FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ALLAN RETT FERREIRA

UTILIZAÇÃO DA METODOLOGIA TDD PARA DESENVOLVIMENTO DE SOFTWARE

MARÍLIA

2011

ALLAN RETT FERREIRA

UTILIZAÇÃO DA METODOLOGIA TDD PARA DESENVOLVIMENTO DE SOFTWARE

Trabalho de Curso apresentado ao Curso de Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Dr. Elvis Fusco

MARÍLIA

2011

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Allan Rett Ferreira

UTILIZAÇÃO DA METODOLOGIA TDD PARA DESENVOLVIMENTO DE SOTWARE

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 5,0 (Cinco)

Orientador: Elvis Fusco

1º. Examinador: Rodolfo Barros Chiaramonte

2º. Examinador: Fabio Lucio Meira

Marília, 29 de novembro de 2011.

Dedicatória

Eu dedico esse trabalho primeiramente a Deus, porque sem ele nada disso seria possível. Dedico também aos meus pais pela força e apoio dado, aos meus amigos em geral que me ajudaram para isso, também ao meu orientador que me guiou e me apoiou para que esse trabalho pudesse ser feito, e por fim, a todos os professores pelo ensino oferecido e pela ajuda em toda essa minha caminhada.

Agradecimentos

Agradeço a Deus pela vida, pela saúde e por ter me dados forças em todos os momentos, quando mais precisei ou até mesmo quando pensei em desistir.

Agradeço aos meus pais pelo apoio que me deram constantemente, me incentivando em todos os momentos para que eu não desistisse e tentasse até o último minuto.

Agradeço ao meu orientador também por ter me dado essa oportunidade, por ter me guiado, em geral, foi um grande aprendizado para mim.

Agradeço também a todos os professores pelo ensino oferecido.

Agradeço aos meus colegas de trabalho pela ajuda nas pesquisas sobre o tema, aumentando assim meu conhecimento sobre o tema.

E por fim agradeço a todos meus amigos e colegas que me apoiaram e a todos que de alguma maneira ajudaram para que esse trabalho pudesse ser feito.

Ferreira, Allan Rett, **Utilização da Metodologia TDD para desenvolvimento de software.** 2011. Trabalho de Curso Bacharelado em Ciência da Computação – Centro Universitário Eurípides de Marília, Fundação de Ensino “Eurípides Soares da Rocha”, Marília, 2011.

RESUMO

Este trabalho apresenta o estudo de viabilidade de utilização da metodologia Test Driven Development para desenvolvimento de software. O objetivo desse trabalho é demonstrar que utilizando a metodologia citada à qualidade do software final será muito maior do que um software desenvolvido sem a utilização da metodologia, diminuindo assim custos que o desenvolvedor teria com suporte, correções de bugs, erros, etc. O Test Driven Development tem como objetivo aumentar a qualidade do desenvolvimento de software utilizando como base testes unitários, permitindo ao desenvolvedor testar cada parte de seu software de forma independente.

Palavras-chave: Test Driven Development, TDD, Software.

Ferreira, Allan Rett, **Utilização da metodologia TDD para desenvolvimento de software.** 2011. Trabalho de Curso Bacharelado em Ciência da Computação – Centro Universitário Eurípides de Marília, Fundação de Ensino “Eurípides Soares da Rocha”, Marília, 2011.

ABSTRACT

This work presents the study of feasibility of using the Test Driven Development methodology for software development. The aim of this work is to demonstrate that using the methodology cited the quality of the final software will be much larger than a software developed without the use of the methodology, thereby lowering costs that the developer would have to support, bug fixes, errors, etc. The Test Driven Development aims to increase the quality of software development using unit tests as a basis, allowing the developer to test each of its software independently.

Keywords: Test Driven Development, TDD, Software.

Lista de Ilustrações

Lista de Tabelas

Lista de abreviaturas e siglas

ACL: Access Control List

API: Application Programming Interface

HTML: Hyper Text Markup Language

HTTP: Hyper Text Transfer Protocol

IDE: Integrated Development Environment

JS: JavaScript

MVC: Model-View-Controller

PDO: PHP Data Objects

PHP: Hypertext Processor

SGBD: Sistema de Gerenciamento de Banco de Dados

TDD: Test Driven Development

URL: Universal Resource Locator

ZF: Zend Framework

Sumário

INTRODUÇÃO

Nas últimas décadas o software tem estado cada vez mais presente no cotidiano das pessoas. Ele é encontrado nos aparelhos de telefonia celular, nos fornos de microondas, nos carros, nas máquinas digitais e até nos porta-retratos. Em um estudo (REED, 2000), foi constatado que se alguns sistemas de uso global deixarem de funcionar, aproximadamente 40% da população mundial sofrerão as conseqüências (ABRAN ET AL., 2004).

O desenvolvimento de sistemas de software envolve uma série de atividades de produção em que as oportunidades de inserção de falhas humanas são enormes (PRESSMAM, 2006). Uma das maneiras de garantir a qualidade de determinado software é a aplicação da técnica de teste de software, a qual engloba 40% do esforço total de uma instituição na confecção de um software (PRESSMAM, 2006).

Em contrapartida à grande importância que o software tem tido no cenário mundial, estudos comprovam que a grande maioria dos projetos de software não atende aos objetivos traçados. De acordo com esses estudos, isso é decorrente da falta de processos adequados nas organizações em que eles são desenvolvidos.

Segundo Borges (2006), atualmente, as falhas de software são grandes responsáveis por custos e tempo no processo de desenvolvimento de software. Embora não seja possível remover todos os erros existentes nas aplicações, é possível reduzir consideravelmente o número dos mesmos utilizando uma infra-estrutura de testes mais elaborada, que permita identificar e remover defeitos mais cedo e de forma mais eficaz.

Estes defeitos podem resultar de diversas causas como erros de conhecimento, comunicação, análise, transcrição, codificação, etc. Existem essencialmente três formas de tratar falhas de software:

1. Evitar falhas (*fault-avoidance*): com atividades apropriadas de especificação, projeto, implementação e manutenção sempre visando evitar falhas em primeiro lugar. Inclui o uso de métodos de construção de software avançados, métodos formais e reuso de blocos de software confiáveis.
2. Eliminar falhas (*fault-elimination*): compensação analítica de erros cometidos durante a especificação, projeto e implementação. Inclui verificação, validação e teste.
3. Tolerar falhas (*fault-tolerance*): compensação em tempo real de problemas residuais como mudanças fora da especificação no ambiente operacional, erros de usuário, etc.

Uma pesquisa publicada pelo *National Institute of Standards and Technology* (NIST) e pelo Departamento de Comércio dos Estados Unidos revela que os erros de software custam cerca de 60 bilhões de dólares à economia norte americana a cada ano (NIST, 2002). Portanto, faz-se necessário o estudo de técnicas orientadas a testes, pois este pode proporcionar uma economia considerável para empresas de desenvolvimento e aumentar a qualidade do software produzido.

Visando esta economia e aumento da qualidade, o *Test Driven Development* (TDD) foi criado para antecipar a identificação e correção de falhas durante o desenvolvimento do software.

Atualmente o fluxo do desenvolvimento de um software sem a utilização do TDD é exibido na figura 1.

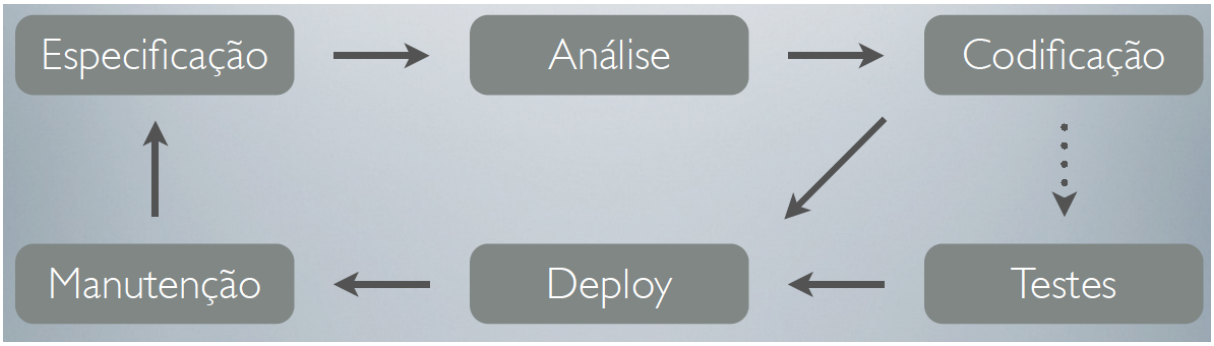

Figura 1 - Fluxo desenvolvimento de software sem TDD (DOHMNS, 2010)

O fluxo demonstrado na figura 1 oferece diversos riscos quanto à qualidade do software desenvolvido, pois erros, bugs podem aparecer de forma mais freqüente, além do desenvolvimento de código não determinado na fase de análise, pois o desenvolvedor não estará tão atento a especificação de cada parte do software como se estivesse utilizando o TDD.

Corrigir um erro ou introduzir mudanças no software possui um custo muito alto (PRESSMANN, 2006). A cada etapa de desenvolvimento esse custo aumenta consideravelmente, pois uma mudança efetuada em uma parte do software pode inserir um erro em uma outra parte do mesmo. Sem a utilização de testes a freqüência com que isso ocorre é muito maior.

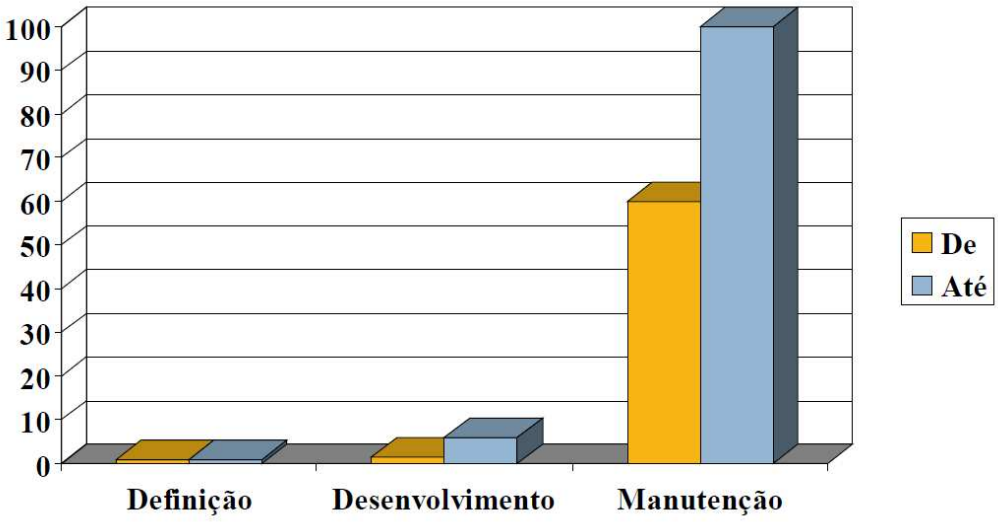

Figura 2 - Custos de introdução de mudanças em um software (PRESSMAN, 2006)

Müller e Hagner (2002) conduziram um experimento estruturado comparando TDD com programação tradicional. Este experimento mediu a eficiência do TDD em termos do

time de desenvolvimento, qualidade do código resultante e nível de entendimento do mesmo. Dada a especificação e a declaração de alguns métodos, os times deveriam completar o corpo destes métodos. O grupo que desenvolveu com TDD escreveu os casos de teste antes de começar a implementação (o TDD não é a única metodologia que prega escrever os testes antes da codificação). Já o grupo da programação tradicional escreveu os testes depois de completar o código.

