto, ao final da produção do *software*, pretende-se também, utilizar uma ferramenta de testes automatizados que realizam os testes de um *software* seguindo a metodologia proposta no estudo.

1.2 **Objetivo Geral**

Os objetivos desta pesquisa são estudar e aprender o conceito de testes de software, compreender a especificação de qualidade abordada em softwares computacionais, utilizando-se de técnicas, documentação e ferramentas de testes e compreender as metodologias de desenvolvimento ágil que enfatiza os testes de software, como a técnica de desenvolvimento em TDD.

1.3 **Motivação**

O motivo para este projeto consiste em compreender a busca pela qualidade final de um produto computacional desenvolvido sob medida com a utilização de testes automatizados na validação do produto final, seja ele manual ou automatizado.

Apesar de tantas tecnologias, ferramentas de testes e modelos de processo de desenvolvimento, ainda possuem erros e falhas na produção de um software, pois quando o mesmo é projetado estima-se atingir o maior grau de qualidade possível, com todas suas funcionalidades em execução e seguras. Porém o retrabalho nas aplicações desenvolvidas ainda existe. A correção nas funcionalidades que poderiam ser identificadas durante a construção da aplicação, é constante e isto, ocorre por falha na modelagem, falta de testes ou ainda por falha nos teste que existem, continuam acontecendo mesmo após a entrega do produto.

Como conclusão pretende-se identificar os pontos positivos e negativos deste processo de produção, identificando a viabilidade do desenvolvimento da técnica criada por Kent Beck, o TDD.

1.4 **Organização do Volume**

Este projeto é constituído e separado por capítulos que descrevem as especificações necessárias para se chegar a uma conclusão sobre os testes automatizados e o TDD.

O primeiro capítulo é constituído pela introdução do volume contendo uma breve descrição dos assuntos abordados neste documento.

O segundo capítulo refere-se a metodologia ágil de desenvolvimento computacional, o *Scrum.*

O Terceiro capítulo é constituído por informações relevantes sobre os testes de software, apresentado assuntos como qualidade, testes orientado a objeto, testes automatizados e as ferramentas que auxiliam na gestão dos mesmos.

No quarto capítulo encontram-se informações referentes ao desenvolvimento guiado por testes e como utilizá-lo em um ambiente de desenvolvimento.

O quinto capítulo é composto por uma abordagem referente a adaptação da técnica de desenvolvimento guiado por testes na metodologia ágil *scrum*.

O sexto capítulo é composto pelo estudo de caso referenciando o conteúdo descrito neste documento.

No sétimo capítulo encontra-se a conclusão referente ao estudo realizado e descrito neste documento.

2 METODOLOGIAS ÁGEIS

As metodologias ágeis surgiram, devido as crescentes pressões do mercado por inovação, produtividade (prazos cada vez mais curtos), flexibilidade e melhoria no desempenho dos projetos de desenvolvimento de software.

O ágil surgiu dado a necessidade de melhorarmos a forma como estamos desenvolvendo software. (STEFFEN, 2012).

Em 2001 Kent Beck e 16 outros notáveis desenvolvedores assinaram o termo conhecido como “Manifesto ágil”. (PRESSMAN -2006) Declararam:

- Indivíduo e Interação em vez de processos e ferramentas.
- Software Funcionando em vez de documentação abrangente.
- Colaboração do cliente em vez de negociação de contratos.
- Respostas e modificações em vez de seguir o plano.

Para este, ainda que os dois primeiros valores sejam importantes os itens seguintes devem ser mais valorizados e postos em prática. Processos, ferramentas, documentação, negociação e planos são importantes, porém trabalhar em equipe, entregar o produto, cumprir prazos e entender a necessidade do cliente são fundamentais.

Metodologias não são regras ou livros de receitas para criação de produtos, mas sim, uma maneira de se pensar na construção dos mesmos. Esta maneira de pensar foi criada buscando uma melhor interação com o cliente. Nas metodologias tradicionais, um sistema era construído por inteiro e depois chegava-se a conclusão que o mesmo não servia ao propósito para o qual foi planejado e construído, porque as regras mudaram e as adaptações tornara-se complexa demais para voltar e refazer a aplicação.

A metodologia ágil encoraja atitudes de equipes estruturadas. Enfatiza a rápida entrega do software. Os clientes fazem parte da equipe de desenvolvimento e reconhecem que um plano de projeto deve ser flexível. (PRESSMAN, 2006).

No capítulo a seguir será abordado informações relevantes referente a modelos de processos de desenvolvimento, em especial o *Scrum*, pois este é um dos modelos de desenvolvimento mais utilizados e as informação são de extrema importância.

2.1 Scrum

O *Scrum* é um nome originário do jogo rugby em que um grupo de jogadores se formam ao redor da bola e os companheiros de equipe trabalham juntos para mover a bola pelo campo como exibe a figura 1. Em desenvolvimento de software, o *Scrum* é conhecido como uma metodologia ágil para gestão e planejamento de projetos de software, foi desenvolvido por Jeff Sutherland e por sua equipe no início da década de 1990documentaram e implementaram o *Scrum*, na Easel Corporation. (PRESSMAN, 2006). Esta metodologia junta conhecimentos de Lean, Hirotaka Takeuchi e Ikujiro Nonaka, que uni teorias da gestão do conhecimento apresentando o processo de interação entre o conhecimento explícito e o conhecimento tácito.

Figura1 – Jogada Scrum do jogo rugby (blogs.independent.co.uk/)

Em desenvolvimento de software os princípios do *scrum*, são utilizados para guiar as atividades de desenvolvimento dentro de um processo que se repete sendo chamados de *Sprints*, no qual a equipe envolvida trabalha para chegar a um objetivo definido pelo líder, que é chamado de *scrum* master. As *sprints* podem variar de acordo com o tamanho do projeto. (PRESSMAN, 2006).

Para que esta metodologia funcione papéis são dados aos integrantes. Estes papéis são descritos por Sanchez, 2007sendo eles:

Equipe: responsável por entregar soluções, geralmente é formada por um grupo pequeno entre cinco e nove pessoas, que devem trabalhar de forma auto gerenciada;

Product Owner: Mais conhecido como PO, responsável pela visão de negócios do projeto e é ele quem define e prioriza o *Product Backlog* Geralmente é o papel desempenhado pelo cliente;

Scrum Master: é uma mistura de gerente, facilitador e mediador. Seu papel é remover obstáculos da equipe e assegurar que as práticas de *Scrum* estejam sendo executadas com eficiência.

2.2 Como Funciona o Scrum

‘ *Scrum*é um ciclo de atividades que contém grupos de práticas e papéis pré-definidos. Os principais papéis são apresentados na figura 2.

O *Product Backlog* é uma lista de atividades que podem ser divididas em equipes. Estas atividades podem ser pontuadas, com notas de zero a cem. Estes pontos destacam o nível de dificuldade para conclusão da atividade, exigindo assim um nível de prioridades, levando a uma situação em que podem ser constantemente atualizada e repriorizada. (SCHWABER, 2004). As atualizações podem ocorrer de acordo com a solicitação do cliente. Uma análise de requisito bem definida e planejada, dá ao projeto um *Product Backlog* mais consistente, refletindo diretamente na solicitação do cliente, trazendo maior confiabilidade das informações que devem ser desenvolvidas.

O *Sprint Backlog* é uma lista de tarefas que o *Scrum Team* se compromete a fazer e consiste em uma lista ordenada de tudo que é necessário para o produto final, para que o *Development Team* escolha quantos itens será capaz de desenvolver na próxima Sprint. (NETO, 2012). De acordo com as necessidades e prioridades exigidas pelo PO, a lista de tarefas priorizadas, são apresentadas ao time em uma reunião.

As *Sprints* são unidades de trabalho usadas para satisfazer um requisito. Uma *Sprint* consiste em um espaço de tempo pré-definido. Cada Sprint deve ter um objetivo a ser alcançado pelo *Development Team* (Equipe de desenvolvimento). Esse objetivo resulta no incremento do produto final. (PRESSMAN, 2006). O comprometimento do time em cumprir com o que foi decidido em reunião é de extrema importância, para que as necessidades do cliente seja cumprida no prazo determinado e os gerentes realizarem as estimativas de conclusão do projeto.

O *Product Owner*(PO) está ligado à visão de negócio do projeto. Ele representa o interesse dos clientes que investem no desenvolvimento do produto. Suas responsabilidades estão em gerenciar o *Product Backlog*,e garantir que as funcionalidades que agreguem maior valor ao produto sejam desenvolvidas primeiro. (Libardi e Barbosa, 2010). O PO, escreve as unidades de trabalho e apresenta para o time, determinando qual será o foco do desenvolvimento em cada *Sprint*.

O *Scrum Master* é responsável pelo processo *Scrum.* Sua responsabilidade esta em garantir que o processo seja entendido e aplicado, sendo também responsável por remover quaisquer obstáculos (impedimentos) que sejam levantados pela equipe.

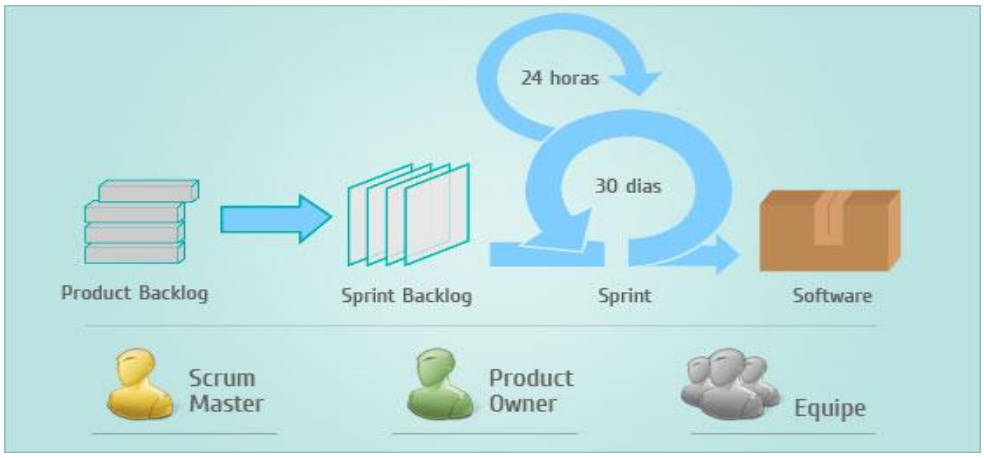

Figura2–Ciclo de desenvolvimento usando o scrum (Brazip)

O ciclo de desenvolvimento se repete até que todas as unidades do *Product Backlog* sejam desenvolvidas e concluídas.

Vale destacar algumas características do *scrum* e seus benefícios quando bem definidos.

- Clientes se tornam parte da equipe de desenvolvimento
- Entregas intermediárias frequentes com 100% de funcionalidade.
- Planos frequentes de contingencia de riscos desenvolvidos pela equipe com discussões de status do projeto.
- Desenvolvimento transparente com problemas que não são ignorados.
- Ninguém é penalizado por reconhecer ou descrever qualquer problema não visto durante a análise de requisitos.

A análise de requisitos bem feita e alinhada de acordo com a necessidade do cliente torna-se fundamental para a conclusão do projeto. Desta forma, é possível obter uma documentação sobre o que deve ser feito durante um período de tempo, junto com a equipe, porem o software somente estará concluído após os testes de validação, e os mesmos devem verificar se o que foi solicitado foi produzido apontaram as possíveis falhas do produto possibilitando a correção.

Um teste de software não pode ser realizado de qualquer forma. Eles devem possuir uma documentação na qual é possível verificar o que está de acordo com o solicitado, podendo as mesmas serem definidas como estratégia de teste.

3 ESTRATÉGIA DE TESTE

Testes refere-se a um conjunto de atividades que podem ser planejadas antecipadamente e conduzidas sistematicamente. Um conjunto de passos no qual é possível incluir técnicas de projeto de caso de teste e métodos, conhecidos como verificação e validação [1](V&V). Em Software computacional uma estratégia é aquela que integra métodos de projetos de casos de testes em uma série bem planejada de passos que resultam no processo de construção do software. (PRESSMAN, 2006).

Alguns autores conceituam testes de software como:

- *Testar software é o processo de executar um determinado software, com a intenção de encontrar defeitos. (MYERS, 1979).*

- *Teste de software é o processo formal de avaliar um sistema ou componente de um sistema por meios manuais ou automáticos para verificar se ele satisfaz os requisitos especificados ou identificar a diferença entre os resultados esperados e obtidos. (IEEE729 1983 - NOGUEIRA 2009).*

O processo de teste de software conduz aos testes de validação, no qual se espera que o sistema seja executado corretamente e atenda aos requisitos espera dos e os teste de defeito que expõe o projeto a possíveis defeitos que causam o funcionamento incorreto da aplicação. Os testes são solicitados para garantir a funcionalidade de um produto, pois é um processo voltado a atingir a confiabilidade do mesmo. (SUMMERVILLE, 2007).