A experiência ocorreu em duas fases, uma de implementação e outra de aceitação de teste. Ambos os grupos tiveram oportunidade de corrigir o código após a implementação. Apesar do tempo total de desenvolvimento dos métodos ter sido o mesmo para as duas equipes, o código do grupo TDD teve menos erros significantes quando reusado. Baseados nesse fato, os pesquisadores concluíram que a abordagem *test-first* aumenta substancialmente a qualidade do software e proporciona maior entendimento do código.

Mugridge (2003) faz uma analogia entre o método científico e o TDD. O método científico é um modelo de como desenvolver teorias sobre quaisquer fenômenos.

Primeiramente é definida uma hipótese. Após é projetado um experimento que comprove esta hipótese. Então, o experimento é executado e, baseado em seus resultados, são redefinidas as hipóteses até que seja comprovada a teoria. Esses passos são apresentados na figura 3. Pode-se perceber a grande semelhança com as iterações no processo do TDD. Escolher uma hipótese que comprove uma teoria é um problema bastante difícil.

A primeira hipótese sugerida por um cientista pode ser extremamente subjetiva e não muito clara. Tal hipótese deve passar por refinamentos sucessivos até que seja definida uma teoria. No TDD, os primeiros testes são difíceis de escolher, mas ajudam a clarificar e especificar melhor o problema a ser resolvido. Portanto, o TDD é uma abordagem de desenvolvimento que segue as mesmas idéias do método científico, sendo assim, uma técnica adequada para implementar grandes soluções que tendem a mudar com o passar do tempo. Além disso, é um método que permite o melhor entendimento do sistema.

OBJETIVOS

O presente trabalho apresenta um estudo da utilização da metodologia *Test Driven*

Development (TDD) para desenvolvimento de softwares, demonstrando como a metodologia ajuda no desenvolvimento de softwares com maior qualidade.

Para isso será realizado o desenvolvimento de uma aplicação utilizando a linguagem PHP em conjunto com o Zend Framework (ZF) e o framework de testes PHPUnit.

O PHPUnit em conjunto com o XDebug (uma extensão que auxilia no *debug* de *scripts* PHP) fornece várias informações sobre o andamento do projeto, sobre a porcentagem de testes realizados no código fonte da aplicação, com isso é possível determinar quanto o código do software está testado e aprovado segundo as especificações do projeto.

O projeto desenvolvido foi um breve cadastro de papéis, recursos e privilégios utilizados pelo ZF para validação de autorização de usuários em sistemas protegidos, utilizando o TDD para testar os métodos da classe Zend ACL (Access Control List) o projeto visa demonstrar como a metodologia TDD aumenta a qualidade final de um software.

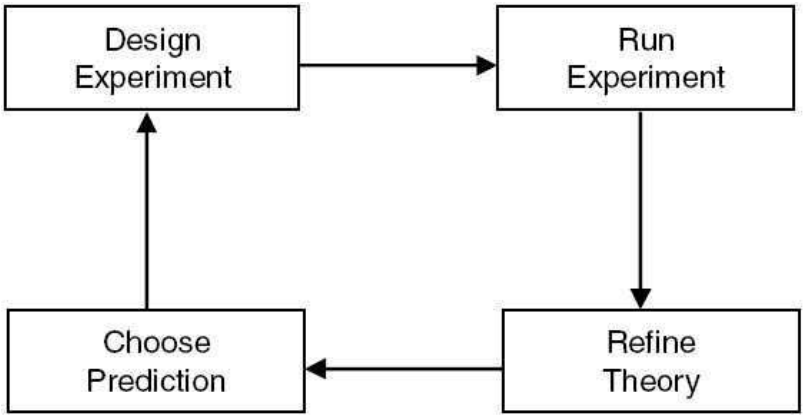

Figura 3 - Passos do método científico (BORGES, 2006)

CAPÍTULO 1 – TESTE DE SOFTWARE

Teste de software é um processo de execução de um programa com a intenção de encontrar erros e buscando proporcionar assim uma melhor qualidade ao software. Serve, portanto, como uma fonte importante de *feedback*, proporcionando uma larga interação com os diversos participantes do projeto (PEDRYCZ e PETERS, 2001). Também podem ser definidos como uma série bem planejada de passos, que resultam na construção bem sucedida de um software (PRESSMAN, 2006).

Os testes de software tornaram-se indispensáveis para detectar os defeitos que escapam das revisões e para avaliar o grau de qualidade de um produto juntamente com seus componentes (PÁDUA, 2003).

O IEEE Software Knowledge Body define o teste como sendo "a atividade executada para avaliar e melhorar a qualidade de um produto, através da identificação dos defeitos e dos problemas" (ABRAN ET AL., 2004). Mais particularmente para o domínio da computação, essa mesma organização vê que o teste de software consiste na verificação dinâmica do comportamento de um programa, para um conjunto finito de casos de teste, comparando-o com seu comportamento esperado (ABRAN ET AL., 2004). Testar todos os casos para um sistema real é impossível ou, ao menos, impraticável em um tempo finito. Além disso, o oraculo[1], que consiste na tomada da decisão se o comportamento de um sistema é correto ou não, também aparece nessa definição de teste de software.

Desenvolver software é uma atividade complexa, na qual muitos erros podem ser introduzidos. A fim de minimizar a ocorrência de erros associados ao processo de desenvolvimento de software, atividades de garantia de qualidade são adicionadas a esse

[1] Um oráculo pode ser visto como a entidade que analisa o resultado do teste, comparando-o com o resultado esperado, para saber se o programa que está sendo testado se comporta corretamente. Portanto, ele produz um veredicto sobre um teste, dizendo se este "passou" ou "falhou" (ABRAN ET AL., 2004).

processo, como VV&T (Verificação, Validação e Teste). Dentre as atividades de verificação e validação, uma das técnicas mais utilizadas é o teste (BARBOSA, E. F. ET AL., 2000).

Myers (2004) afirma que um teste bem executado é aquele que encontra erros. O teste, portanto, tem como objetivo de executar um programa a fim de encontrar suas anomalias e defeitos. Ele pode ser visto como uma atividade “destrutiva”, já que no teste procura-se encontrar as falhas de um programa, enquanto que na fase de desenvolvimento procura-se construir um programa correto.

Principais objetivos dos testes:

- Foco na prevenção de erros;
- Descobrir sintomas causados por erros;
- Fornecer diagnósticos para que os erros sejam facilmente corrigidos;
- Mostrar que o software tem erro;
- Segurança.

1.1 FASES DE TESTE

O teste de software é geralmente efetuado em diferentes fases (níveis) ao longo do processo de desenvolvimento e manutenção. Os alvos dos testes podem variar: um único módulo, um grupo de módulos ou um sistema completo. De maneira geral, três grandes fases de teste podem ser identificadas (ABRAN ET AL., 2004).

- **Teste de unidade ou teste unitário:** é a fase onde são testados os componentes mínimos de um sistema, como por exemplo, uma classe ou um método, considerando-se a abordagem orientada a objetos. O objetivo básico é encontrar falhas em pequenas partes do sistema (unidades), que funcionam

independentes do todo (o sistema completo). Este tipo de teste concentra-se em erros de lógica e de implementação de cada módulo do software separadamente. Nos testes de unidade pode-se verificar se o sistema atende aos requisitos especificados no escopo do projeto (as necessidades do cliente) (BARBOSA, E. F. ET AL., 2000).

- **Teste de integração:** é o processo de testar a interação entre os componentes de software (ABRAN ET AL., 2004). Seu objetivo básico é encontrar falhas provenientes dessa interação, como por exemplo, erros na transmissão de dados entre dois componentes. Ele visa, portanto a descobrir erros associados às interfaces entre os módulos de um software (BARBOSA, E. F. ET AL., 2000).

- **Teste de sistema:** este teste preocupa-se com o funcionamento do sistema como um todo (ABRAN ET AL., 2004). Nesta fase, o software desenvolvido é combinado com outros elementos do sistema (hardware, banco de dados, etc). Este tipo de teste visa erros de funções e características de desempenho que não estejam de acordo com as especificações do software separadamente (BARBOSA, E. F. ET AL., 2000).

Geralmente, inicia-se o processo de teste pelos testes de unidade. Após cada unidade do sistema ter sido validada, efetuam-se os testes de integração. Finalmente, para testar o software como um todo e assegurar que ele atende aos requisitos especificados, efetua-se o teste de sistema.

1.1 TESTES UNITÁRIOS

Segundo Bergmann (2006), teste unitário é um procedimento automático usado para validar se uma pequena parte do código funciona corretamente.

A metodologia ágil de gestão de projeto de software XP, de Beck (1999), prega como uma de suas principais práticas que os testes unitários devem ser escritos antes do módulo (unidades) a ser testada. Considerando-se uma abordagem dirigida a objetos, é recomendado, portanto, escrever os testes de uma classe antes mesmo do código que a implementa.

Alguns objetivos do teste unitário são:

- Testes para pequenos trechos de código (unidades);
- Verifica se o comportamento de classes e funções é o esperado;
- Em caso de erros, exceções são lançadas;
- Não interfere com o seu código-fonte.

Algumas das vantagens do teste unitário são:

- Os testes podem ser automatizados;
- São executados continuamente durante o ciclo de desenvolvimento;
- Detectam falhas tanto de digitação e lógica e também comportamentos inesperados;
- Escalam conforme o desenvolvimento;
- Correção de erros;
- Refatoração.

Os testes devem ser automáticos e executados em intervalos regulares de validação e verificação sobre o correto funcionamento das unidades testadas. Idealmente, cada teste deve ser independente do outro. A unidade a ser testada é um pequeno pedaço de código de software. Hoje em programação orientada a objeto, esses pequenos pedaços de código são geralmente métodos individuais de uma determinada classe. Teste de unidade nos dá um caminho para testar nossas implementações de design e comportamento nas classes que serão desenvolvidas. Hoje pode-se afirmar que o teste de unidade é uma parte fundamental do desenvolvimento de software de qualidade moderno (PRESSMAN, 2006).

1.2 BENEFÍCIOS DO TESTE DE UNIDADE

Teste de unidade tem três benefícios principais:

- Faz você pensar sobre a sua aplicação e sobre a rota de implementação que você escolheu / está prestes a escolher.
- Ele dá uma visão imediata sobre o bom funcionamento do sistema e facilita a localização de um defeito no sistema.
- Facilita o refactoring do código existente. Que é igualmente importante para garantir a longevidade de um aplicativo.

Quando algum erro danifica o aplicativo, devido a fatores externos ou por alterações feitas no código, testes de unidade, irão rapidamente identificar qual é o erro que foi introduzido no código. O que lhe permite localizar rapidamente e corrigir o mesmo. No final acelera o processo de desenvolvimento do software, pois com os testes sendo executados corretamente, o desenvolvedor vai sentir mais confiança em seu código e nas suas mudanças futuras. Refletindo essa confiança aos clientes, gestores e outros desenvolvedores.

CAPÍTULO 2 – TEST DRIVEN DEVELOPMENT (TDD)

O TDD (*Test Driven Development* ou desenvolvimento orientado a testes) é uma metodologia de desenvolvimento de software ágil derivado do método *Extreme Programming* (XP) (Beck 2000) e do *Agile Manifesto* (Agile Alliance 2000). É baseado também em técnicas de desenvolvimento utilizadas há décadas (Gelperin e Hetzel 1987) (Larman e Basili 2003).

O TDD é uma metodologia usada na fase de implementação do software onde os desenvolvedores usam testes para guiar o projeto durante o desenvolvimento. Esta prática envolve a implementação de um sistema começando pelos casos de teste de um objeto (BORGES, 2006), é organizada em torno de um conjunto de práticas e valores que atuam para assegurar um alto retorno do investimento efetuado pelo cliente.

A metodologia TDD consiste em pequenas iterações onde novos casos de testes são escritos contemplando uma nova funcionalidade ou melhoria e, somente depois, o código necessário e suficiente para passar estes testes é implementado. Logo após esta fase, o software é refatorado para contemplar as mudanças de forma que os testes continuem passando (FEITOSA, 2007).

Os desenvolvedores usam testes para guiar o projeto do sistema durante o desenvolvimento. Eles automatizam estes testes para que sejam executados repetidamente. Através dos resultados (falhas ou sucessos) julgam o progresso do desenvolvimento. Os programadores fazem continuamente pequenas decisões aumentando as funcionalidades do software a uma taxa relativamente constante. Todos estes casos de teste devem ser realizados com sucesso antes do novo código ser considerado totalmente implementado.

O TDD pode ser visto como um conjunto de iterações realizadas para completar uma tarefa (Beck 2000). Cada iteração envolve os seguintes passos:

- Escolher a área do projeto ou requisitos da tarefa para melhor orientar o desenvolvimento.
- Projetar um teste concreto da maneira mais simples possível para que o resultado obtido seja o desejado na especificação do projeto. Checar se o teste falha.

- Alterar o sistema para satisfazer este teste e outros possíveis testes.
- Possivelmente refatorar o sistema para remover redundância, sem quebrar nenhum teste.

Uma importante regra no TDD é: *"If you can't write a test for what you are about to code, then you shouldn't even be thinking about coding"* ("se você não consegue escrever um teste para o que você está pensando em codificar, então você não deveria estar pensando em codificar") (Chaplin 2001). Outra regra no TDD diz que quando um defeito de software é encontrado, casos de teste de unidade são adicionados ao pacote de teste antes de corrigir o código.

TDD utiliza uma das técnicas do XP chamada *refactoring* (Beck 2000) para conseguir a compreensão do código e gerenciar a complexidade do mesmo. Como um grande programa ou sistema é continuamente modificado, ele torna-se muito complexo, sendo extremamente necessária a facilidade de manutenção (Lehman e Belady 1985). Esta técnica é essencial para reduzir a complexidade do software e torná-lo manutenível.

A pequena granularidade do ciclo *test-then-code* oferece um *feedback* contínuo ao programador. Falhas são identificadas mais rapidamente, enquanto o novo código é adicionado ao sistema. Assim, o tempo de depuração diminui compensado pelo tempo de escrita e execução dos casos de teste. Alguns estudos indicam que cerca de 50% das tarefas no processo de manutenção de software são envolvidas no processo de entendimento do código (Corbi, 1989). A abordagem TDD ajuda na compreensão do programa porque os casos de teste e próprio código explicam melhor o funcionamento do sistema. Entretanto, esta prática permite somente o entendimento de uma parte do software. Para a compreensão da sua totalidade, é necessário que se faça uso de várias abstrações.

Os resultados de métodos ou funções são testados automaticamente. Estes valores são comparados aos resultados esperados ainda na etapa de codificação. Já na etapa de manutenção, as unidades de teste criadas anteriormente permitem avaliar mais facilmente

novos defeitos que podem ter sido inseridos no software. Este benefício é essencial para o desenvolvimento e controle de novos releases, reduzindo a injeção de novas falhas no sistema.

De acordo com Beck (2000), um método ágil é comparável ao ato de dirigir um carro: você deve observar a estrada e fazer correções contínuas para se manter no caminho.

Neste contexto onde a agilidade é fundamental, o testador seria aquele que ajuda o motorista a chegar com segurança ao seu destino, impedindo que sejam feitas conversões incorretas durante o percurso, evitando que o motorista se perca e fazendo com que ele pare e peça instruções quando necessário. Neste ambiente, o TDD se destaca, como sendo uma abordagem evolutiva na qual o desenvolvedor escreve o teste antes de escrever o código funcional necessário para satisfazer aquele teste (Baumeister, 2002).

Segundo Marrero (2005), o objetivo principal do TDD é especificação e não validação. Em outras palavras, é uma forma de refletir sobre a modelagem antes de escrever código funcional. Já segundo Baumeister (2002), desenvolvimento dirigido por testes é uma técnica de programação onde o principal objetivo é escrever código funcional limpo a partir de um teste que tenha falhado. Como efeito colateral, obtém-se um código fonte bem testado.

Após escrever os casos de teste, que devem especificar apenas uma pequena parte da funcionalidade, o desenvolvedor deve implementar o código necessário apenas para passar pelo teste. Feito isto, fica a cargo do desenvolvedor realizar o *refactor* do código, garantindo que o código continue simples e consiga satisfazer determinada funcionalidade (BECK 2000).

O ciclo, de uma visão macro, poderia ser visto como: escrever poucos casos de teste, implementar o código; escrever poucos casos de teste, implementar o código, e assim por diante.

TDD é primariamente uma técnica de programação que garante que o código de um sistema esteja inteiramente testado de forma unitária (Dasgupta, Sanjoy, Papadimitriou, Christos e Vazirani, Umesh 2006). Entretanto, há mais teste do que isto, pois ainda é necessário que sejam realizados os tradicionais testes, como testes funcionais, testes de usuário, testes de integração, entre outros. A maioria destes testes podem ser realizados tardiamente ao sistema (Dasgupta, Sanjoy, Papadimitriou, Christos e Vazirani, Umesh 2006).

Com a realização de testes tradicionais, serão encontrados mais defeitos. Isto ocorre com o TDD; quando um teste falha, o desenvolvedor sabe que necessita resolver o problema (Dasgupta, Sanjoy, Papadimitriou, Christos e Vazirani, Umesh 2006). TDD aumenta a credibilidade de que o sistema realmente funciona, e está de acordo com os requerimentos propostos (BECK, 2000).

Um interessante efeito do TDD, é que é possível cobrir 100% do sistema com testes, pois cada linha de código é testada.

Estudos comprovam que sistemas desenvolvidos utilizando-se de TDD resultam em um código melhor e com menos defeitos do que sistemas que utilizaram somente testes tradicionais (L. WILLIAMS) (JONES, 2004).

Existem várias questões que são levantadas da adoção do TDD em um projeto. Uma delas é em relação ao que não deve ser testado. Kent Beck (2000) diz que devem ser testados condições, operações e polimorfismos, porém, somente aqueles que o próprio desenvolvedor escreveu.

Há três pontos que devem ser observados para execução de um teste:

i) Se forem necessárias muitas linhas de código criando objetos para uma simples asserção, então há algo de errado;

ii) Se não é possível encontrar facilmente um lugar comum para o código de inicialização, então existem muitos objetos fortemente acoplados.

iii) Testes que quebram inesperadamente sugerem que uma parte da aplicação está afetando outra parte. É necessário projetar até que esta distância seja eliminada, ou quebrando esta conexão ou trazendo as duas partes juntas.

Uma vantagem significativa do TDD é que ele possibilita ao desenvolvedor dar pequenos e controlados passos durante a codificação do software (Baumeister, 2002). Esta prática é mais aconselhável do que escrever grandes quantidades de código de uma só vez.

Alguns motivos para a adoção do TDD, segundo Kaufmann (2003), são:

i) O desenvolvimento parte do princípio dos objetivos, após isto, é pensado nas possíveis soluções;

ii) O entendimento do sistema pode ser feito a partir da leitura dos testes;

iii) Não é desenvolvido código desnecessário;

iv) Não existe código sem teste;

v) Uma vez um teste funcionando, é sabido que ele irá funcionar sempre, servindo também, como teste de regressão;

vi) Os testes permitem que seja feito *refactor*, pois eles garantirão que as mudanças não alteram o funcionamento do sistema.

Segundo Mugridge (2003), como conseqüências do uso do TDD, têm-se que o código é escrito de maneira testável isoladamente, pois geralmente, códigos escritos da maneira tradicional, possuem alto acoplamento. Os testes atuam como uma documentação do sistema, pois eles seriam os primeiros clientes a usarem as classes desenvolvidas, mostrando ao

desenvolvedor, o que é necessário fazer. Outros benefícios provindos da adoção do modelo de desenvolvimento TDD foram abordados por Müller (2002), no qual, foi feita uma análise do retorno de investimento em se utilizar TDD ao invés de modelos convencionais de desenvolvimento. Nele a qualidade final do código, era superior ao modelo convencional, pois não seria necessário perder uma grande fatia de tempo no final do projeto para a correção de defeitos, pois estes eram capturados e corrigidos ao longo do desenvolvimento do sistema.

A figura 4 mostra de forma clara a diferença existente entre os dois modelos.

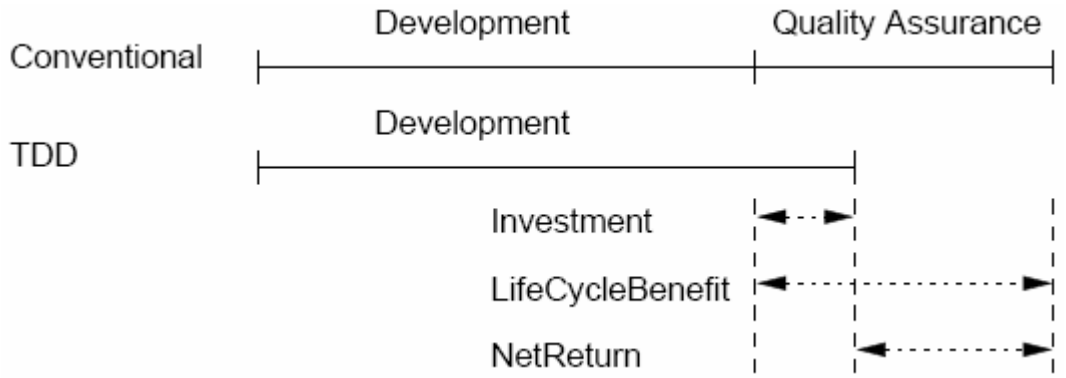

Figura 4 - Comparação TDD x Modelo convencional de desenvolvimento (GASPARETO, 2006)

2.1 MOCKS

Mocks são utilizados com o propósito de escrever testes como se o ambiente possuísse tudo o que realmente é necessário para a realização do teste (FREEMAN, 2004). Este processo apresenta ao desenvolvedor o que o ambiente deve prover para o desenvolvimento do sistema.

Ao se testar partes isoladas do sistema, como apenas um objeto, por exemplo, o programador deve levar em consideração a interação deste objeto em questão com outros objetos. A figura 5 apresenta como é realizado um teste usando-se mocks. No caso, têm-se um objeto A, que necessita interagir com um serviço que é disponibilizado pelo objeto S, mas

como ainda não se tem uma implementação de S, ou este será implementado por terceiros, realiza-se o mock do objeto S, forçando assim, o uso de interfaces para o desenvolvimento.

Um exemplo de mock pode ser dado como a consulta a um web service, para que o sistema não fique a todo momento requisitando informações do web service, é criado um mock (uma função, um dado, informações, etc) com o resultado requisitado no web service, assim o sistema terá o resultado esperado do web service sem efetuar uma requisição ao mesmo, as informações serão retiradas do mock.