James Bach[2]diz, "A testabilidade do *software* é simplesmente quão fácil [um programa de computador] pode ser testado."

Testar *software* requer do engenheiro planejamento, documentação e dedicação, para estes existem as estratégias de teste que unidas fazem destas, uma técnica necessária e fundamental para atingir a qualidade desejada em um produto.

---

[1]Verificação – Se refere ao conjunto de atividades que garante que o software implementa corretamente uma função específica.
Validação – Se refere ao conjunto de atividades que garantem que o software construído corresponde os requisitos do cliente. (PRESSMAN -2006)

[2] O paragrafo 1994 refere-se a um pôster adaptado no grupo de noticias comp. software-eng, citado por Presmam em seu livro.

Este capítulo é composto por informações relevantes para entendimento sobre qualidade e testes de softwares envolvendo as aplicações computacionais. As informações aqui contidas servem para entendimento sobre testes e qualidade, além da automatização dos processos de teste, verificando a diferença entre os testes orientados a objetos e os testes de softwares tradicionais.

3.1 Por que Testar Softwares?

Com tanta tecnologia disponível no mercado, tanto modelo de processo, ainda é possível fazer esta pergunta? Sim, pois a quantidade de software que não funciona 100% ainda é grande, Aniche (2012), destaca alguns bugs de software que são inclusive famosos: o foguete Ariane 5 explodiu por um erro de software; um hospital panamenho matou pacientes, pois seu software para dosagem de remédios errou. Continua, apontando um dado estatístico dos Estados Unidos em que estima seque bugs de software lhes custemaproximadamente60 bilhões de dólares por ano.

3.2 Por que Não Testamos Softwares?

Relativamente algumas empresas só se preocupam com esta etapa no final do projeto, quando as mesmas designam uma pessoa ou várias para testar um sistema inteiro. E muitas vezes chegam a conclusão de que o projeto ainda não esta pronto por que encontrou erros de validação ou regra de negócio. E isto sai caro. Um ponto que é sempre levantando em qualquer discussão sobre testes manuais versus testes automatizados é produtividade. O argumento mais comum é o de que agora a equipe de desenvolvimento gastará tempo, escrevendo código de teste; antes ela só gastava tempo escrevendo código de produção. Portanto, essa equipe será menos produtiva. (Aniche, 2012).

3.3 Qualidade Em Software

Qualidade é uma característica ou atributo de alguma coisa. A qualidade se refere a características mensuráveis que podem ser comparadas com padrões estabelecidos. Em software estas características podem ser divididas em duas partes, qualidade de projeto e qualidade de conformidade. (PRESSMAN -2006)

- Qualidade de projeto - Refere-se a características que os projetistas especificam para um certo item.
- Qualidade de conformidade - É o grau com que as especificações de projetos são seguidas durante a fabricação do produto.

No entanto, para mensurar as características destacadas algumas perguntas se fazem necessárias e são apresentadas por Ribeiro 2010, referenciando a documentação CMMi for Development 1.2.

Segue o processo corretamente? Esta pergunta que o projetista deve se fazer refere-se a garantia de qualidade (QA).

- "Garantir que o processo seja seguido adequadamente e que seus produtos de trabalho sejam elaborados nos momentos adequados."

Desta forma a possibilidade de obter sucesso na produção do software é muito maior, pois seguir um modelo de desenvolvimento e garantir que as partes que compõem o produto estão sendo construídas sem problemas no momento certo, deixa a sensação de qualidade evidente.

Fiz a coisa corretamente? Refere-se a verificação.

- "Os propósitos da verificação que o produto ou seus produtos de trabalho atendem aos requisitos específicos."

Fazer a coisa certa é sempre um desafio, pois envolve um grau de entendimento muito grande sobre a necessidade do cliente. Verificar se o que foi produzido e o que foi solicitado. Verificar se o que esta sendo executado é o que se espera. Pois cada requisito tem suas particularidades.

- Fiz a coisa Certa? Refere-se a validação

"O objetivo da validação é demonstrar que o componente do produto cumpre o seu uso pretendido quando colocado em seu ambiente de operação."

3.4 Aplicação dos Testes em Software

Um dos problemas para se testar software é o alto custo. Segundo Pressman, 2005, a atividade de teste é um elemento crítico da garantia de qualidade de software e pode assumir até 40% do esforço despendido no desenvolvimento do mesmo.

Para arquitetura de desenvolvimento convencional, os testes podem ser vistos como uma espiral evolutiva para dentro, ao longo de voltas que diminuem o nível de abstração, como destaca a figura 3, (PRESSMAN, 2006). Estes, estão dentro de uma série de quatro passos direção dos testes e também, há o código, que inclui os testes de unidade e o projeto, que engloba o teste de integração e requisitos que é constituído pelos testes de alto nível.

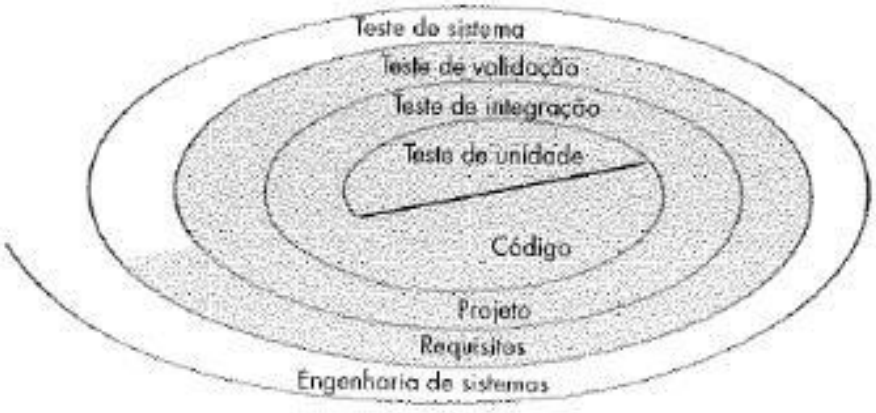

Figura 3 - Estratégia de Testes de Software Fonte: Pressman, 2006

Teste de unidade – Inicialmente, no centro da espiral, há o responsável pelos testes de componentes, assegurando que os mesmos funcionem adequadamente, como uma unidade, sendo ligados para se integrarem e formarem o pacote de software. O Teste de unidade enfoca a lógica interna de processamento e estrutura de dados, garantindo o armazenamento temporariamente, e mantendo sua integridade durante todos os passos de execução de um algoritmo. (PRESSMAN -2006). Em outras palavras, o teste de unidade

foca a descoberta de erros de cálculos, comparações incorretas ou fluxo de controle inadequado.

Testes de Integração – Integra ou unifica os testes unitários. Este, verifica se os componentes que foram testados unitariamente, apresentam problemas de unificação quando testados em conjuntos, focando nas entradas e saídas, chegando ao fim da construção. Iniciando a validação do projeto.

Testes de Validação - Fornecem garantias finais de que o *software* atenda aos seguintes requisitos: os funcionas, comportamentais e de desempenho. Nesta etapa, o cliente deve validar as funcionalidades, verificando se o que foi solicitado está funcionando perfeitamente.

Teste de Sistema – Verifica se todos os critérios foram alcançados e se combina com os outros, como *hardware*, rede, pessoal e banco de dados.

Alguns dos principais tipos de teste estão listados na tabela 1, encontrada no site http://testesdesoftware.blogspot.com.br.Neste, é possível verificar a quantidade de testes existentes e para o que cada um serve, dentro de um modelo de processo existente, em um ambiente de desenvolvimento.

Para controlar, identificar, garantir e documentar os testes que estão sendo realizados no produto existem ferramentas de gerenciamento de projetos como o “redmine”, que auxilia os gestores na tomada de decisão tendo estimativas e prazos para conclusão de etapas de um processo, que são essenciais na qualidade final do produto e na diminuição dos erros de funcionalidade e custos de produção.

Tabela 1 – Tipos de Testes de Softwares

Tipo de Teste	Descrição
Teste de Unidade	Teste em um nível de componente ou classe. É o teste cujo objetivo é um "pedaço do código".
Teste de Integração	Garante que um ou mais componentes combinados (ou unidades) funcionem. Podemos dizer que um teste de integração é composto por diversos testes de unidade*1
Teste Operacional	Garante que a aplicação possa rodar muito tempo sem falhar.
Teste Positivo-negativo	Garante que a aplicação vai funcionar no "caminho feliz" de sua execução e vai funcionar no seu fluxo de exceção. *2
Teste de regressão	Toda vez que algo for mudado, deve ser testada toda a aplicação novamente.
Teste de caixa-preta	Testar todas as entradas e saídas desejadas. Não se está preocupado com o código, cada saída indesejada é vista como um erro.
Teste caixa-branca	O objetivo é testar o código. Às vezes, existem partes do código que nunca foram testadas.
Teste Funcional	Testar as funcionalidades, requerimentos, regras de negócio presentes na documentação. Validar as funcionalidades descritas na documentação (pode acontecer de a documentação estar inválida)
Teste de Interface	Verifica se a navegabilidade e os objetivos da tela funcionam como especificados e se atendem da melhor forma ao usuário.
Teste de Performance	Verifica se o tempo de resposta é o desejado para o momento de utilização da aplicação.
Teste de carga	Verifica o funcionamento da aplicação com a utilização de uma quantidade grande de usuários simultâneos.
Teste de aceitação do usuário	Testa se a solução será bem vista pelo usuário. Ex: caso exista um botão pequeno demais para executar uma função, isso deve ser criticado em fase de testes. (aqui, cabem quesitos fora da interface, também).
Teste de Volume	Testar a quantidade de dados envolvidos (pode ser pouca, normal, grande, ou além de grande).
Testes de stress	Testar a aplicação sem situações inesperadas. Testar caminhos, às vezes, antes não previstos no desenvolvimento / documentação.
Testes de Configuração	Testar se a aplicação funciona corretamente em diferentes ambientes de hardware ou de software.
Testes de Instalação	Testar se a instalação da aplicação foi OK.
Testes de Segurança	Testar a segurança da aplicação das mais diversas formas. Utilizar os diversos papéis, perfis, permissões, para navegar no sistema.

3.5 Testes Para Orientação a Objeto

O modelo e métodos baseados em orientação a objetos surgiram trazendo um enfoque de testes diferente das técnicas tradicionais. Testes de unidade, integração, sistema, aceitação e regressão continuam sendo válidos, porém necessitam de adaptação para o estilo de desenvolvimento OO. (OLIVEIRA, 2010).

Martins e Tschanner, explica em seu artigo que em "orientação a objeto o teste verifica o estado dos objetos e produz um menor número de casos de testes utilizando-se do polimorfismo e das heranças. Esta técnica adota formas mais próximas dos mecanismos humanos para gerenciar a complexidade de um software". Continua dizendo, uma diferença importante do teste de programas procedimentais em relação aos modelos orientados a objetos encontra-se no fato de que as aplicações orientadas a objetos não são executadas sequencialmente.

O teste orientado a objetos consiste em realizar sequências de envios de mensagens. Essas sequências devem ser escolhidas de maneira a explorar o maior número possível de estados que um objeto possa assumir e as transições entre eles.

Ramos, (2010) e Oliveira, (2010) apontam a utilização da forma em que os testes de software em orientação a objeto, siga o seguinte contexto.

• Teste de Classe (substitui teste de unidade clássico)

Um método menor unidade a ser testada. Uma classe na qual o método pertence, sabendo-se que sem a classe não é possível executar um método. Este tipo de teste é realizado como o tradicional teste de unidade em que a classe é a menor unidade a ser testada. As classes são executadas através de objetos, ou seja, para executar uma classe é necessário transformá-la em objeto. Este nível de teste verifica os métodos dentro do objeto classe, as saídas corretas na interação de procedimentos e os estados dos objetos.

• Teste de Cluster

Um software orientado a objeto é composto por várias classes, que interagem entre si. Estes conjuntos, chamamos de cluster. Antes de realizar o teste de cluster é necessário realizar o teste de classe, para que cada classe seja testada individualmente antes de fazer

o teste de cluster. Este nível de teste verifica a interação entre os objetos das diferentes classes e as saídas corretas dos objetos do grupo.

•Teste de sistema

O sistema é feito em cima dos casos de uso que por sua vez dão origem aos casos de teste que também se baseiam em outros requisitos do sistema. São feitas as entradas e verificado se as saídas são as esperadas, dando ao usuário o entendimento e visão final do sistema.

•Teste funcional

Este teste verifica se a funcionalidade da classe está consistente com o que foi especificado. Cada método é analisado pelo menos uma vez, verificando o seu comportamento.

•Teste estrutural

Dentro de uma classe ou de um sistema existem vários caminhos que podem ser percorridos, assim a abertura será expressa a partir da porcentagem de caminhos executados. Dependendo da complexidade do software percorrer todos os caminhos para testá-los torna-se uma tarefa inviável, por isso devemos utilizar critérios para limitar o número de caminhos a um número aceitável e confiável.

• Teste de Interação (substitui teste de integração clássico)

As classes englobam um conjunto de atributos e métodos que manipulam estes atributos. Estes podem interagir para desempenhar funções específicas. Interações entre métodos e atributos dentro de uma classe acontecem o tempo todo. As classes interagem através de uma sequência em comum ou de informações compartilhadas que devem ser verificadas.

• Teste baseado em estados

O teste baseado em estados verifica se os estados apresentados pela classe estão em conformidade com o diagrama. Casos de teste necessitam ser criados para que a classe alcance os estados esperados e possa verificar a conformidade dos mesmos.

• Teste de subclasse

Em OO temos o conceito de herança através do qual cada método da subclasse é classificado como herdado. Quando um método é herdado não há necessidade de um novo caso de teste, pode ser utilizado o caso de teste original do método. Para os métodos novos e redefinidos é necessária a criação de novos casos de teste, pois houve modificação no método original.

• Teste de modelos de análise e projeto OO

O teste feito no final da cadeia não é mais uma realidade em desenvolvimento de software, pois as novas técnicas de teste pregam que o teste deve ser feito durante o processo de desenvolvimento e não só depois dele e este tipo de teste veio para integrar o processo de teste ao processo de desenvolvimento de software.

Quando os testes são incorporados na fase de desenvolvimento existe a possibilidade de detectar erros durante o processo de produção. Estes podem ser corrigidos, reduzindo o tempo de trabalho e os custos e aumentando a qualidade e a confiabilidade dos sistemas desenvolvidos. (Oliveira, 2010). Na fase final do projeto os custos para testar o software é mais auto, caso seja identificado falhas que não estão de acordo com o solicitado pelo cliente, o retrabalho de correção pode prejudicar outras funcionalidades já prontas.

3.6 Testes Automatizados

A automação de parte do teste de software tem sido vista como a principal medida para melhorar a qualidade do produto final. Este consiste em passar para o computador as rotinas de testes utilizando *softwares, frameworks* específicos que realizam esta tarefa, diminuindo a probabilidade de erros, buscando e orientando sua correção. Quando estes são executados corretamente, diminui-se o tempo de teste e de ciclo de vida do software, aumentando a produtividade da equipe, pois sobrará mais tempo para desenvolvimento de funcionalidades, e qualidade do produto final. (ANDRADE, 2010). Os testes que são realizados por ferramentas especializadas na varredura de código e identificação de problemas, são mais eficazes do que testes executados manualmente, além de ser mais rápido.

Os testes devem exibir características que atingem o objetivo de encontrar a maioria dos erros com o mínimo de esforço. (PRESSMAN, 2006). Desta forma, a equipe de desenvolvimento concentra o esforço em outras funcionalidades de maior importância. Nos testes que são executados manualmente Aniche, (2012) mostra em seu livro um caso de teste exibindo uma situação rotineira em ambientes de desenvolvimento.

Imagine um desenvolvedor que deva testar o comportamento do carrinho de compras da loja virtual quando existem dois produtos lá cadastrados: primeiramente, ele "clicaria em comprar em dois produtos", em seguida "iria para o carrinho de compras", e por fim, verificaria "a quantidade de itens no carrinho (deve ser 2)" e o "o valor total do carrinho (que deve ser a soma dos dois produtos adicionados anteriormente)".

Desta forma o desenvolvedor teria de realizar o procedimento manualmente verificando se os campos e suas ações estão de acordo com o caso de teste. Com a utilização do teste automatizado o mesmo procedimento seria realizado em segundos, retornando ao desenvolvedor um *feedback* sobre o procedimento que foi realizado e se este caso de teste esta aprovado ou não.

"Muitos engenheiros pensam ainda que ferramentas de automação farão toda a execução dos testes, desde o planejamento, execução e métricas". (NOGUEIRA, 2010). Na verdade este processo depende do empenho de toda a equipe, pois as ferramentas auxiliam desenvolvedores e gerentes dando a estes um *feedbeak* mais rápido de cada funcionalidade e suas ações, possibilitando correção momentânea (no momento em que a ferramenta identificou o erro), alertando ao desenvolvedor o que deu problema e onde.

Para testar manualmente o caso de teste descrito por Aniche destaca um cenário onde a figura 4 mostra os passos que um testador geralmente faz quando deseja testar uma funcionalidade.

Figura4 – Cenário de testes (Aniche, 2012)

O desenvolvedor primeiro pensa em um cenário (dois produtos comprados), depois executa uma ação (vai ao carrinho de compras), e por fim, valida a saída (vê a quantidade de itens e o valor total do carrinho). Estes passos executados manualmente podem ter

falhas humanas e algumas validações passarem despercebidas. Em projetos de grande investimento pode custar caro o tempo gasto para os testes manuais e as correções deles.

Ribeiro 2010, exibe um gráfico comparativo que mostra os custos dos erros em sistemas, apresentados na figura5. Verifica-se o custo do retrabalho em produção de um produto sendo que este inicia-se a partir da análise de requisitos.

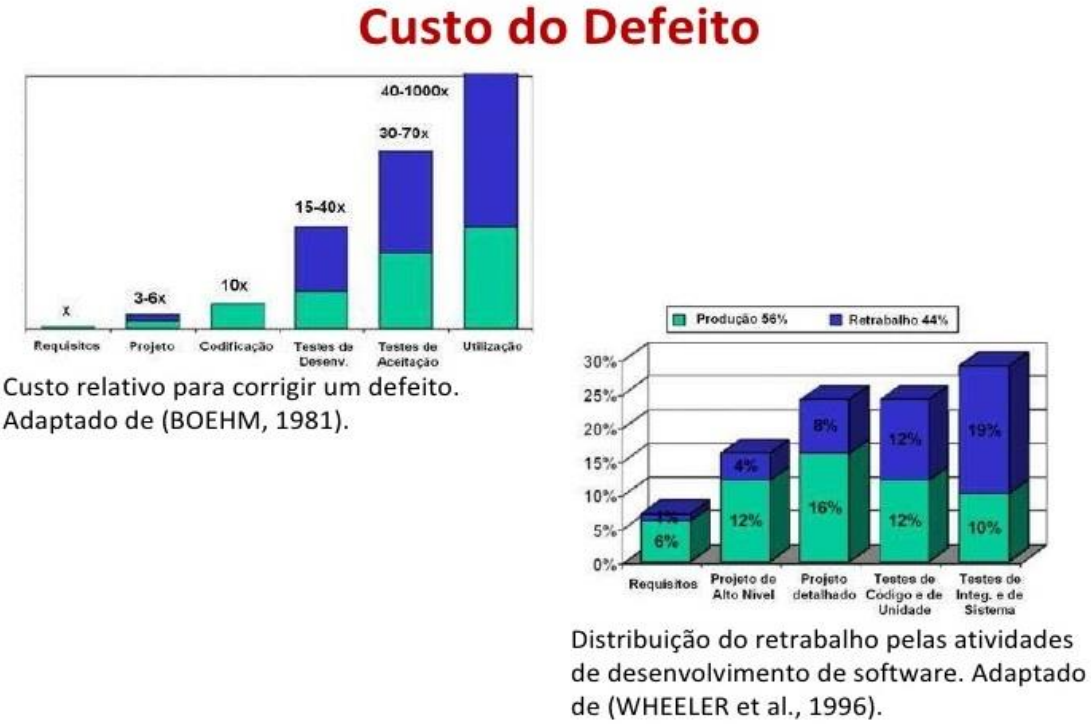

Figura5–Custo dos erros em sistemas (Ribeiro, 2010).

Boehm 1981, destaca o custo da produção e o retrabalho relativo a correção de um defeito durante a produção do produto computacional, onde os problemas começam a aparecerem durante elaboração do projeto passando pela codificação. Quando inicia-se os testes de desenvolvimento o retrabalho aumenta significativamente até a utilização do produto. Wheelleret al., 1996, apresenta que o retrabalho já se inicia nos requisitos do sistema, passando pelo projeto de alto nível e detalhado, subindo os níveis nos testes de unidade. Representa 19% quando chega ao teste de integração e de sistema e um total de 44% de retrabalho na construção da aplicação.

Desenvolver um software que satisfaça as necessidades do cliente e que esteja em conformidade com os requisitos, necessita ter um teste de qualidade, no qual qualidade

significa satisfazer os requerimentos e executar com excelência sua funcionalidade. (CAETANO, 2008).

Segundo a NBR ISO 9000 (2005), "qualidade é o grau no qual um conjunto de características inerentes satisfaz aos requisitos". Ou seja, se um produto ou serviço atinge com exatidão, sem falhas, todas as características do requisito solicitado, chegando-se à conclusão que este possui a qualidade desejada.

Em seu artigo Martinho, 2008, aponta algumas questões fundamentais para a busca de qualidade nos softwares, são elas:

- Tentar prevenir defeitos ao invés de consertá-los;
- Ter a certeza que os defeitos que foram encontrados, sejam corrigidos o mais rápido possível.
- Estabelecer e eliminar as causas, bem como os sintomas dos defeitos;
- Auditar o trabalho de acordo com padrões e procedimentos previamente estabelecidos.

Além das citações de Martinho (2008), Pressman (2005) também relata questões fundamentais para a busca de qualidade na produção do software, sendo eles:

- "Definir explicitamente o termo qualidade de software, quando o mesmo é dito".
- "Criar um conjunto de atividades que irão ajudar a garantir que cada produto de trabalho da engenharia de software exiba alta qualidade".
- "Realizar atividades de segurança da qualidade em cada projeto de software".
- "Usar métricas para desenvolver estratégias para a melhoria de processo de software e, como consequência, a qualidade no produto final".

3.7 Ferramentas de Testes de Software

Para se automatizar um processo de testes de software precisamos passar para o computador as rotinas que seriam executadas, caso fossem feitas manualmente e para esta funcionalidade existem inúmeras ferramentas. CAETANO 2007, diz em seu livro Automação e Gerenciamento de Testes: Aumentando a Produtividade com as Principais Soluções Open Source e Gratuitas, referenciando "*Guideto the CSTE Common Bodyof Knowledge" do QAI (Quality Assurance International* – Garantia de qualidade

internacional), apesar de não existir uma categoria para as ferramentas de testes elas estão agrupadas em 8 áreas distintas, são elas:

1. Ferramentas de automação de testes de regressão;
2. Ferramentas para gestão de defeitos;
3. Ferramentas para testes de Performance/Estresse;
4. Ferramentas manuais;
5. Ferramentas de rastreabilidade;
6. Ferramentas de cobertura de código;
7. Ferramentas para gestão de testes;
8. Ferramentas de apoio à execução dos testes;

Para se utilizar dos benefícios que os testes automatizados proporcionam vale destacar algumas das ferramentas que podem auxiliar de forma significativa na qualidade final do produto.

A Figura6**,** exibe uma lista das ferramentas mais utilizadas que com suas funcionalidades ajudam engenheiros e testadores na realização da automatização dos testes de softwares.

Todas as ferramentas destacadas na figura6tem suas características que ajudam os engenheiros e testadores a tomar decisões durante o projeto. Suas funcionalidades são fundamentais para a qualidade final. As ferramentas apresentadas são de uso nos testes de performance, testes funcionais, testes de estimativas, teste de controle de versão e gerenciamento de projetos incluído a gestão dos testes e gestão dos requisitos.

Estas ferramentas devem ser usadas por pessoas que entendam a necessidade de se automatizar processos e o projeto identificando erros e *bugs* durante a produção, corrigindo-os. Com isso, desenvolver diretrizes e ações para garantir que o produto esteja de acordo com o solicitado pelo cliente e este atinja a confiabilidade e segurança desejada por todos.