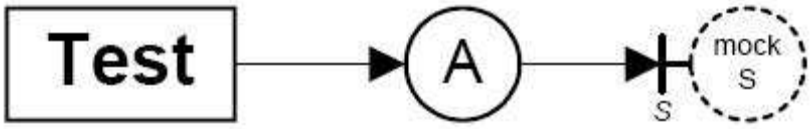

Figura 5 - Uso de mock (GASPARETO, 2006)

2.2 PADRÕES PARA O TDD

Existem inúmeros padrões para várias atividades na área de informática que são utilizados na construção de sistemas. Para a disciplina de TDD não é diferente, e existem alguns padrões, que no âmbito geral, tornam-se mais dicas sobre o que testar em um sistema.

- **Testes isolados** - A execução de um teste não pode afetar outro teste. Segundo Kent Beck (2000), os testes devem ser desenvolvidos de maneira que sejam rápidos de serem executados, assim, eles podem ser rodados a todo o momento, não necessitando de um momento específico de execução de testes. Dessa forma, os testes isolados servem para que uma execução de um determinado teste, não interfira na execução do teste seguinte, não necessitando assim, iniciar e parar o sistema assim que cada teste é executado.

- **Lista de testes** - De acordo com Kent Beck (2000), antes de iniciar a codificação, deve ser feita uma lista com todos os testes que o desenvolvedor acredita que será necessário escrever. Assim, quando o desenvolvedor iniciar a programar, ele terá um guia do que é necessário fazer, e não corre o risco de esquecer algo.

- **Teste primeiro** - Os testes devem ser escritos antes do código que será testado (BECK, 2000). Os testes servem para dar um *feedback* rápido do desenvolvimento do sistema. Assim que o tempo passa, mais estressado fica o desenvolvedor, e, portanto, caso os testes sejam deixados para o final, o desenvolvedor irá aumentar o nível de estresse, diminuindo a quantidade de testes no sistema, e impactando na qualidade do produto final.

- **Dados de Teste** - Segundo Kent Beck (2000), devem ser utilizados dados que tornem os testes fáceis de ler e seguir. Caso o sistema tenha múltiplas entradas, então, devem existir testes que abranjam estas entradas. Porém, é desnecessário ter uma lista com dez entradas, se com apenas três, cobrem-se todas as alternativas (BECK, 2000).

- **Dados evidentes** - Devem ser incluídos nos testes os dados esperados e o resultado atual, e tentar exibir um relacionamento aparente entre eles, pois os testes não são escritos apenas para o computador, e sim, para que futuros desenvolvedores possam entender o código como sendo uma parte da documentação do sistema (BECK 2000).

2.3 DISCIPLINAS RELACIONADAS AO XP

Mesmo TDD sendo uma disciplina do XP (BECK, 2000), esta não se encontra isolada no universo do *Extreme Programing*. A seguir uma lista de disciplinas que são complementares e/ou completadas usando-se o TDD:

- **Programação em pares** – os testes escritos no TDD são excelentes peças de conversação quando se está trabalhando de forma pareada. A programação pareada melhora o TDD no momento em que o programador que está codificando o código está cansado, assim, o outro se encarregaria de codificar (BECK, 2000);

- **Integração Contínua** – testes são um excelente recurso para esta disciplina, permitindo que sempre seja feita uma integração. Os ciclos são definidos em períodos curtos, algo entre quinze e trinta minutos, ao invés de uma ou duas horas de codificação (BECK, 2000);

- **Design Simples** – codificando apenas o necessário para os testes, e removendo código duplicado, automaticamente obtém-se um design adaptado para a necessidade atual (BECK, 2000).

- ***Refactoring*** – a regra de remoção de código duplicado é outra maneira de se fazer *refactoring*. Mas os testes conferem ao desenvolvedor, que grandes mudanças feitas no sistema, não alteraram o funcionamento do mesmo. Quanto mais seguro de que mudanças feitas não alteram o sistema, mais agressivo se torna o *refactoring* aplicado pelo desenvolvedor (BECK, 2000).

- **Entrega Contínua** – com testes realizados no sistema, têm-se mais confiança na entrega de partes do sistema, e o código vai para produção mais rápido (BECK, 2000).

2.4 POR QUE TDD?

A vantagem significativa de TDD é que ele permite que o desenvolvedor ande pequenos passos ao escrever software. Por exemplo, suponha que você adicione um novo código funcional, compile e teste. As chances são muito boas que seus testes serão quebrados por defeitos que existem no novo código. É muito mais fácil de encontrar, e em seguida, corrigir, estes defeitos se você escreveu duas novas linhas de código do que duas mil. A

implicação é que quanto mais rápido for seu compilador e suite de testes de regressão, mais atraente ele é para continuar em passos cada vez menores. Segundo Beck (2000) "o ato de escrever um teste de unidade é mais um ato de design do que de verificação. Também é mais um ato de documentação do que de verificação. O ato de escrever um teste de unidade fecha um número notável de loops de *feedback*".

A primeira reação que muitas pessoas têm de técnicas ágeis é que são as melhores técnicas para projetos pequenos, talvez envolvendo um punhado de pessoas durante vários meses, mas que não é recomendado para trabalhar em projetos muito maiores. Isso simplesmente não é verdade. Beck (2003) relata trabalhando em um sistema *Smalltalk* uma abordagem totalmente orientada a testes, que teve 4 anos e 40 pessoas trabalhando, resultando em 250 mil linhas de código funcional e 250 mil linhas de código de teste. Há 4.000 testes em menos de 20 minutos, com o conjunto completo que está sendo executado várias vezes ao dia.

2.5 O CICLO DO TDD

O ciclo do TDD se resume em:

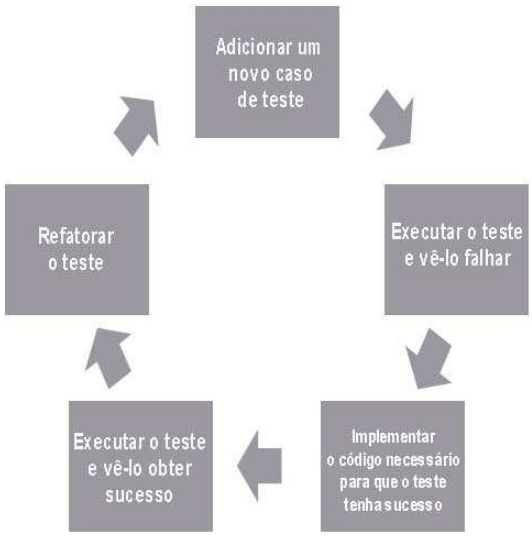

Figura 6 - Ciclo de um caso de teste na metodologia TDD (BORGES, 2006)

Cada pequena iteração possui um micro objetivo, que terá sido alcançado quando os testes criados antes da implementação do código passarem. Essa forma de implementar tem como vantagem a redução do escopo que o desenvolvedor deve focar, pois assim este pensará apenas em um micro objetivo, ao invés de se preocupar com todo o software (FEITOSA,

2007). Com isso, o desenvolvedor escreve um caso de teste pensando no comportamento que o software deve ter, ao invés de se preocupar em como será implementado, sendo este o verdadeiro foco da metodologia TDD.

No TDD, os testes são automatizados para que sejam executados repetidamente, através do uso de frameworks como o PHPUnit (BORGES, 2006). Através dos resultados, falha ou sucesso, os programadores julgam o progresso do desenvolvimento. A partir disso, os programadores fazem continuamente pequenas decisões aumentando as funcionalidades do software. Todos estes casos de teste devem ser realizados com sucesso sucessivamente antes de um novo código ser considerado totalmente implementado (BORGES, 2006).

Devido ao fato de que nenhum código é escrito a não ser para passar em um teste que esteja falhando, os testes automatizados tendem a cobrir cada caminho de código e evita que sejam criados códigos desnecessários, e como todas as funcionalidades e melhorias do código começam com um teste, o desenvolvedor precisa conhecer os casos de uso que contemplem todos os requisitos e exceções do sistema. Essa técnica obriga o desenvolvedor a focar no requisito para escrever bons testes.

Na figura 7 será demonstrado um pequeno exemplo de um ciclo do TDD utilizando o PHPUnit. O objetivo do teste demonstrado na figura 7 é retornar o protocolo HTTP de um site qualquer.

```
1  <?php
2
3  /**
4   * Definição do teste:
5   * Retornar o protocolo do site
6   *
7   * @author Allan
8   */
9
10 require_once '../public/library/SplitDomain.php';
11
12 class SplitDomainTest extends PHPUnit_Framework_TestCase {
13
14     public function testDeveRetornarHttp(){
15         $url = "http://www.google.com.br";
16         $splitDomain = new SplitDomain();
17         $this->assertEquals('http',$splitDomain->getProtocol($url));
18     }
19 }
```

Figura 7 - Teste de verificação de protocolo utilizando PHPUnit

O primeiro passo do TDD é construir o teste baseado no escopo do caso de teste. Na figura 7 o teste para retornar o protocolo HTTP é criado. O método “assertEquals” recebe dois parâmetros, o resultado esperado e o dado a ser testado, no caso o método “getProtocol” passando a variável “url” como parâmetro do método.

```
7   class SplitDomain {
8
9       public function getProtocol($url){
10
11      }
12  }
```

Figura 8 - Classe SplitDomain e método getProtocol a serem testados

Após a criação do teste foi criada a classe e o método que serão testados.

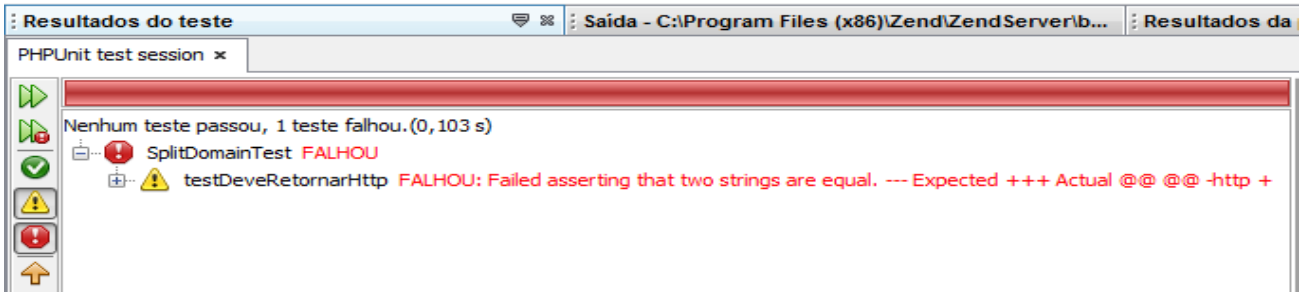

Figura 9 - Ciclo do TDD: falha de teste

Parte do ciclo está completo, o teste foi executado e resultou em falha, pois o teste deveria retornar o protocolo http, mas conforme a figura 9 exibe, não foi retornado nada.