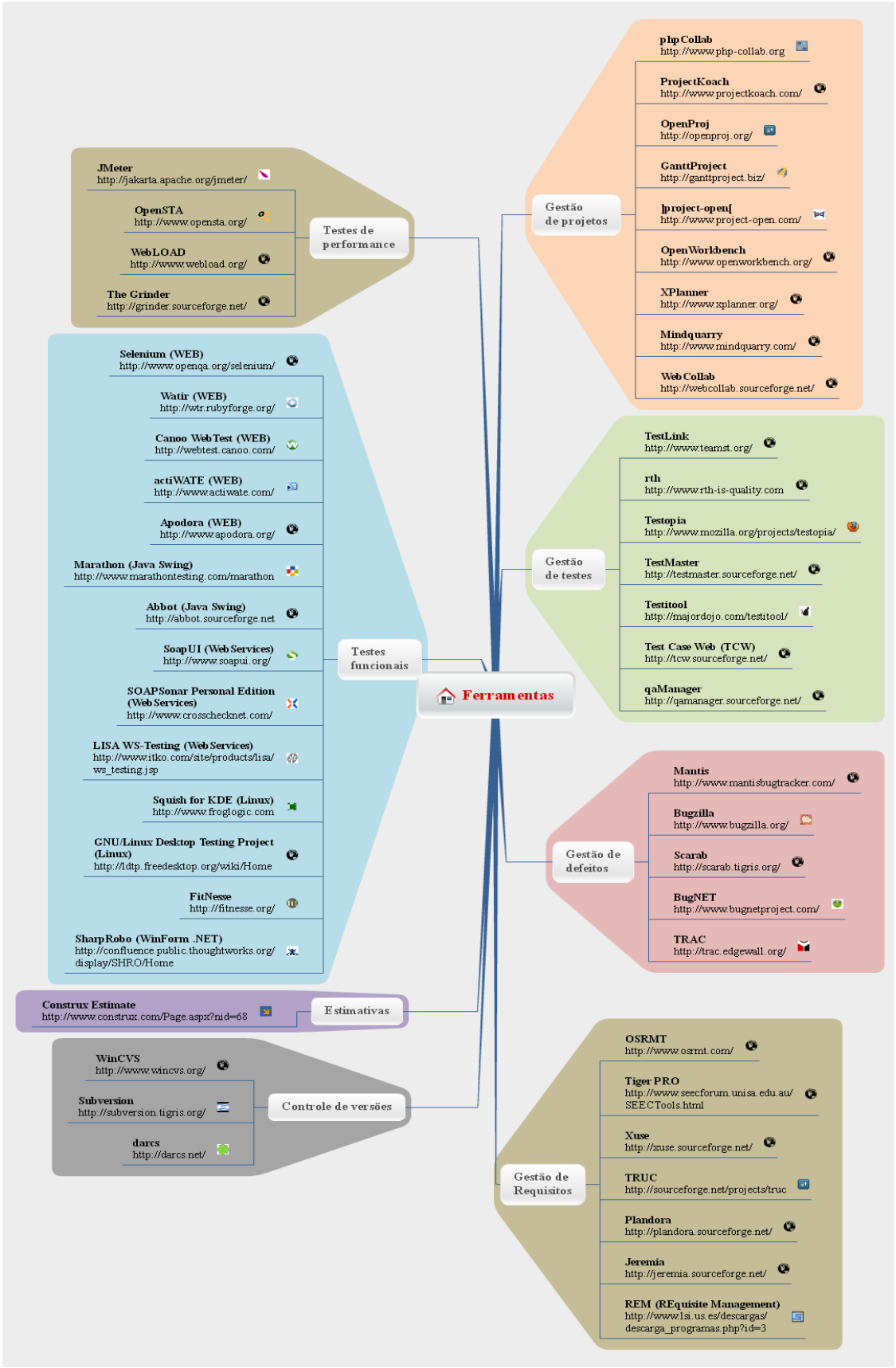

Figura6 - Ferramentas de Software (Cristiano Caetano, 2008)

Testar software requer dedicação de todos envolvidos. É necessário entender que as ferramentas irão auxiliar a equipe, e não fazer o trabalho sozinho. Para que a estes trabalhem juntos, uma técnica conhecida como *Test Driver Development (TDD)*, aborda uma estratégia de desenvolvimento que contempla os testes como foco principal. O TDD, de certa forma dá ao desenvolvedor mais segurança para desenvolvimento de novos métodos ou manutenção neles. Pois ao passar por essas ações ele terá a certeza que um teste foi realizado ao menos uma vez em uma funcionalidade e poderá executar novamente.

4 TDD – TEST DRIVER DEVELOPMENT

Falhas de software são os grandes responsáveis por custos altos em projetos e tempo de trabalho, no processo de desenvolvimento de um produto computacional. Estes erros, muitas vezes identificado muito tarde, seja por falta de documentação ou falha na identificação dos requisitos, afetam de forma significativa a qualidade do software.

Em virtude destes problemas surgiu o Test Driver Development (TDD)ficou bastante popular após a publicação do livro TDD: By Example, do Kent Beck, em 2002. Esta técnica de desenvolvimento trata-se deum processo criado a partir do *Test Frist Programming* do XP essa e uma abordagem que oferece muita agilidade dentro do ciclo de desenvolvimento.

A ideia do TDD é codificar um sistema com 100% de cobertura dos testes. Ou seja, testar toda a aplicação na fase de desenvolvimento, iniciando-se com os testes unitários, garantindo que todas as classes e métodos existentes tenham sido validados e estão de acordo com as especificações do projeto. Testar um projeto após seu desenvolvimento, não se encaixa nesta metodologia. Nessa técnica de produção faz-se uso do processo em que são formadas pequenas interações entre objetos, validando suas ações, em que os testes são codificados primeiro.

Uma vantagem significativa do TDD é que ele permite aos desenvolvedores dar pequenos passos ao escrever software. “Sistemas que não são testáveis não são verificáveis. E um sistema que não pode ser verificado, não pode ser implantado”. (SANTOS, 2009). Para um sistema ser considerado testável, ele deve possuir uma cobertura de testes devendo passar em todos os testes realizados.

O TDD não é uma pratica muito agradável para os programadores, por ser um processo que exige do desenvolvedor mais concentração e esforço mental antes de programar, tornando o desenvolvimento mais demorado. Porém, com essa abordagem é possível chegar a um *designe* onde as classes ficam pequenas e a interação dos objetos mais claras, com um único propósito, controlar a qualidade do código que está sendo escrito, uma vez que sempre pode ser possível refatorar (modificar) o que foi escrito.

Ter a disposição um processo que garanta que classes e métodos estão sendo testados, evita a duplicação dos mesmos, transmitindo mais segurança para os desenvolvedores ao saber que ao realizar manutenções no código, saberão que aquela função já foi testada. (SANTOS, 2009)

Equipes de desenvolvimento estão acostumadas a seguir modelos de processo, nos quais cada uma, com suas características, coloca a equipe em uma rotina. Porém será que é possível inverter a maneira de desenvolvimento destes produtos? ou seja, testar primeiro e depois implementar? E mais importante, faz algum sentido? Para responder a essas perguntas, a seção seguinte aborda informações referente à técnica de desenvolvimento guiado por testes.

4.1 Definição

O TDD é uma técnica de desenvolvimento derivado do método Extreme *Programming* (XP) (Beck 2000) e do Manifesto Ágil. TDD ficou bastante popular após a publicação do livro TDD: *ByExample*, do KentBeck, em 2002. Kent Beck conta que a ideia surgiu ao ler um dos livros de seu pai onde um dos capítulos era abordado uma técnica de testes mais antiga, onde o programador colocava na fita o valor que ele esperava daquele programa, e então o programador desenvolvia até chegar naquele valor. (Aniche, 2012).

A prática envolve a implementação de um sistema começando pelos casos de teste. Apesar de o nome sugerir que TDD esteja relacionado apenas a teste é uma técnica de projeto de software, pois, a mudança da ordem de execução dos testes tem profunda influência no projeto de software. (FERREIRA, 2013). Em1994, Kent Beck escreveu o seu primeiro *framework* de testes de unidade, o *SUnit.* Esta *framework* era específica para *Smalltalk*. Em1995, ele apresentou TDD pela primeira vez na OOPSLA. Em 2000, o JUnit surgiu e Kent Beck, junto com Erich Gamma, publicou o artigo chamado de "*Test Infected*", que mostrava as vantagens de se ter testes automatizados. (Kent Beck).

Segundo CASTRO, (2007),"*Clean codethat Works*" (Código limpo que funciona), foi para alcançar este objetivo que o TDD surgiu e consiste de duas regras principais:

"Código novo só é escrito se um teste automatizado falhar;"

"Todas as duplicações devem ser eliminadas."

São duas regras simples paras eguir o caminho da prevenção, porem tem consequências complexas em um projeto e no comportamento individual e coletivo dos desenvolvedores envolvidos. Para isso é preciso incorporar hábitos que resultem numa maior interação e busque a menor probabilidade de ocorrência de erros.

4.2 TDD Na Prática

Fazendo-se uso da técnica de desenvolvimento guiada por testes em que o processo esteja de acordo com suas especificações, uma aplicação deve seguir os passos descritos na figura 7.Esta abordagem em ciclo, faz com que o *designe* do código seja implementado de maneira "automática" pois a cada refatoração, busca-se uma melhor implementação para

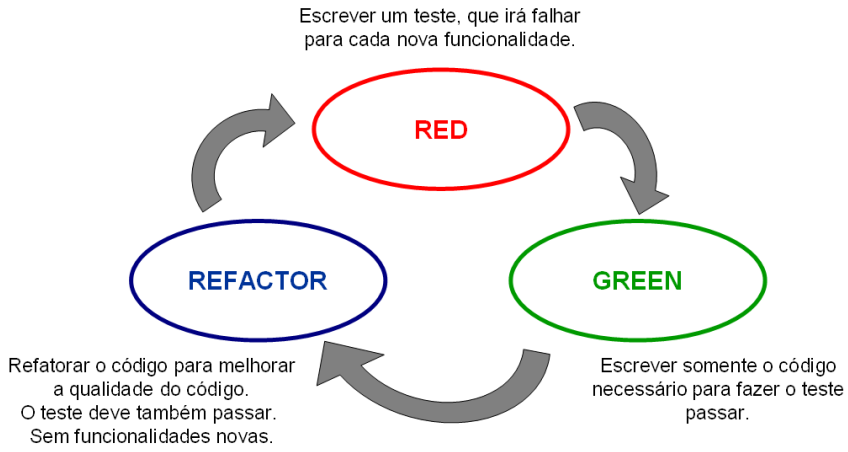

Figura7 - Ciclo da técnica de TDD (WagnerR.Santos)

1- **Escrever um teste simples** e criar uma asserção (assertion) que valide o teste. Ao executar a primeira vez, obviamente o teste deve falhar (vermelho);

2- **Implemente somente o código necessário** para fazer com que os testes passem (verde). Nada mais do que isso, caso contrário, o desenvolvedor perde o foco nos testes;

3- **Refatore o código sempre que necessário**. Então comece o ciclo novamente

Os testes, não são apenas testes, servem também como documentação, fornecendo uma descrição executável do que o código faz (Design), isto facilita para a equipe de desenvolvimento. Durante a execução dos testes, o desenvolvedor consegue detectar erros de implementação, e os testes internos dão suporte aos desenvolvedores, pois ajudam a manter a qualidade do código, evitando que novos bugs sejam adicionados

Passos de Bebê (*Baby Steps).* Este nome é dado pelos praticantes de TDD à maneira como a programação nesta fase é realizada, pois a ideia de sempre tomar o passo mais simples que resolva o problema, naquele momento(e faça o teste passar) dá se a impressão de lentidão na programação. Aniche, (2012) destaca em seu livro que muitos desenvolvedores confundem passos de bebê e destaca, "Se o desenvolvedor está implementando um trecho de código complexo e têm dúvidas sobre como fazê-lo, ele pode desacelerar e fazer implementações mais simples; mas caso o desenvolvedor esteja seguro sobre o problema, ele pode tomar passos maiores e ir direto para uma implementação mais abstrata".

4.2.1 Implementando o TDD

TDD geralmente faz uso de ferramentas e de um framework para a criação de unidades de teste orientadas a objetos. Um exemplo de framework é o JUnit cujas unidades de caso de teste são adicionadas uma por classe pública, usualmente, com o nome <ClassName>TestCase. Seguindo a técnica foi realizada a construção das somas de uma calculadora básica como demonstração. A primeira forma de se utilizar o TDD é de imediato criar os testes unitário, dando um nome para o teste, neste caso SomaCalculadora.

Quando rodamos com sucesso os nossos testes unitários escritos usando TDD, não estamos verificando se a aplicação funciona. Estamos apenas verificando se nossas unidades de código (métodos, por exemplo) ainda se comportam como especificamos anteriormente. (PITANG, 2012)

Analisando o exemplo da figura 7, foi criado uma classe de teste calcSomar, este é implementado pela função @TEST do JUNIT que possui a implantação do método testarSoma. Em sua primeira execução, seguindo as boas práticas do desenvolvimento

guiado por testes o mesmo deve falhar, pois a classe **calculadora** e o método **somar** descritos na figura8, ainda nem foram criados.

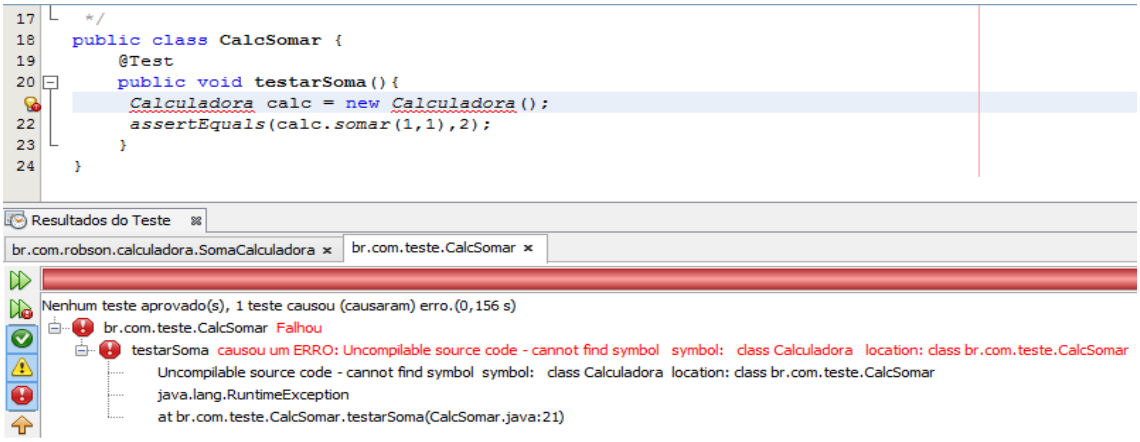

Figura8 - Representação de um teste unitário em TDD

Para que os teste da soma de uma calculadora passe, deve-se implementar as funcionalidades da mesma em uma classe que receba todos os parâmetros que serão utilizados pelos testes. Neste caso a figura 8 pede a soma de 1 e 1 que deve obter o resultado 2.

Para realizar esse cálculo, a figura 9 apresenta a implementação da classe **Calculadora** que faltava para o método calcSomar apresentado na figura 10 funcionar, este método contém a soma de **um inteiro a** e **um inteiro b,** retornando a soma dos **dois valores.** Neste momento, quando executado o teste o mesmo irá passar retornando o ***status*** de **aprovado**, pois neste momento todas partes essenciais para o funcionamento do teste da classe estão corretamente implementados. Não significa que esta é a melhor forma de se implementar essas funcionalidades. O objetivo deste é apenas fazer o código funcionar para se analisar o que deve ser desenvolvido posteriormente. Desta forma, o programador se preocupa somente nas funcionalidades necessárias para o teste passar, evitando desperdício de tempo, pensando em como cada classe deve funcionar. Após a execução deste procedimento vem a refatoração, caso necessário, para melhorar o designe do código implementado.

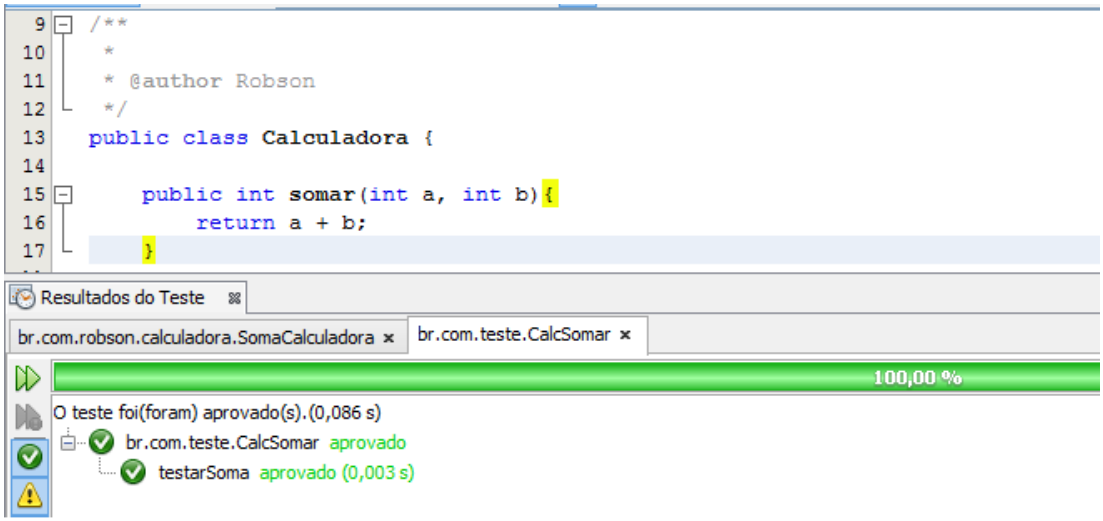

Figura9 – Classe calculadora, exibindo a soma de um inteiro B e um inteiro B

Desta forma, dando continuidade na construção do projeto SomaCalculadora, podem ser criados as implementações da divisão, subtração e multiplicação, na classe **Calculadora,** como exibe o trecho de código da figura 10, alternando somente os sinais para que os mesmos possam realizar suas operações.

Identificando que os testes estão de acordo com a necessidade solicitadas, conclui-se este processo.

Neste momento pode-se pensar em, como melhorar esta implementação? Existe uma maneira melhor de realizar estes cálculos? Como? A resposta para estas questões e outras é o que chamamos de *REFACTOR* ou seja, refatorar.

```
/**
 *
 * @author Robson
 */
public class Calculadora {

    public int somar(int a, int b){
        return a + b;
    }

    public int subtrair(int a, int b){
        return a - b;
    }

    public int dividir(int a, int b){
    return a / b;
    }

    public int multiplica (int a, int b){
        return a * b;
    }
}
```

Figura 10 - Métodos de soma, subtração, divisão e multiplicação da calculadora

É nesta etapa que é possível enxergar os erros de implementação e o código é analisado chegando-se à conclusão que é possível melhorar o *designe* do código trazendo mais segurança para quem desenvolve e para o projeto.

Veja no comparativo apresentado na figura 11, a maneira como o mesmo teste da calculadora foi implementado de inicial, e como este ficou após a refatoração. Durante a verificação da função testarSoma, testarSubtração, testarDivisão.

A mesma técnica serve para o objeto **Calculadora** que pode ser refatorado até que se seja possível identificar uma melhora em sua implementação deixando o código mais limpo, de fácil entendimento e documental para manutenções futuras.

Chega-se a conclusão que quando acabamos, "realmente acabamos", pois ao final do projeto, dificilmente temos que voltar e analisar a implementação. Utilizando o TDD este processo já foi todo realizado durante sua construção, com isso, é possível chegar a um nível de qualidade muito maior. (http://improveit.com.br/xp/praticas/tdd)

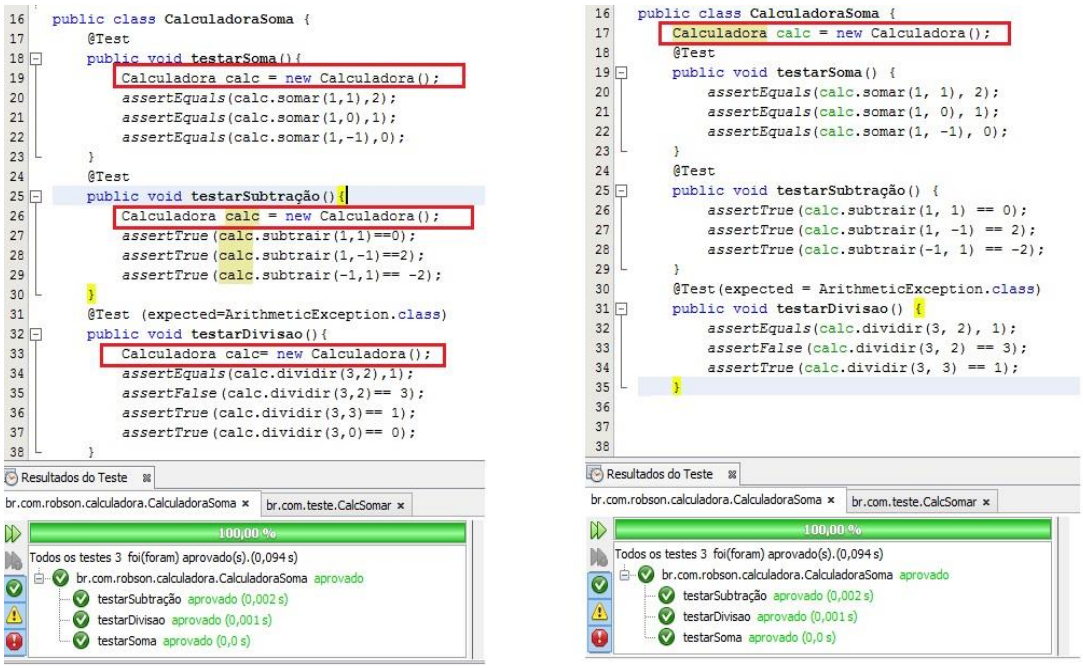

Figura11 - Comparativo entre os métodos de testes

Desta forma, a utilização do desenvolvimento orientado a objeto, junto com a técnica de desenvolvimento guiado por testes, torna-se muito mais eficaz agregando uma maior confiança na implementação dos códigos, pois utilizando-se da execução de testes,

durante o desenvolvimento garante que o mesmo funciona antes de entrar em união com outros métodos.

Estes testes são conhecidos como testes de unidade, e validam o código isoladamente, garantindo que o mesmo esteja funcionando.

Podemos escrever testes de diferentes maneiras, tudo depende do que se espera como resultado. Se quisermos um teste que se pareça com o mundo real, ou seja, que realmente teste a aplicação do ponto de vista do usuário, é necessário escrever o que chamamos de teste de sistema. Estes possuem os testes unitários executados em conjunto, conhecido como teste de integração. A ideia de escrever testes antes, pode ser aplicada a qualquer contexto Independente do *framework* escolhido para ajudar no desenvolvimento (Struts, Spring MVC, VRaptor, JSF, etc), em ambos, o desenvolvedor é obrigado a escrever uma camada que "conecta" o mundo web, cheio de requisições, respostas, HTTP e HTML. (Aniche, 2012). E para auxiliar nas validações dos sistema pelo lado do usuário, existem várias frameworks, como o *Selenium* que realiza as requisições, abre o browser e manipula os elementos.

5 SCRUM COM TDD

Este capítulo está composto por informações referente a metodologia de desenvolvimento *scrum* alinha com a técnica de desenvolvimento guiado por testes. Neste, é possível identificar os benefícios e os problemas que a união destas práticas podem possuir.

O Scrum com seu processo de gerenciamento ágil de projetos possui fundamentos para otimizar a previsibilidade e controlar riscos (Schwaber, 2011).

Segundo Astels, (2003) o TDD é um estilo de desenvolvimento que traz melhorias significativas ao projeto, pois, o código é baseado nos testes de unidade e de componentes sendo o desenvolvedor, o responsável pela criação e execução destes testes.

No entanto, ALVES, (2011), culturalmente, os desenvolvedores têm resistência à adoção da prática TDD, pois eles precisam produzir códigos extras e, com isso, acreditam que será perda de tempo.

O processo de testes fundamental (ISO, 2009) possui etapas bem definidas e sequenciais, por exemplo: Gerenciamento e Planejamento, Projeto, Implementação, Execução e Análise. A necessidade de adequar praticas, " a falta de um processo de testes adaptado para o desenvolvimento ágil é um dos obstáculos encontrados pelos times ágeis em criar a estratégia para a realização dos testes. Isto pode provocar a criação de uma estratégia de testes deficiente, ou mesmo, a não criação deste artefato". (ALVES et al, 2011).

Nesta situação o mau planejamento age diretamente na execução das atividades prejudicando a equipe e o produto. A organização destas estratégias necessitam estar alinhadas com o conhecimento do time, adequando as características abordadas por France e Rumpe, (2000). A agilidade de um processo é determinada pelo grau em que uma equipe de projeto pode dinamicamente adaptar o processo com base nas mudanças, no ambiente e nas experiências coletivas dos membros da equipe de produção.

"Três papéis são sugeridos para compor o time de testes deste processo: o Líder de Testes, o Projetista de Testes e o Analista de Testes que devem estar treinados nas metodologias ágeis, a fim de entenderem e colocarem em prática os princípios do

Manifesto Ágil". (ALVES et al, 2011). Estes planejam e auxiliam a equipe durante o processo de desenvolvimento.