O próximo passo é fazer com que o teste passe da maneira mais fácil possível, conforme mostra a figura 10.

```
7   class SplitDomain {
8
9       public function getProtocol($url){
10          return 'http';
11      }
12  }
```

Figura 10 - Código necessário para que o teste passe

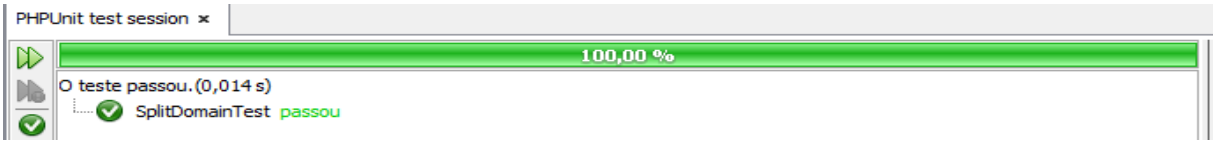

Figura 11 - Resultado do teste

A próxima fase do TDD é refatorar o código e fazer com que o teste continue passando.

```
7   class SplitDomain {
8
9       public function getProtocol($url){
10          $url = explode("://",$url);
11          return $url[0];
12      }
13  }
```

Figura 12 - Código da classe refatorado

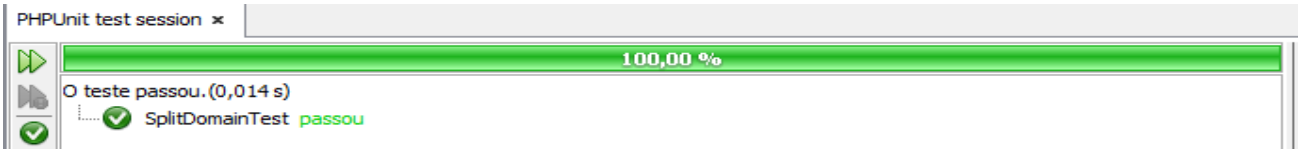

Figura 13 - Resultado do teste após refatoração do código

Após a refatoração do código, o teste continua passando, o que mostra ao desenvolvedor que o que foi definido no escopo do teste está funcionando corretamente, no andamento do projeto, o código pode ser refatorado N vezes, aumentando assim a qualidade do mesmo.

Basicamente o fluxo de desenvolvimento de um software utilizando o TDD seria esse demonstrado na figura 14:

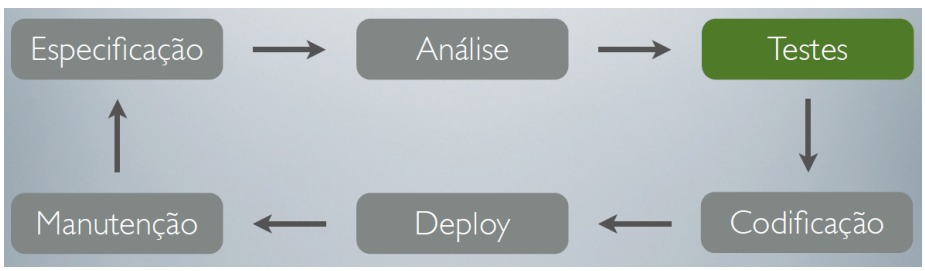

Figura 14 - Fluxo desenvolvimento de software utilizando TDD (DOHMNS, 2010)

2.6 REFACTORING

Refactoring é uma técnica utilizada para melhorar o software existente. Ter o código fonte que é compreensível ajuda a garantir um sistema sustentável e extensível. "Refactoring é o processo de mudança de um sistema de software de tal forma que não altere o comportamento externo do código, melhorando também a sua estrutura interna. É uma disciplina que visa melhorar código e que minimiza as chances de introduzir erros. (Martin Fowler)"

Refatoração simplesmente significa "melhorar o projeto de código existente sem alterar seu comportamento observável" (BECK, 2000). Originalmente concebido na comunidade *Smalltalk*, que se tornou agora uma técnica de desenvolvimento muito utilizada. Enquanto as ferramentas de refatoração tornam possível aplicar refactorings muito facilmente, é importante que o desenvolvedor entenda o que o refactoring faz e por que ele vai ajudar nesta situação por exemplo, permitir a reutilização de um bloco de código repetitivo.

Cada refatoração é um processo simples que faz uma mudança lógica para a estrutura do código. Ao mudar várias linhas de código de uma só vez é possível que erros sejam introduzidos. Mas quando e onde estes erros foram criados não é algo fácil de descobrir. Se, no entanto, uma mudança é implementada em pequenos passos com testes efetuados a cada etapa, o erro será descoberto no teste executado imediatamente após a sua introdução no sistema.

Este é o benefício de testes de unidade em um sistema abrangente, algo defendido por técnicas de *Extreme Programming*. Estes testes, fornece aos desenvolvedores e gestores a confiança de que a refatoração não rompeu o sistema, o código se comporta da mesma forma como ele se comportava antes (BECK, 2000).

A tabela 1 exibe como o processo de refatoração é realizado:

Ação	**Perguntas a serem feitas e ações a serem tomadas**
Detectar um problema	Existe um problema? Qual é o problema?
Caracterizar o problema	Por que é necessário mudar alguma coisa? Quais são os benefícios? Há algum risco?

Projetar uma solução	Qual deve ser o "estado objetivo" do código? Qual a transformação de código (s) irá mover o código para o estado desejado?
Modificar o código	Passos que irá realizar a transformação de código (s) que deixar o código funcionando da mesma forma como fazia antes.

Tabela 1 - Refactoring (AGILE DATA)

2.6.1 VISÃO GLOBAL

Quando o código fonte de um sistema é facilmente compreensível, o sistema é mais rentável, levando à redução de custos e permitindo que os recursos de desenvolvimento possam ser usados em outro lugar. Ao mesmo tempo, se o código é bem estruturado, novos requisitos podem ser introduzidos de forma mais eficiente e com menos problemas. Estas duas tarefas de desenvolvimento, manutenção e reforço, muitas vezes entram em conflito com novos recursos, especialmente aqueles que não se encaixam corretamente dentro do projeto original, resultando em um esforço de manutenção maior. O processo de refatoração visa reduzir este conflito, adicionando alterações não destrutivas para a estrutura do código-fonte, a fim de aumentar a clareza do código e de manutenção (GOLDMAN, 2002).

No entanto muitos desenvolvedores e gerentes hesitam em usar o *refactoring*. As razões mais óbvias para isso é a quantidade de esforço necessária para introduzir mesmo uma pequena mudança, e uma possibilidade de introduzir erros no sistema. Ambos estes problemas podem ser resolvidos usando uma ferramenta de *refactoring* automatizada (GOLDMAN, 2002).

2.6.2 UMA ATIVIDADE EMERGENTE PARA O DESENVOLVIMENTO DE SOFTWARE

Software se inicia em pequena escala, e mais do que isso, bem desenhado. Ao longo do tempo o tamanho dos softwares e a complexidade dos mesmos aumentaram, aumentando assim a ocorrência de erros e, assim, diminuindo a confiabilidade do código. Desenvolvedores de software, especialmente quando eles não são os autores originais, estão encontrando cada vez mais dificuldade em manter o código, e ainda mais para ampliar. A base de código, que

em qualquer empresa de software deveria ser um bem valioso, em algum momento pode tornar-se um bem passivo.

A atenção dos desenvolvedores e gestores do software é o fator mais importante para evitar o envelhecimento prematuro do software. O *refactoring* pode reverter esse envelhecimento, quando aplicado adequadamente, de preferência com boas ferramentas de software que ajudam na detecção, análise e caracterização dos problemas e, finalmente, permitir que possa corrigi-los (GOLDMAN, 2002).

Desenvolvedores de software bem treinados que estão familiarizados com o seu código, muitas vezes são bem conscientes dos problemas escondidos no código. No entanto, a maioria dos desenvolvedores não está disposta a fazer alterações na estrutura do código, especialmente se as alterações podem levar algum tempo. Se os desenvolvedores encontrarem meios simples de aplicar operações de refatoração para seu código, eles vão mostrar menos resistência para trabalhar tal reestruturação (GOLDMAN, 2002).

Reescrever um componente é muitas vezes visto como mais fácil pelo desenvolvedor, ou pelo menos, menos confusa. O código fonte atual pode ter mudado ao longo do tempo a partir do desenho original, e pode não ser imediatamente claro para um desenvolvedor que está vendo o código pela primeira vez. Alternativamente, o desenvolvedor original pode se arrepender por certas decisões de projeto, e agora acredita que há uma maneira melhor. No entanto, este ignora o fato de que o código fonte tem muito valor escondido. As correções de erros contidos no código fonte, não podem ser todos documentados, porém eles são muito valiosos (GOLDMAN, 2002). O componente foi previamente testado exaustivamente no ambiente de produção, e isso não é algo que deve ser jogado fora. Refatoração mantém esse valor oculto, garantindo que o comportamento do sistema não se altere.

Gestores são muitas vezes relutantes em permitir alterações que não irão dar qualquer benefício imediato visível, "Se não está quebrado, não conserte". Assim, a administração pode se preocupar com o problema da introdução de erros a um sistema que já foi exaustivamente testado. No entanto, se as operações de refatoração não representam uma ameaça de introdução de erros, o gestor será menos relutante para deixar o refactoring prosseguir (GOLDMAN, 2002).

CAPÍTULO 3 – IMPLEMENTAÇÃO

Foi feito um estudo aprofundado de diversas tecnologias passíveis de serem usadas, cujos parâmetros analisados foram: confiabilidade, segurança e desempenho.

Na camada de apresentação deste estudo, foi utilizada a linguagem de marcação padrão, o HTML (HyperText Markup Language), seguindo as regras da W3C (Consórcio World Wide Web, órgão responsável pelo desenvolvimento de padrões e diretrizes para a Web, http://www.w3c.br) e o Javascript, como linguagem usada do lado do cliente (client-side). Já na camada de lógica de negócios, a tecnologia que mais se adequou às exigências foi o PHP, correspondendo às demandas de segurança, estabilidade e alta produtividade. E na camada de dados foi usado o banco de dados MySQL.

Nas sessões que se seguem serão descritas detalhadamente as ferramentas, as tecnologias e as metodologias de desenvolvimento utilizadas.

3.1 PHP 5

O PHP é uma linguagem de extensão (scripting, i.e., que permite o controle de uma ou mais aplicações) interpretada, de propósito geral, perfeitamente adequada para o desenvolvimento web e que pode ser embutida no código HTML. Foi criada por Rasmus Lerdorf, 1995 e logo foi disponibilizada para que a comunidade pudesse contribuir com sua evolução e na correção de problemas.

O suporte à programação orientada a objetos (OO) se deu a partir da versão 3.0, após participação de Zeev Suraski e Andi Gutmans, que são atuais diretores da Zend Technologies (empresa que é cooperadora e mantenedora do PHP e que lançou o ZF).

Porém, cabe salientar, que o PHP não é uma linguagem de programação orientada a objetos, e sim, que tem suporte à orientação a objetos. O que significa dizer que é possível desenvolver tanto com programação estruturada (baseada em funções) como com OO (baseada em classes) ou até mesmo combinando-as, dando maior versatilidade e liberdade ao programador.

Tecnologicamente, o PHP é uma linguagem idealizada para aplicações web, uma vez que tem suporte a inúmeras bibliotecas de código e tem a capacidade de ser integrada a outras linguagens e/ou plataformas com facilidade. Já economicamente, é uma linguagem aberta, livre, gratuita e sem restrições para o uso comercial. O fácil aprendizado é outro fator que deve ser levado em consideração, haja vista a existência da vasta documentação, dos fóruns e das listas de discussão. Além disso, o PHP é multiplataforma e portável, uma vez que não há limitação quanto ao sistema operacional.

3.2 A ARQUITETURA MODEL-VIEW-CONTROLLER (MVC)

Anteriormente, a grande maioria das aplicações era monolítica, isto é, de uma só camada e eram desenvolvidos para serem usados em uma única máquina. Geralmente todas as funcionalidades se encontravam em um módulo, gerado por uma imensa quantidade de linhas de código e de manutenção extremamente complexa. A entrada do usuário, a verificação, a lógica de negócio e o acesso ao banco de dados estavam centralizados em um mesmo local.