No scrum, as *Sprint* que são o objetivo a ser alcançado pelo *Development Team*, devem ser medidas de maneira que contemplam a nova forma de desenvolvimento, pois o tempo de trabalho do desenvolvedor será maior e a equipe deverá se adaptar a esta forma. O desenvolvedor deverá testar as linhas de códigos criadas por ele, seguindo as práticas da metodologia em questão, garantindo o funcionamento como um todo do que está sendo criado ou alterado. Durante a fase de desenvolvimento é preciso que o próprio teste em execução, faça a validação e informe o desenvolvedor, caso o resultado não seja o esperado. Desta maneira para melhorar a forma como os testes são realizados, somente introduzindo uma framework de teste automatizado que agilizaria o processo de teste. Essa framework nos daria um relatório mais preciso sobre a quantidade de linhas e métodos que foram aprovados e a quantidade que falharam. (Aniche, 2012)

Como descrito em capítulos anteriores, em TDD, o desenvolvedor deve seguir os seguintes passos:

- Escrever um teste de unidade para uma nova funcionalidade;
- Ver o teste falhar;
- Implementar o código mais simples para resolver o problema;
- Ver o teste passar;
- Melhorar (refatorar) o código quando necessário.

A implementação começa pelo teste e, deve o tempo todo, fazer de tudo para que seu código fique simples e com qualidade.

No scrum, essa rotina deve se repetir, com todos os desenvolvedores da equipe. Desta forma, o PO ao validar as solicitações que foram requisitadas e produzidas pelo time, tem mais segurança e garantia de que as solicitações desenvolvidas estão funcionando corretamente podendo verificar, se necessário, em uma suíte de testes que pode ser fornecido pela framework. Os relatórios de teste, a quantidade e quais testes foram feitos para garantir que o requisito foi realmente cumprido.

As vantagem de introduzir, no scrum o TDD, é que desenvolvedores focam no teste concentrando-se em fazer apenas o teste passar e não na implementação. De forma

que o programador se concentra no que deve fazer para o teste passar e não em como a implementação deve ficar. Se este terá que encapsular, se será usado *Hash Map, if, else, for, List,* etc. Código nasce testado. Ou seja, nesta prática é possível verificar que ao se concluir um requisito, o código foi testado ao menos uma vez, garantindo que a funcionalidade para o qual o código foi escrito está funcionando. Simplicidade, o praticante de TDD, escreve código que apenas resolve os problemas que estão representados por um teste de unidade. (Aniche, 2012)

A observação feita por Aniche, exemplifica de maneira simples, a garantia que o desenvolvedor possui ao seguir os paços da prática do código com qualidade.

> *Uma das vantagens de fazer o teste passar de maneira simples e rápida é "testar o teste". Seu teste é código; e ele pode ter bugs também. Como não faz sentido escrever um teste para o teste, uma maneira de testá-lo é garantir que ele falhe quando precisa falhar, e passe quando precisa passar. A ideia de também ver o teste passar rapidamente é justamente para que você "teste o teste".*

Chega-se a conclusão que, a união da técnica de desenvolvimento guiado por testes, e a metodologia de desenvolvimento ágil scrum, podem trabalhar juntas, trazendo muitas melhorias para o projeto. Porém, para que as mesmas reflitam na qualidade do produto final, programadores e projetistas devem estar preparados para as mudanças que esta abordagem proporciona. Programadores devem garantira a funcionalidade das linhas de códigos, e projetistas devem ficar atentos as estimativas do projeto.

6 ESTUDO DE CASO

Preparando o ambiente, para o estudo de caso, será utilizado a ide Eclipse com a utilização das *frameworks Hibernate*, *plug-ins* do *selenium* web driver e o *JUnit* para testes unitários. Para controle de versão do projeto, será utilizado o repositório do *git hub*.

Após a configuração do ambiente de trabalho e criar um novo projeto web com a especificação Java EE deve ser realizado as configurações de conexões com o banco de dados utilizando *Hibernate*.

Seguindo Exemplos de implementação em TDD por Caldas, (2013), os passos a seguir, são uma das maneira possíveis de se trabalhar com TDD.

Será criado a configuração para persistência dos dados com *hibernate* no banco de dados. Esta frameworks armazenara as informações necessárias para conexão com o banco de dados, chamada de *Hibernate.cfg.xml*, exibida no quadro 1.

```
<?xmlversion="1.0"encoding="UTF-8"?>
<!DOCTYPEhibernate-configurationPUBLIC"-//Hibernate/Hibernate Configuration
DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<propertyname="hibernate.connection.driver_class">org.gjt.mm.mysql.Driver</
property>
<propertyname="hibernate.connection.password">root</property>
<propertyname="hibernate.connection.url">jdbc:mysql://localhost/sistema</pr
operty>
<propertyname="hibernate.connection.username">root</property>
<propertyname="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prope
rty>
<propertyname="hibernate.hbm2ddl.auto">update</property>
<propertyname="hibernate.current_session_context_class">thread</property>
</session-factory>
</hibernate-configuration>
```

Quadro 1 - Código para o Hibernate.cfg.xml

A próxima entidade *HibernateUtil*, exibida no quadro 2 - é responsável por devolver a sessão para o usuário, no qual este cria uma configuração e atribui esta configuração do arquivo xml do quadro 1, que persiste os dados com esta conexão.

```
package br.com.sistematdd.util;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
publicclass HibernateUtil {
	privatestaticfinal SessionFactory session = buildSessionFactory();
	privatestatic SessionFactory buildSessionFactory() {
	try {
	Configuration cfg =  new Configuration();
	cfg.configure("hibernate.cfg.xml");
	return cfg.buildSessionFactory();
	} catch (Throwable e) {
	System.out.println("Houve um problema com a conexão ! Verifique o log
do servidor!\n " + e );
	thrownew ExceptionInInitializerError();
		}
	}
	publicstatic SessionFactory getSession() {
		returnsession;
}
}
```

Quadro2 – Código para HibernateUtil

O quadro 3 exibe o filtro de sessão, que deve ser adicionado ao web.xml do projeto

```
package br.com.testetdd.util;
import java.io.IOException;

publicclass ConexaoHibernateFilter implements Filter{

private SessionFactory sf;

publicvoid destroy() {
}

publicvoid doFilter(ServletRequest servletFilter, ServletResponse
servletResponse,FilterChain chain) throws IOException, ServletException {
try {
	this.sf.getCurrentSession().beginTransaction();
	chain.doFilter(servletFilter, servletResponse);
	this.sf.getCurrentSession().getTransaction().commit();
	this.sf.getCurrentSession().close();
	} catch (Throwable ex) {
	try {
	if(this.sf.getCurrentSession().getTransaction().isActive()){
		this.sf.getCurrentSession().getTransaction().rollback();
	    }
	} catch (Throwable t) {
		t.printStackTrace();
  }
thrownew ServletException();
 }
}
publicvoid init(FilterConfig conf) throws ServletException {
	this.sf = HibernateUtil.getSession();
}
```

```
}
```

Quadro3- Filtro de sessão

Nesta fase, será criado a entidade principal que tem as informações do cliente. Após este o nome da classe (quadro 4), deve ser mapeado ao hibernate.cfg.xml

```
publicclass Cliente {

	@Id
	@GeneratedValue
	private Integer id;
	private String cpf;
	private String email;
	private String endereco;
	privateStringnome;
	@Column(name = "data_cadastro")
	private Date dataCadastro;
	privatefloatrenda;

	public Cliente(){

	}
	public Cliente(String cpf, String email, String endereco, String
nome,
			Date dataCadastro, float renda) {
	super();
	this.cpf = cpf;
	this.email = email;
	this.endereco = endereco;
	this.nome = nome;
	this.dataCadastro = dataCadastro;
	this.renda = renda;

	}
//get e set
```

Quadro4 - Código da entidade Principal

O próximo passo, exibido no quadro 5, é a implementação dos testes no qual possui anotações especificas para a execução dos testes pela ferramenta Junit, são elas @BeforeClass, que prepara as informações antes de inicial o processo de teste, o @AfterClass que executa as informações depois da execução dos testes. E o @Test que são os testes. Estas anotações são necessárias para que a framework saiba o que fazer com os métodos de códigos implementados.

```
 1.   packagebr.com.testestdd.clienteTeste;
 2.   importstatic org.junit.Assert.*;
 3.   importjava.util.Date;
 4.   importjava.util.List;
 5.   importorg.hibernate.Session;
 6.   importorg.hibernate.Transaction;
 7.   importorg.junit.*;
 8.   importbr.com.testetdd.cliente.*;
 9.   import br.com.testetdd.util.HibernateUtil;
10.   publicclassClienteTeste {

11.   privatestaticSessionsessao;
12.   privatestaticTransactiontransacao;
13.   @BeforeClass
14.   publicstaticvoidabreConexao() {
15.   sessao = HibernateUtil.getSession().getCurrentSession();
16.   transacao = sessao.beginTransaction();
17.   }
18.   @AfterClass
19.   publicstaticvoidfechaConexao() {
20.   try {
21.   transacao.commit();

22.   }catch (Throwable e) {
23.   System.out.println("Problema ao realizar o commit : " +
      .getMessage());
24.   }finally {
25.   try {
26.   if (sessao.isOpen()) {
27.   sessao.close();
28.   }
29.   }catch (Exception e2) {
System.out.println("Problema no fechamento da sessão: "+ e2.getMessage());
30.   }
31.   }
32.   }
```

Quadro5 – Código para Cliente Teste

Para verificação se a configuração realizada esta funcional, o quadro 6 exibe o código do primeiro teste automatizado. Após as linhas de código é possível visualizar um relatório do teste realizado pela ferramenta de testes.

```
@Test
publicvoidsalvarTest() {
Cliente c1 = newCliente();
c1.setNome("Luís Fabiano ");
c1.setEndereco("Rua Teste");
c1.setRenda(5000f);
c1.setCpf("123456789");
sessao.save(c1);
assertNull(null);
}
```

Quadro 6 – Código paraprimeirotesteautomatizado

Ao executar o teste com a ferramenta na figura 12, é possível verificar sua validação, no relatório de execução, exibindo o tempo que foi necessário para executar esta inserção, a quantidade de erros que o teste apresentou e a quantidade de falhas do teste.

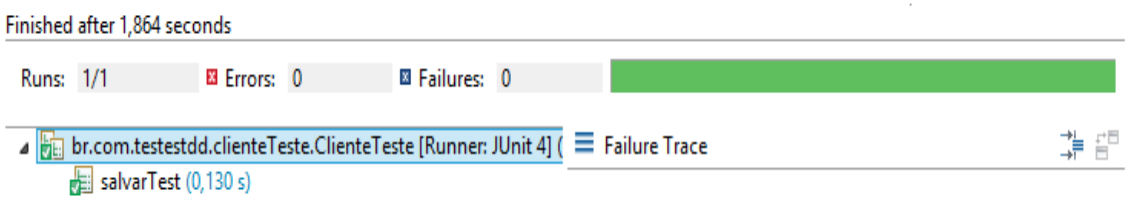

Figura12 – Relatório de Execução do JUnit

Para testar a gravação de vários clientes na base, este método teria de ser executado várias vezes, alterando somente os dados que serão gravados no banco de dados. Porem desta forma, o processo de testes automatizados ficaria trabalhoso. A ferramenta de teste também proporciona a utilização da anotação @Before. Como descrito anteriormente a anotação *before irá* preparar os dados antes dos testes serem executados, pois quando estes forem executado, as informações já estarão gravadas no banco de dados.

```
1.   @Before
2.   publicvoid setup() {
3.   Cliente c1 = newCliente("23132313336", "teste1@mail", "Rua são
     paulo","Luís Fabiano 1", new Date(), 2000);
4.   Cliente c2 = newCliente("09576813209", "teste2@mail", "Rua São
     joão","Rogério Ceni 2", new Date(), 3000);
5.   Cliente c3 = newCliente("81548193056", "teste3@mail", "Rua
     laguna","Júlio Cesar 3", new Date(), 4000);
6.   }
```

Quadro7– Código com anotação @Before

Para verificar, se os registros inseridos no código do quadro **8,** foram gravados no banco de dados, um teste de busca pode ser realizado. No exemplo do quadro 8, uma busca pelo e-mai, é realizado, para verificar se o CPF do cliente cadastro existe na base de dados.

```
@Test
publicvoid pesquisaClienteSalvoTest() {
String sql = "from Cliente c where c.email like :email";
Query consulta = sessao.createQuery(sql);
consulta.setString("email","%e3%");
Cliente clientePesquisado = (Cliente) consulta.uniqueResult();
assertEquals("81548193056", clientePesquisado.getCpf());
}
```

Quadro9 – Código de teste para listar os clientes cadastrados

Como forma de validação, após a execução do teste do quadro 9, a figura 13, exibe o relatório com os resultados obtidos após o teste ser realizado.

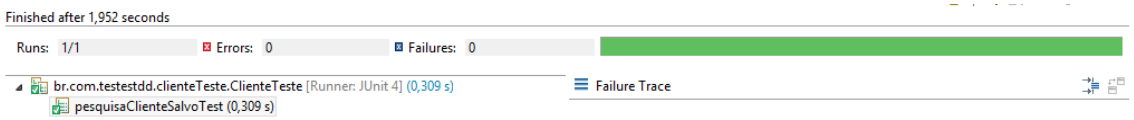

Figura13 – Relatório do *JUNIT* com os resultados do teste de pesquisaClienteSalvoTest

Caso o CPF do cliente pesquisado não exista no banco de dados, o exemplo da figura 14, exibe o mesmo relatório do teste, porem informando ao desenvolvedor que um problema aconteceu durante a validação do teste. O resultado que era esperado, comparando com resultado obtido pela *framework*, não foi o mesmo. Neste momento a ferramenta mostra ao desenvolvedor onde aconteceu o problema, para que o mesmo possa ser corrigido e executado novamente.

expected:<8154819[]056>butwas:<8154819[3]056>

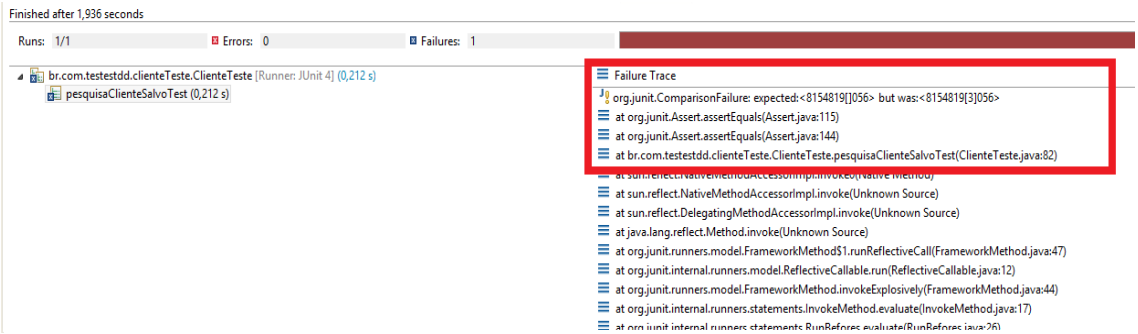

Figura14 - Relatório do *JUNIT* com os resultados do teste de pesquisa ClienteSalvoTest caso o cpf não esteja na base de dados

Para saber a quantidade de clientes que tem gravado no banco de dados, um teste de listagem de cliente pode ser executado. O quadro 10, apresenta as informações da seguinte forma.

```
@Test
publicvoidlistarClienteTest() {
Criteria lista = sessao.createCriteria(Cliente.class);
List<Cliente> clientes = lista.list();
assertEquals(3, clientes.size());
}
```

Quadro10 – Código para listarClienteTest

Para excluir um cliente da base de dados, a mesma consulta realizada para pesquisar os clientes salvos pode ser aproveitada. Para verificar se o cliente foi realmente excluído, no quadro 11 é informado a ferramenta que a expectativa do retorno seja nula, utilizando o *assertNull(**null**)*da ferramenta.

```
@Test
publicvoid excluirClienteTest() {
 String sql = "from Cliente c where c.email like :email";
 Query consulta = sessao.createQuery(sql);
consulta.setString("email","%e3%");
 Cliente clienteDeletado = (Cliente) consulta.uniqueResult();
sessao.delete(clienteDeletado);
assertNull(null);
}
```

Quadro11 – Código excluirClienteTest

Para alterar um registro que foi gravado no banco de dados, a mesma consulta realizada para excluir o cliente é utilizada para alterar, passando para o atributo nome, a nova informação que deve ser gravada no banco de dados. Ao fim, verifica-se o nome inserido foi mesmo gravado no banco de dados, como exibe o código do quadro 12.

```
@Test
publicvoid alterarTest() {
String sql = "from Cliente c where c.email like :email";
Query consulta = sessao.createQuery(sql);
consulta.setString("email","%e2%");
Cliente clienteAlterado = (Cliente) consulta.uniqueResult();
clienteAlterado.setNome("Murici Ramalho");
sessao.update(clienteAlterado);
```

```
clienteAlterado = (Cliente) consulta.uniqueResult();
assertEquals("Murici Ramalho", clienteAlterado.getNome());
}
```

Quadro12 – Código alterarTest

Seguindo as práticas do TDD, neste momento é hora de verificar o que foi desenvolvido. Até o momento foi utilizado o código mais simples para ver o teste passar. Nota-se, os métodos que busca as informações no banco de dados se repetem várias vezes, em vários testes. Estes métodos podem ser modificados e otimizados deixando o código mais limpo e de fácil entendimento. É hora de "Refatorar". O quadro 13, exibe os códigos de teste para a refatoração.

```
private Query pesquisarCliente(Stringparametro) {
String sql = "from Cliente c where c.email like :email";
Query consulta = sessao.createQuery(sql);
consulta.setString("email","%"+parametro+"%");
return consulta;

}

@Test
publicvoidpesquisaClienteSalvoTest() {
Query consulta = pesquisarCliente("e3");
Cliente clientePesquisado = (Cliente) consulta.uniqueResult();
assertEquals("81548193056", clientePesquisado.getCpf());

}


@Test
publicvoidalterarClienteTest() {
Query consulta = pesquisarCliente("e2");
Cliente clienteAlterado = (Cliente) consulta.uniqueResult();
clienteAlterado.setNome("Murici Ramalho");
sessao.update(clienteAlterado);
clienteAlterado = (Cliente) consulta.uniqueResult();
assertEquals("Murici Ramalho", clienteAlterado.getNome());

}

@Test
publicvoidexcluirClienteTest() {
Query consulta = pesquisarCliente("e3");
 Cliente clienteDeletado = (Cliente) consulta.uniqueResult();
sessao.delete(clienteDeletado);
assertNull(null);

}
```

Quadro13 – Códigos de Teste Refatorados

O relatório dos testes com as mudanças realizadas no quadro 13 é exibida pela figura 15. Neste momento, a framework relatou três erros e uma falha. Este problema está acontecendo, pela ocasião em que, os método foram executado mais de uma vez, e desta forma, os dados foram duplicados no banco de dados. Ou seja, no momento de validar as informações dos testes é encontrado mais de um registro, e os testes executados anteriormente são para apenas um registro.

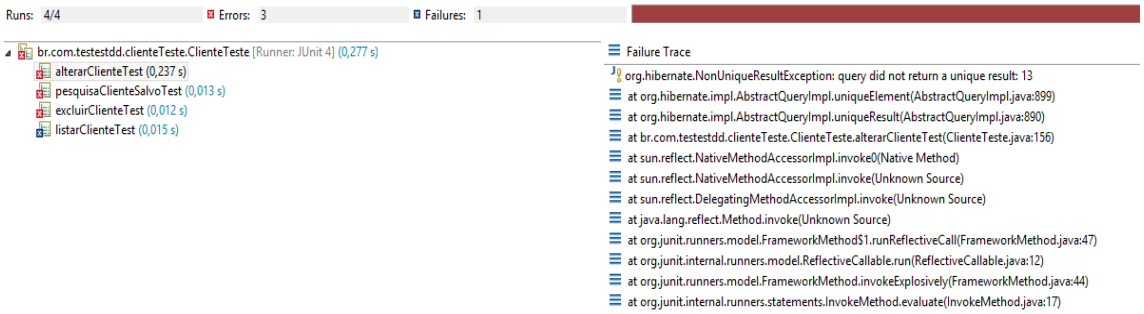

Figura 15 - Relatório do *JUNIT* com os resultados do teste com as alterações

Para resolver o problema, o quadro 14 exibe as informações em que o banco de dados deve ser limpo, depois que todos os testes são executados e validados suas informações, e seus retornos. Por este motivo, em ambiente de produção os códigos de testes e os dados de teste, não podem estar no mesmo pacote dos códigos de produção. Código de teste e código de produção devem ser executados em banco de dados diferentes.

```
@After
publicvoidlimpaInformacoesDoBanco(){
Criteria lista = sessao.createCriteria(Cliente.class);
@SuppressWarnings("unchecked")
List<Cliente> clientes = lista.list();
for (Cliente cliente : clientes) {
sessao.delete(cliente);
}
}
```

Quadro 14 –Código de teste para limpar informações do Banco de dados

Ao executar o teste novamente a figura 16 exibe o relatório da ferramenta, onde a mesma realiza as verificações e as validações desenvolvidas.

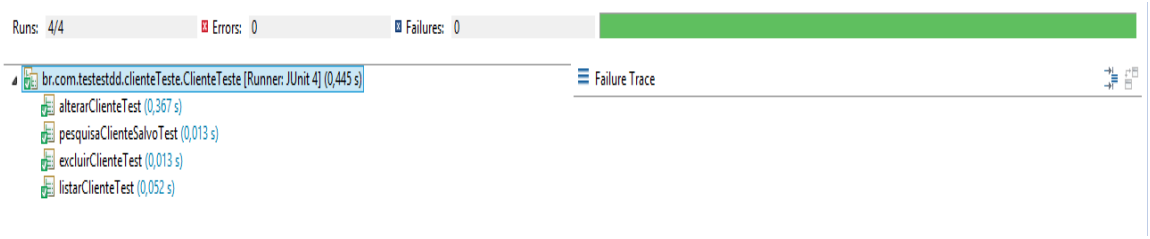

Figura 16 - Relatório do *JUNIT* com os resultados do teste

Nos passos seguintes, será abordado, conceito sobre a realização de testes em projetos que utilizam o modelo mvc (Modelo, visão e controle) de desenvolvimento, criando entidades seguindo a técnica de desenvolvimento guiado por testes, informando somente o que é necessário para fazer o teste passar.

Para realizar a mesma sequencia de realizada anteriormente, o quadro 15 exibe o trecho de código, no qual, se for executado, apresentará um erro, pois a classe ClienteRn ainda não existe.

```
@Test
publicvoid salvarTest() {
Cliente c1 = new Cliente();

ClienteRN clienteRN = new ClienteRN();

clienteRN.salvar(c1);

assertEquals(true, true);
}
```

Quadro 15 – Trecho de código para salvar cliente em mvc

Para que o teste exibido no quadro 15 passe, é necessário que a classe exibida no quadro 16 seja criada. *ClienteRN* é a classe de regra de negocio do cliente. Esta, recebe uma fabrica de *DAO* denominada *DAOFactory. criaClienteDAO(),* que também não existe, e deve ser criada.

```
publicclass ClienteRN {
  private ClienteDAO clienteDAO;
  public ClienteRN() {
   this.clienteDAO = DAOFactory.criaClienteDAO();
  }
  publicvoid salvar(Cliente c1) {
    this.clienteDAO.salvar(c1);
```

```
  }
  public List<Cliente> listar() {
    returnthis.clienteDAO.listar();
}
publicvoid excluir(Cliente cliente) {
this.clienteDAO.excluir(cliente);
}
public Cliente pesquisar(Stringstring) {
returnthis.clienteDAO.pesquisar(string);
}
publicvoid alterar(Cliente cliente) {
this.clienteDAO.alterar(cliente);
}
}
```

Quadro 16 – Códigoparacriação de ClienteRN

Para que a classe exibida no quadro 16 funcione, as classes exibidas nos quadros 17 e 18 devem ser criadas, sendo elas *ClienteDAO e DAOFactory*.

```
package br.com.testetdd.cliente;

import java.util.List;

publicinterface ClienteDAO {
publicvoid salvar(Cliente cliente);
publicList<Cliente> listar();
publicvoid excluir(Cliente cliente);
public Cliente pesquisar(Stringstring);
publicvoid alterar(Cliente cliente);
}
```

Quadro 17 – Código para criação de ClienteDAO

```
package br.com.testetdd.util;

import br.com.testetdd.cliente.ClienteDAO;

publicclass DAOFactory {
publicstatic ClienteDAO criaClienteDAO() {
   ClienteDAOHibernate clienteDAOHibernate = new ClienteDAOHibernate();
   clienteDAOHibernate.setSessao(HibernateUtil.getSession()
       .getCurrentSession());
      returnclienteDAOHibernate;
}
}
```

Quadro 18 – Código para criação de DAOFactory

A classe exibida no quadro 19, faz referência a classe ClienteDAOHibernate. Esta classe implementa todos os métodos existentes dentro da interface clienteDAO exibida no quadro 17.

```
publicclass ClienteDAOHibernate implements ClienteDAO {

privateSessionsessao;

@Override
publicvoid salvar(Cliente cliente) {
this.sessao.save(cliente);
}

public Session getSessao() {
       returnsessao;
}

publicvoid setSessao(Session sessao) {
 this.sessao = sessao;
}

@SuppressWarnings("unchecked")
@Override
publicList<Cliente> listar() {
Criteria lista = sessao.createCriteria(Cliente.class);
returnlista.list();
}

@Override
publicvoid excluir(Cliente cliente) {
this.sessao.delete(cliente);
}

@Override
public Cliente pesquisar(String string) {
 Query consultaNome = this.sessao.createQuery("from Cliente c where c.nome
like :nome");
consultaNome.setString("nome", "%" + string + "%");
 return (Cliente) consultaNome.uniqueResult();
}

@Override
publicvoid alterar(Cliente cliente) {
this.sessao.update(cliente);
}
}
```