Entretanto, com a demanda de compartilhamento da lógica de acesso aos dados entre diversos usuários simultaneamente, surgiram as aplicações de duas camadas (2-tier), onde a base de dados foi separada da camada de apresentação e da camada da lógica de aplicação. Porém, ainda havia problemas, como o controle de atualizações, cuja gerência era descentralizada e, a dificuldade de manutenção do sistema, haja vista que os programas eram instalados nos clientes.

Com o intuito de sanar tais problemas, vieram as aplicações de três camadas (3-tier). E, para acompanhar esse novo modelo, passa a ser necessária a adoção de diversos padrões de projetos (design patterns), dentre eles o MVC, que visa aumentar a escalabilidade, a eficiência, a reutilização e, consequentemente, a produtividade.

O MVC é um padrão de arquitetura de software que separa a lógica de negócio da lógica de apresentação, permitindo que o desenvolvimento cada uma seja feito separadamente.

Na arquitetura em questão, o modelo representa os dados da aplicação e as regras de negócio que governam o acesso e a manipulação deles. Além disso, ele mantém a persistência

dos dados e fornece, ao controlador, o acesso às funcionalidades presentes na aplicação, encapsuladas no próprio modelo.

Já na visualização, um componente processa o conteúdo desejado (proveniente do modelo) e encaminha ao controlador as ações do usuário. Alimenta templates com esses dados, mas de forma alguma os modifica, só os apresenta.

O núcleo da aplicação é a camada de controle, que é onde o comportamento do sistema é definido. É nessa camada que as ações do usuário são interpretadas e mapeadas para as chamadas do modelo. Esta camada que seleciona as visualizações a serem exibidas, fazendo com que o modelo manipule os dados, a partir da respectiva resposta à solicitação do usuário e os envie para que sejam apresentados.

Na Figura 15 é possível ter uma visualização superficialmente de como é a estrutura da arquitetura em questão.

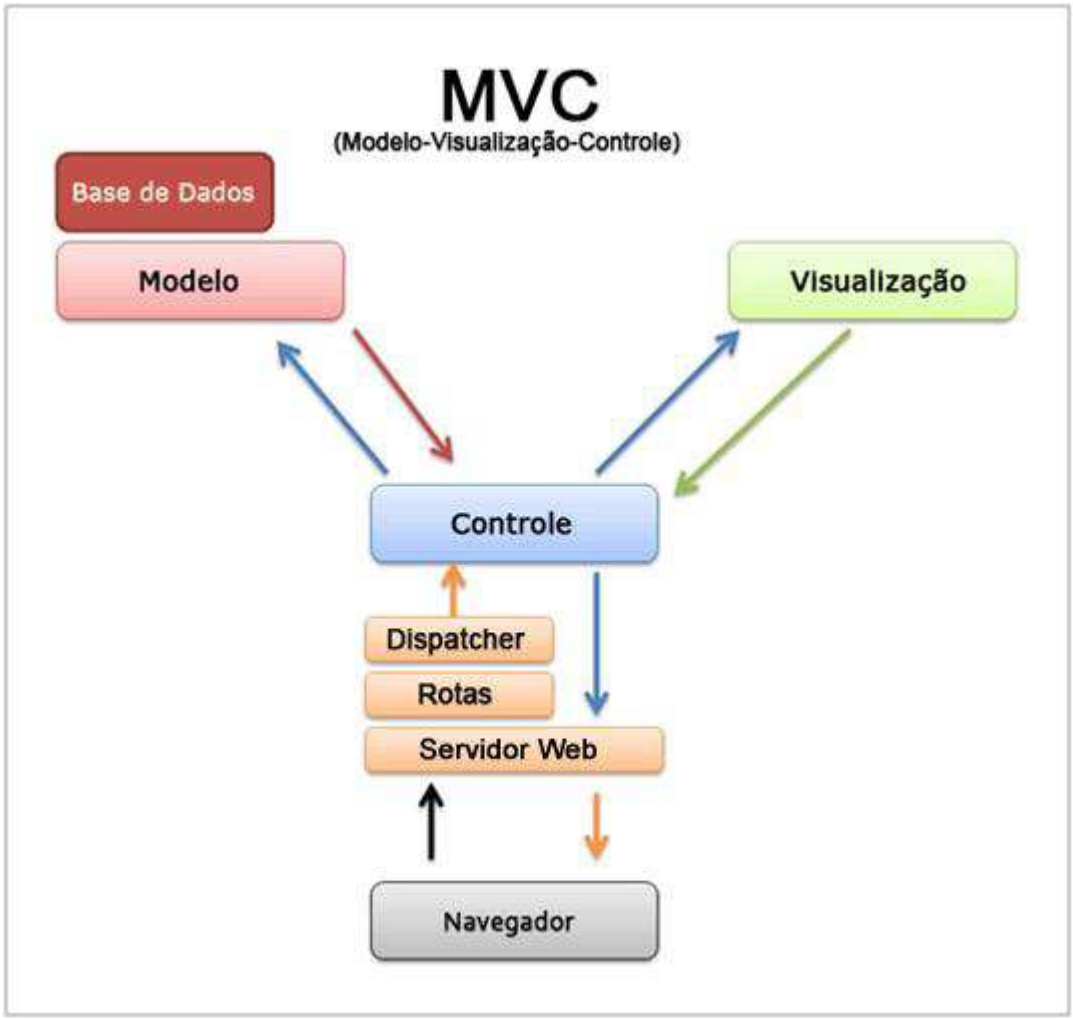

Figura 15 - Fluxo MVC Zend Framework (SANTOS, 2009)

Na figura 16, é mostrado o funcionamento, propriamente dito, das 3 camadas. A função de cada camada, o fluxo de dados e onde acontecem os devidos processamentos.

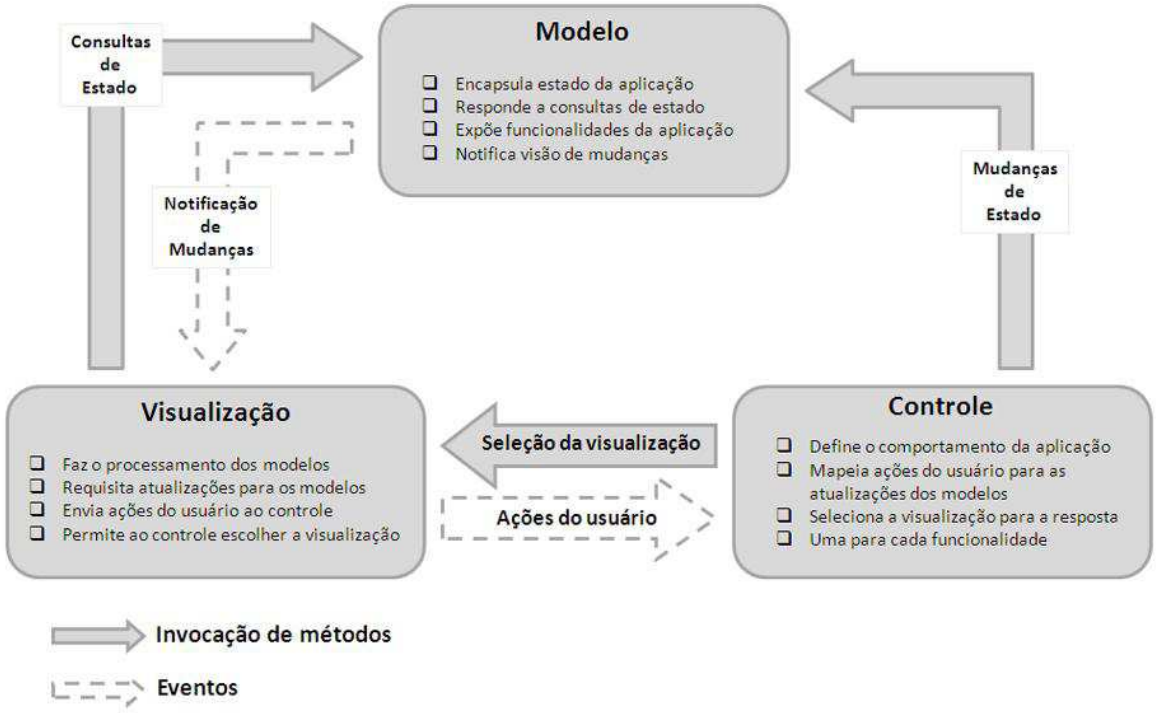

Figura 16 - Fluxo MVC (SANTOS, 2009)

Foi assumido como um padrão de projeto e tem sido considerada uma boa prática, quando se trata de aplicações web, pois o MVC é capaz de gerenciar múltiplos visualizadores utilizando um mesmo modelo, tornando fácil efetuar testes, manutenções e atualizações; torna a aplicação escalável; aumenta, significativamente, a produtividade, por ser possível o desenvolvimento em paralelo do modelo, da visualização (design) e dos controles, independentemente.

3.3 FRAMEWORKS

De acordo com o Prof. Elton Luis Minetto (2007): — O *framework* é uma base de onde se pode desenvolver algo maior ou mais específico. São coleções de códigos, classes, funções, técnicas e metodologias que facilitam o desenvolvimento de novos softwares.

A motivação de utilizar um *framework* passa por diversos campos, indo desde o reuso de código já trabalhado (naturalmente isso depende da credibilidade e da experiência dos

desenvolvedores), passando pela padronização de uma estrutura de projeto já idealizada e efetivamente elaborada, chegando à complexidade que alguns alcançaram, devido ao número de colaboradores, de tempo de mercado e também da quantidade de erros que já foram tratados.

Outro quesito de extrema importância é a segurança do projeto, uma vez que a qualidade de uma implementação se dá, em grande parte, na segurança que está embutida no sistema produzido.

Afora o já exposto, os *frameworks* atuais, adotam a programação orientada a objetos e diversos padrões de projetos, como o MVC, o *singleton*, *observer*, *factory*, DAO (*Data Access Object*), entre outros, o que uniformiza o desenvolvimento, a estrutura dos sistemas e as metodologias utilizadas, diminui o custo de aprendizagem, e com isso, aumenta a manutenabilidade das aplicações.

O conhecimento deve estar sempre documentado, devem haver padrões e métodos que uniformizem os processos, seja em empresas pequenas ou nas grandes, de modo que um projeto não dependa das pessoas, mas das funções que elas exercem. Isso é uma maneira de assegurar qualidade e disciplina, para atingir as metas propostas. A utilização de um *framework* de desenvolvimento estimula esse processo.

3.4 ESCOLHA DO FRAMEWORK

Tendo terminada a fase de análise e estudo, surge uma nova fase de pesquisa: a busca pelo melhor conjunto de ferramentas de desenvolvimento, pensando na produtividade, na segurança e na confiabilidade do produto.