Quadro 19 – Código para criação da classe ClienteDAOHibernate

Após a criação das classes necessárias para fazer um CRUD em mvc, os teste que foram criados anteriormente pode ser refatorados para que trabalhem com a nova estrutura de camadas. Como exibe o quadro 20.

```
publicclass ClienteTeste {
  privatestatic Session sessao;
  privatestatic Transaction transacao;

@BeforeClass
  publicstaticvoid abreConexao() {
  sessao = HibernateUtil.getSession().getCurrentSession();
  transacao = sessao.beginTransaction();
}

@AfterClass
publicstaticvoid fechaConexao() {
  try {
transacao.commit();
  } catch (Throwable e) {
   System.out.println("deu problema no commit : " + e.getMessage());
  } finally {
    try {
      if (sessao.isOpen()) {
            sessao.close();
      }
}catch (Exception e2) {
System.out.println("Deu erro no fechamanto da sessao"+ e2.getMessage());
}
}
}

@Before
publicvoid setup() {
 Cliente c1 = newCliente("7565650909", "teste1@mail", "Rua dos testes
1","Cliente 1", new Date(), 2000);
Cliente c2 = newCliente("75923509019", "teste2@mail", "Rua dos testes
2","Cliente 2", new Date(), 3000);
Cliente c3 = newCliente("82737650909", "teste3@mail", "Rua dos testes
3","Cliente 3", new Date(), 4000);
ClienteRN clienteRN = new ClienteRN();
clienteRN.salvar(c1);
clienteRN.salvar(c2);
clienteRN.salvar(c3);
}

@After
publicvoid limpaBanco() {
ClienteRN clienteRN = new ClienteRN();
List<Cliente> lista = clienteRN.listar();
for (Cliente cliente : lista) {
clienteRN.excluir(cliente);
}
}

@Test
publicvoidsalvarClienteTest() {
Cliente c1 = newCliente();
c1.setNome("Paulo Henrrique");
c1.setEndereco("São João Teste");
c1.setRenda(5000f);
c1.setCpf("44459456789");
```

```
ClienteRN clienteRN = new ClienteRN();
clienteRN.salvar(c1);
assertEquals(true, true);
}

@Test
publicvoid listarClienteTest() {
ClienteRN clienteRN = new ClienteRN();
List<Cliente>listaCli = clienteRN.listar();
assertEquals(3, listaCli.size());
}

@Test
publicvoid excluirClienteTest() {
ClienteRN clienteRN = new ClienteRN();
List<Cliente>listaCli= clienteRN.listar();
Cliente clienteExcluido = lista.get(0);
clienteRN.excluir(clienteExcluido);
listaCli= clienteRN.listar();
assertEquals(2, listaCli.size());
}
@Test
publicvoidpesquisarClienteTest() {
ClienteRNclienteRN = newClienteRN();
Cliente clientePesquisado = clienteRN.pesquisar("te 1");
assertEquals("teste1@mail", clientePesquisado.getEmail());
}

@Test
publicvoid alterarClienteTest() {
ClienteRN clienteRN = new ClienteRN();
Cliente clientePesquisado = clienteRN.pesquisar("te 1");
assertEquals("teste1@mail", clientePesquisado.getEmail());
clientePesquisado.setEndereco("Novo Enderecopa testes de cliente");
clienteRN.alterar(clientePesquisado);
Cliente clienteAlterado = clienteRN.pesquisar("te 1");
assertEquals("Novo Enderecopa testes de cliente",
clienteAlterado.getEndereco());
}
}
```

Quadro 20 – Código para execução de testes automatizados no modelo mvc

Após todas modificações realizadas, a execução do teste pela ferramenta pode ser executada. O resultado exibido pela figura 17, mostra a quantidade de testes realizados, o tempo de duração total para execução destes testes, a quantidade de erros encontrados durante os testes e a quantidade de falhas durante sua execução.

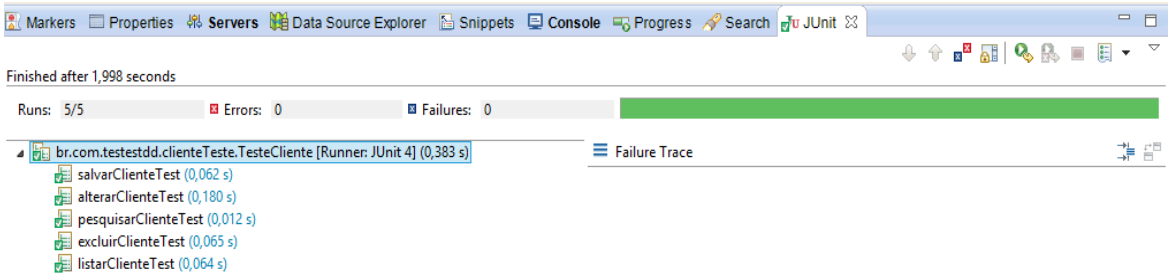

Figura 17 – Teste de Unidade após refatoração do código

A cor verde, na barra de progresso indica que o resultado esperado pelo teste foi encontrado. O mesmo passou com sucesso.

O ciclo pode continuar com qualquer outra entidade que deva ser criada para constituir uma estrutura de um sistema computacional. As regras de negócio para cada sistema pode mudar, porém a maneira como o sistema é desenvolvido deve seguir um fluxo, para que as alterações sejam realizadas no momento certo. Com a continuidade do TDD, qualquer alteração nas linhas do código necessária, um teste deve ser realizado para verificar se alguma funcionalidade foi quebrada.

Os testes de unidades descritos, servem para garantir que as funcionalidades estejam de acordo.

Seguir o TDD é uma das formas de realizar testes de unidade, porém não é a única maneira. Testes de softwares podem ser realizados de várias formas. O TDD, é um ciclo bem elaborado que auxilia no desenvolvimento de softwares partindo dos testes.

No caso de softwares que utilizam a web como ambiente de execução, uma ferramenta de testes como o *Selenium*, pode ser realizado para executar os testes de aceitação.

7 CONCLUSÃO

Conclui-se que, para que o software seja de qualidade, os testes “nunca” acabam, pois o mesmo, sai da produção e continua no cliente, após a implantação. Toda vez que executa uma funcionalidade específica, este é testado pelo cliente.

Testar o software, assim como outros produtos, não pode ser apenas uma etapa do desenvolvimento, mas sim, uma constante garantia de que aquilo que foi solicitado em reuniões e documentações seja executado corretamente cumprindo com as exigências documentais.

Para que este seja feito com qualidade, a automatização do processo de teste deve conter ferramentas que são preparadas para trabalhar nele. Estas ferramentas auxiliam de forma significativa no retorno do *feedback*, pois, os erros são encontrados mais rapidamente e corrigidos, diminuindo os custos e o tempo para a entrega do produto.

O desenvolvimento guiado por testes, auxilia em um melhor *design* para os desenvolvedores que neles trabalham. A técnica faz com que os mesmos pensem antes de implementar as funcionalidades da unidade da classe. Esta forma enfatiza a segurança dos métodos, pois estas classes só entram em execução após devidamente testadas e validadas seu funcionamento.

Comparado o desenvolvimento de software a um carro ou um avião, se o software fosse construído conforme os mesmos requisitos destes produtos, a condição de existência de *bugs* seria diminuída drasticamente. Pois a produção de um carro ou avião, possui modelos de produção bem diferentes da elaboração um software. O software, hoje é testado somente no final do projeto, identificando falhas que poderiam ser corrigidas no momento de produção. Já um carro ou avião possuem testes desde o início do projeto e estende-se por todas as etapas; modelagem, engenharia, construção, componentes, validação, analise e conclusão.

Os testes automatizados, passam para o computador as ações que poderiam ser realizadas pelo ser humano, porém com uma velocidade maior, que reduz o tempo proporcionando ao desenvolvedor mais tempo para se concentrar na regra de negócio que está sendo desenvolvida.

Para que os *bugs* do produto construído não cheguem até o cliente, uma bateria de testes devem ser planejadas, documentadas e executadas durante todo o processo de desenvolvimento. Desta maneira, é possível diminuir a probabilidade da conclusão das funcionalidades com erros. Para ter a validação desta estrutura de desenvolvimento, o desenvolvimento guiado por testes auxilia a equipe envolvida no processo em geral, a ter uma maior cobertura dos testes, com mais *feedback* de funcionalidades, diminuindo a ocorrência de bugs durante a construção.

Ver os testes falharem nesta técnica, é de extrema importância, para que o mesmo seja reavaliado fazendo com que passe.

Por ser uma técnica de desenvolvimento, o TDD pode ser adaptado e introduzido não somente no scrum, mas também, em outras metodologias ágeis. Para este propósito é necessário que os praticantes tenham experiência nas duas abordagem, principalmente no TDD, para que as estimativas do projeto não se perca durante a fase de desenvolvimento.

Como trabalhos futuros, é possível realizar avaliação de eficácia entre TDD e outras metodologias, com outras ferramentas. Também avaliar a produtividade em equipes heterogêneas, distribuídas, verificando o desempenho quando aplicado o TDD como técnica de desenvolvimento.

8 REFERÊNCIAS BIBLIOGRÁFICAS

Elias Nogueira (2010), Disponível em <http://www.testexpert.com.br/?q=node/1731> Pesquisado em 10/4/2013

Cristiano Caetano (2008), Disponível em <http://www.testexpert.com.br/?q=node/795>

Pesquisado em 10/4/2013

RIOS, Emerson e MOREIRA, Trayahu (2013). O que é teste de software Teste de Software Editora Alta books 2ª Edição

Fabio Martinho (2008), Disponível em http://www.testexpert.com.br/?q=node/669

Pesquisado em 10/4/2013

Inmetro, Disponível em http://www.inmetro.gov.br/qualidade/> Pesquisado em 10/5/2013

Arilo Claudio Dias (2008), Disponível em <http://www.devmedia.com.br/space.asp?id=185582>Pesquisado em 11/5/2013

Cazuza Neto (2012), Disponível em, <http://www.devmedia.com.br/conhecendo-o-scrum/25744> Pesquisado em 16/5/2013

Marcelo F Andrade (2010), disponível em <http://www.slideshare.net/mfandrade/testes-de-software-automatizados> Pesquisado em 16/5/2013.

Camilo Ribeiro (2010), Disponível em <http://www.slideshare.net/camiloribeiro/teste-de-software-introduo-qualidade-7784496> Pesquisado em 19/5/2013.

Wagner R. Santos (2009), Disponível em http://www.infoq.com/br/articles/levison-TDD-adoption-strategy> Pesquisado em 19/5/2013.

Jefferson Carlos Martins e Helbert Luiz Tschannerl (2013), Disponível em <http://www.batebyte.pr.gov.br/modules/conteudo/conteudo.php?conteudo=1100>Pesquisado em 24/5/2013.

RAMOS, Ricardo Argenton (2010), Teste de Software Orientado a Objeto – Universidade Federal do Vale do São Francisco.

Ronaldo Caldas da Silva: http://javaelinux.wordpress.com/ Pesquisado em 19/8/2013.

ISO. ISO/IEC 29119 Software Testing: The new international "software testing standard. 2009".

Mauricio Aniche Test-Driven Development: Teste e Design no Mundo Real 2012

SCHWABER, Ken. The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process, Washington, 2007.

Gustavo Alves, Juvenal Feitosa, Felipe Furtado (2011) disponível em: <http://www.devmedia.com.br/websys.4/webreader.asp?cat=6&revista=javamagazine_105#a-4704>Pesquisado em 24/5/2013.

ASTELS, David. Test-Driven Development: A Practical Guide 5° ed. New Jersey: Pearson Education, 2003.

SANCHEZ, Ivan (2007), Disponível em: <http://dojofloripa.wordpress.com/2007/02/07/scrum-em-2-minutos> Pesquisado em 26/5/2013.

Pitang Agili 2012 – Disponível em: http://www.pitang.com/blogs/agil/index.php/2012/04/test-driven-development-tdd-nao-e-teste/

BORGES, Eduardo N, Conceitos e Benefícios do Test Driven Development, Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)

Nayara C Ferreira, (2013), Disponível em: <http://www.trabalhosfeitos.com/ensaios/Tdd-Desenvolvimento-Dirigido-Por-Teste/756316.html>Pesquisado em 27/5/2013.

Ronni Oliveirahttp://gerenciandoprojetosdesoftware.blogspot.com.br/2010/12/teste-de-software-orientado-objeto.html

Kent Beck. Tdd 10 years later. http://junit.sourceforge.net/doc/testinfected/

testing.htm.