A escolha do Zend Framework para este projeto foi obtida através de dados coletados em pesquisa, alguns desses dados que auxiliaram na decisão estão na tabela 2:

A tabela 2 demonstra um comparativo de frameworks:

Frameworks \ Aspectos avaliados	**CakePHP**	**CodeIgniter**	**Yii**	**Zend**	**Prado**
PHP 4.0	✓	✓	✗	✗	✗
PHP 5.0	✓	✓	✓	✓	✓

MVC	✓	✓	✓	✓	✓
Suporte a diversos SGBDs	✓[1]	✓	✓[1]	✓	✓
Segurança	✓	✓	?	✓	?
Alta velocidade de processamento	✗	✓	?	✓	?
Facilidade de configuração	✓	✓	✗	✓	✗
Facilidade de aprendizado	✓	✓	?	✗	?
Documentação	✓	✓	✗	✓	?
Confiabilidade	?	✓	?	✓	?
Manutenção	?	✓	?	✓	?
Interface	✓	✓	✓	✗	✓
Escalabilidade / Modularização	✓	✓	✓	✓	✓
Adaptabilidade	✓	✓	?	✓	?
Ajax	✓	✓	✓	✓	✓
ORM	✓	✗	✓	✓	✓
Validação	✓	✓	✓	✓	✓

Tabela 2 - Comparação Frameworks

* Tabela adaptada de http://www.phpframeworks.com/, maio de 2011

Legenda:
✓ Avaliação positiva do aspecto
✓[1] Avaliação positiva, porém com suporte limitado a alguns SGBD
✓[2] Avaliação positiva, porém com análise um pouco subjetiva desta facilidade
✓[3] Avaliação positiva, porém a documentação da comunidade deixa a desejar
✗ Avaliação negativa do ponto em questão
? Avaliação inconclusiva do quesito analisado

Tabela 3 - Legenda

Os *frameworks* mais profundamente analisados foram: CakePHP, CodeIgniter e o Zend, porém foi dado uma ênfase maior ao Zend Framework (ZF) devido à quantidade de documentação disponível e ao suporte ao SGBD usado nesta pesquisa.

O ZF foi o que mais se adequou ao ambiente existente. Além da credibilidade dos desenvolvedores do ZF, por serem criadores da tecnologia PHP, outros fatores foram

determinantes para a escolha desta ferramenta, bem como: a existência de diversas classes já implementadas e exaustivamente testadas, trazendo mais confiabilidade ao sistema desenvolvido e o excelente desempenho obtido quando submetido a altas cargas de processamento.

Feito isso, é realizada a análise da primeira camada, que e de suma importância, uma vez que é a interface do sistema que será exibida para o usuário. Pensando nisso, foi observada a necessidade da utilização de uma biblioteca JavaScript, visando facilitar a validação de dados e a implementação de requisições assíncronas, o Ajax (*Asynchronous Javascript And XML*), tornar a interface mais amigável e aumentar a interatividade do usuário com o sistema, já que havia a intenção de adequação aos padrões da Web 2.0, onde a interatividade e uma constante.

Então, usando como base a escolha do ZF, foram realizados testes de adaptabilidade com algumas das mais famosas bibliotecas de Javascript, e obtive-se os seguintes resultados:

- jQuery - Integração média. Possuí um amplo leque de efeitos oferecidos em sua versão (1.4.2), além de ter sido considerada, nesta análise, a mais completa, de uso mais simples e, comprovadamente, a biblioteca com maior numero de menções na web nos últimos 12 meses (http://www.google.com.br/trends?q=jquery%2Cdojo%2Cprototype&ctab=0&geo=all&date=ytd&sort=1), o que dá uma maior confiança para usá-la.

3.5 ZEND FRAMEWORK

Tomando como premissa a simplicidade de desenvolvimento, o ZF conseguiu atingir esse objetivo. Tem como foco a construção de aplicações web, seguindo as regras da Web 2.0 e possui *web services* seguros, confiáveis e atuais, usando API’s de grandes empresas como o Google, a Amazon e o Yahoo.

Vale ressaltar que o aumento da produtividade foi idealizado desde a implementação do framework. Através de uma biblioteca leve e fracamente acoplada, permitindo a fácil personalização do produto, o Zend usa também uma base de códigos exaustivamente testada e

extensível, uma arquitetura flexível, além da integração com diversas ferramentas modernas aumentando a acessibilidade e a usabilidade do sistema.

Com licença segura e confiável, a Zend Framework's License, baseada na BSD License, assegura que seu código seja livre, ou seja, irrepreensível, e protegido, seguindo os padrões, em conformidade com as regras que são consideradas as melhores práticas de programação.

O núcleo do ZF é um diretório de classes, seguindo um padrão semelhante ao usado no Java, simulando uma estrutura de pacotes. O objetivo dessa padronização de arquitetura é a reutilização do código sem que o desenvolvedor tenha que alterar a biblioteca nativa do framework, podendo simplesmente criar pastas e inserir conteúdos dependendo do uso do software.

O Zend é uma ferramenta com uma diversidade de funções. Suporta conexão a praticamente todos os SGBDs, uma característica que foi essencial para a definição do framework utilizado. Implementa a conexão, consultas e alteração de dados, bem como controle de transações via PDO (PHP *Data Object*), que é uma biblioteca de abstração da camada de banco de dados para PHP, isto é, fornece suporte a diversos SGBDs, tornando o seu código reutilizável - porém muito mais rápida, pois foi escrita em uma linguagem compilada (C/C++), e, ao invés de trabalhar com erros, lida com exceções, o que torna facilita o tratamento de inconformidades de dados e outras falhas. O PDO representa uma camada de abstração de acesso a banco de dados, não importando, portanto, qual o banco usado, uma vez que as mesmas funções de manipulação de dados ou recuperação de informações são as mesmas.

Em suma, com o intuito de facilitar qualquer tipo de manutenção ou atualização necessária ao sistema, o Zend se adéqua perfeitamente na solução proposta.

3.6 INSTALAÇÕES E CONFIGURAÇÕES

Para que o sistema web funcionasse corretamente foi necessária a transformação do computador utilizado em um servidor web. Para tanto, foi definido o seguinte ambiente de desenvolvimento:

- Instalação do Zend Server

- Com o Apache 2.2.5.
- Usando o PHP 5.2.5.
- Foi necessário o download do ZF e do jQuery.
- A IDE (Integrated Development Environment) utilizada foi o Netbeans.
- Instalação da PEAR
- Instalação do PHPUnit em conjunto com o Zend Server

Apos a instalação dos devidos programas e o ajuste de detalhes, a aplicação funcionará corretamente assim que toda ela for transferida para o servidor em questão.

Foram feitas configurações iniciais em arquivos .htaccess, garantindo que o fluxo das informações siga o desejável, de acordo com a premissa do modelo MVC e seja passível de gerenciamento centralizado.

No arquivo que controla esse fluxo, o chamado bootstrap ou index.php, diversas definições foram implementadas, como por exemplo:

- Forma de exibição de exceções;
- Caminho de arquivos para inclusão;
- Carregador automático de classes do ZF;
- Tratamento de requisições;
- Tipo de documento;
- Codificação de dados;
- Processamento de maquinas de templates;
- Caminho base;
- Inicio de sessão;

- Diferentes rotas, com o intuito de criar URLs amigáveis;
- Controladores;
- Configurações de banco de dados;
- Localização;
- Internacionalização;
- Entre outras.

3.7 A APLICAÇÃO

O projeto desenvolvido foi um breve cadastro de papéis, recursos e privilégios utilizados pelo ZF para validação de autorização de usuários em sistemas protegidos.

A seguir serão explicados os componentes de Autorização do Zend Framework, demonstrando como restringir acesso a recursos de um sistema, baseando-se nos privilégios que um usuário autenticado possui.

Autorização é o ato de verificar as permissões de um usuário já autenticado no sistema e, baseando-se nessas permissões, permitir ou bloquear o acesso deste usuário a determinados recursos da aplicação. Se, por exemplo, em um Sistema de Gestão de Conteúdo (CMS), o usuário logado é um escritor, ele poderia ter acesso à escrita de artigos, porém não poderia ter acesso ao cadastro de novos usuários, pode-se obter este tipo de funcionalidade por meio de um componente de autorização. No Zend Framework este componente é o **Zend_Acl**, que fornece a funcionalidade de Lista de Controle de Acesso (ACL) e gestão de privilégios. Em geral, uma aplicação pode usar esta funcionalidade para controlar o acesso a certos objetos protegidos, requeridos por outros objetos.

Existem alguns termos utilizados para melhor explanar as entidades envolvidas no controle de acesso, cada um deles é detalhado a seguir.

Papel (Role)

Um papel corresponde a uma responsabilidade de um usuário dentro de um sistema como, por exemplo, o papel de “colunista” ou de “membro”. Isto é encapsulado através da classe **Zend_Acl_Role**, que é uma classe simples que apenas armazena o nome do papel. Existe também herança de papéis, onde ao se herdar de um papel, é possível fazer tudo que o papel pai faz, além das ações específicas do papel filho. Para se criar um papel e adicioná-lo na lista de controle de acesso.

Recurso (Resource)

Um recurso é algo a ser protegido, o que pode ser um controlador ou uma ação. Um recurso é encapsulado pela classe **Zend_Acl_Resource**, que simplesmente armazena o nome do recurso que será protegido. Assim como no caso do **Zend_Acl_Role**, a classe **Zend_Acl_Resource** oferece suporte a herança, esta sendo definida no segundo parâmetro do método **add()** de **Zend_Acl**. Um exemplo de utilização de recursos é o caso de um sistema de ERP onde usuários do papel “administrador” podem ter acesso ao controlador **manutencao**, já usuários do papel “operador” podem ter acesso ao controlador **produto**.

Privilégios (Privilege)

Um dado recurso pode ter diversas permissões a um determinado papel, estas permissões são, tipicamente, baseadas nas operações que serão executadas. Exemplos destas operações podem ser ações de um controlador, como, por exemplo: “adicionar” e “visualizar”. Este tipo de acesso exigido é facilmente configurado com dois métodos do Zend_Acl: **allow()** e **deny()**.

Estes são os principais conceitos e métodos relacionados aos componentes de autorização do Zend Framework.

No projeto proposto foi implementado somente a inserção e atualização de papéis, recursos e privilégios, pois a proposta desse estudo não é implementar um sistema utilizando a ACL, mas sim demonstrar como o TDD auxilia no desenvolvimento de softwares.

3.8 DIAGRAMA DE CLASSES

Vale mencionar que somente estão ilustradas uma parte das classes implementadas, com o intuito de facilitar a compreensão e o entendimento do que foi feito no sistema. Horizontalmente destacou-se o que foi desenvolvido pelo ZF e o que pertence ao projeto desenvolvido.

A Figura 17 exemplifica algumas classes do ZF e mostra de que forma o sistema interage com elas. As classes desenvolvidas herdam ou estendem às do *framework*. Então, no exemplo da figura 17, as classes “TDD_Forms_Privilege”, “TDD_Forms_Resource” e “TDD_Forms_Role” herdam da classe Zend_Form.

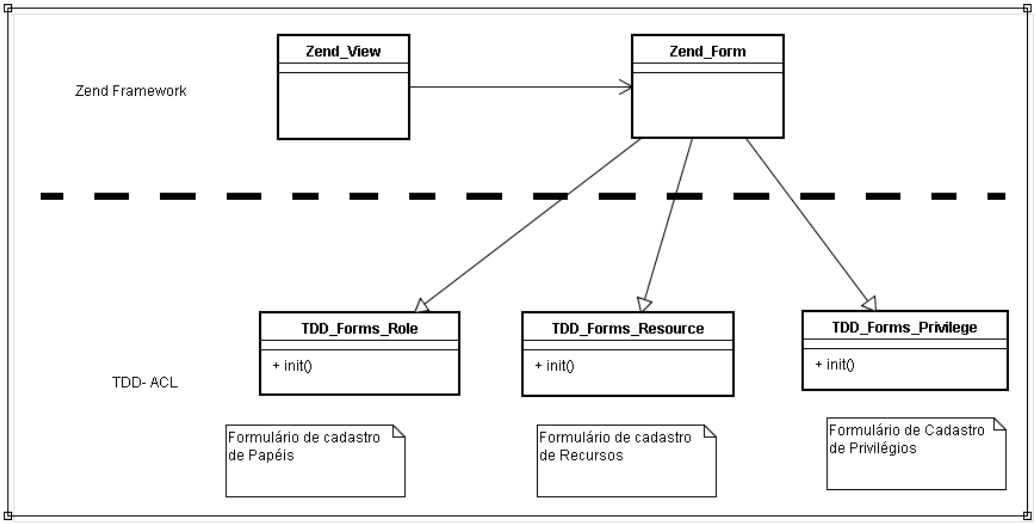

Figura 17 - Diagrama de classes formulários

As classes da Figura 17, que possuem o prefixo “TDD_Forms” implementam formulários, como de cadastro de papéis, de alteração de papéis, de inclusão de novos privilégios e de listagem dos dados. Estas herdam atributos da classe do ZF, a Zend_Form, que interagem com outra classe do ZF. A Zend_View, responsável pela visualização do conteúdo.

Da mesma forma, a Figura 18 demonstra que as classes “Admin_RoleController”, “Admin_ResourceController” e “Admin_PrivilegeController” herdam da classe “TDD_Controller_Action” que herda da classe “Zend_Controller” e “Default_ErrorController” que herda da classe “Zend_Controller”.

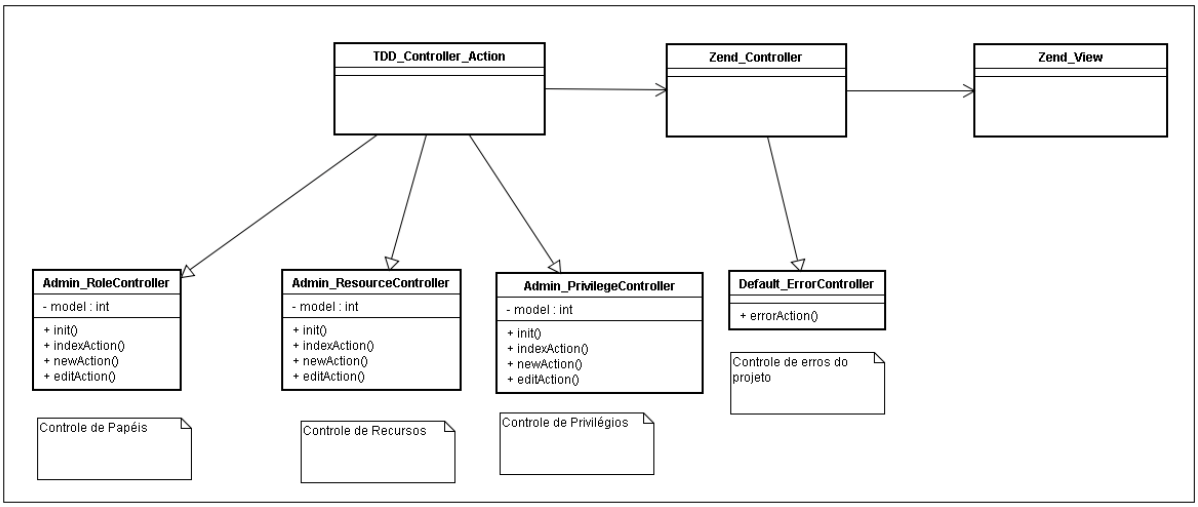

Figura 18 - Diagrama de classes controllers

Na Figura 18, as classes com o sufixo Controller são os chamados controladores do modelo MVC e são responsáveis pelo processamento dos dados. O Admin_RoleController, por exemplo, trabalha na validação e na verificação dos dados dos papéis que serão cadastrados no sistema. Já o Admin_ResourceController é responsável pela inclusão de recursos no banco de dados, e também da validação de cada campo enviado para realizar essa inserção. Já o Admin_PrivilegeController é responsável pela inclusão de privilégios no banco de dados, inserindo um privilégio para cada recurso e papel previamente cadastrado.

3.9 INTERFACE

O primeiro contato do usuário com o sistema se dá nessa camada. Tudo o que foi desenvolvido será visível somente nesse nível de abstração.

Por usar uma grande quantidade de formulários, foi necessária uma validação encadeada de dados, através da linguagem do lado do cliente, o javascript alinhado com o jQuery. Todas as interações do usuário serão avaliados e processados pelo navegador de modo a serem aprovadas dependendo da classificação do campo e do conteúdo a ser submetido.

Utilizaram-se requisições assíncronas conhecidas como AJAX, evitando que a mesma página fosse recarregada a cada envio de dados, o que traria certo desconforto ao operador do sistema.

Além disso, seguimos algumas práticas desejáveis no desenvolvimento do *layout*, como: uso de imagens para representar funcionalidades; uso de metáforas e de uma linguagem coloquial; liberdade maior ao controlador durante o uso.

A seguir uma seqüência de imagens de telas e páginas do sistema desenvolvido.

A figura 19 exibe a tela inicial do sistema.

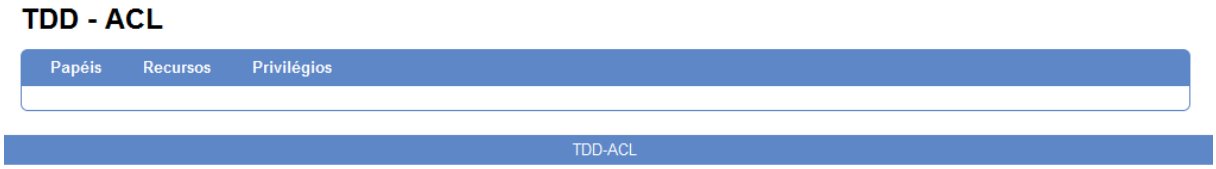

Figura 19 - Página inicial

A figura 20 exibe a listagem de papéis cadastrados no sistema.

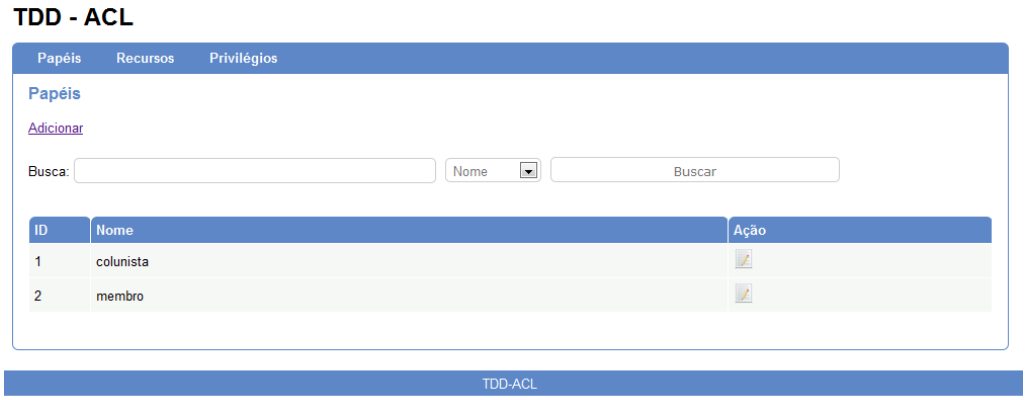

Figura 20 - Adicionar Papéis

A figura 21 exibe os privilégios cadastrados para os recursos “manutenção” e papel “colunista”. Os usuários de papel “colunista” que tiverem o privilégio “visualizar” e “editar” em seus cadastros poderão visualizar e editar os controles de “manutencao”.

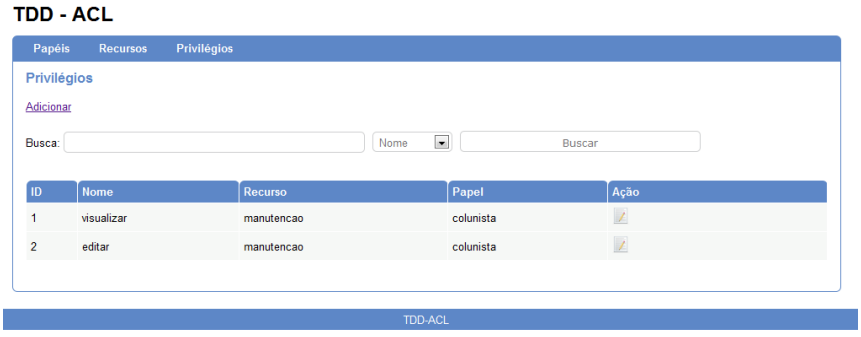

Figura 21 - Privilégios

3.10 MVC DO SISTEMA

A Figura 22 exemplifica o trabalho do modelo MVC usado nesse projeto.

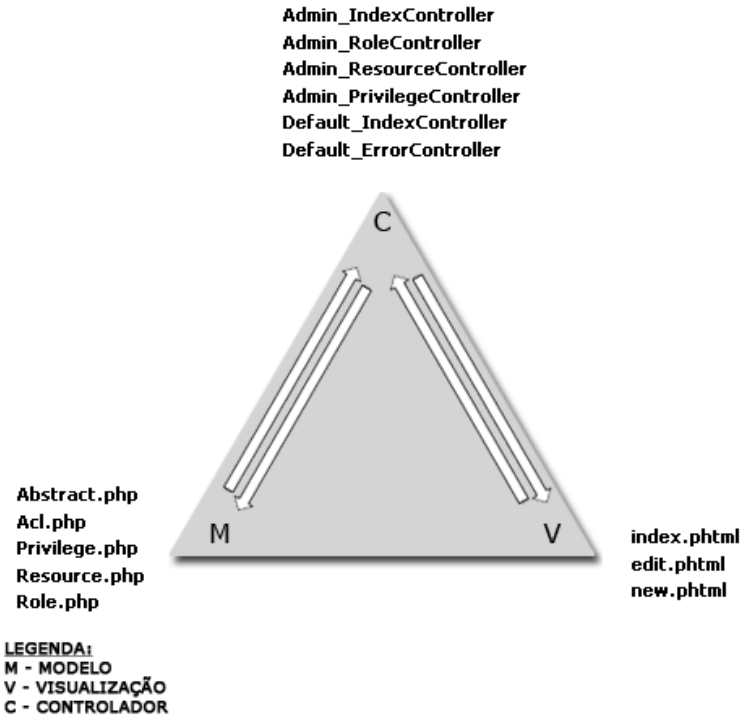

Figura 22 - MVC do sistema

Ocorre uma interação direta entre o controle e o modelo e a visualização, porém, o modelo nunca é acessado pela interface, o que é uma questão importante, pois se trata da segurança do programa.

Para exemplificar, tem-se o Admin_RoleController, que tem um modelo, já com seu objeto mapeado em Role.php, onde todas as consultas ao banco são estruturadas e feitas, e, ao receber o retorno disso, trabalha as informações, seja em formulários seja em uma simples tabela, repassa para a sua visualização, em index.phtml.

Outra forma de interação é no Admin_PrivilegeController, que remete ao modelo a incumbência de acessar o banco e fazer suas consultas, para então tratar os dados e enviá-los para a interface da forma que devem ser exibidos.

É dessa forma que é garantido um único fluxo de dados e, conseqüentemente uma segurança e uma maior organização nos procedimentos.

3.11 TESTES

Müller (2002) relaciona diretamente a fase de testes com a qualidade do serviço produzido. Afirma que “a motivação subjacente ao teste de programa é afirmar a qualidade do software com métodos que podem ser aplicados econômica e efetivamente tanto a sistemas de grande porte quanto de pequeno porte”.

A principal ferramenta utilizada foi o PHPUnit, que é uma comumente usada para testar frameworks e é também um framework por si só.

A figura 23 exemplifica a interface usada juntamente com a IDE Netbeans, como uma extensão do programa.

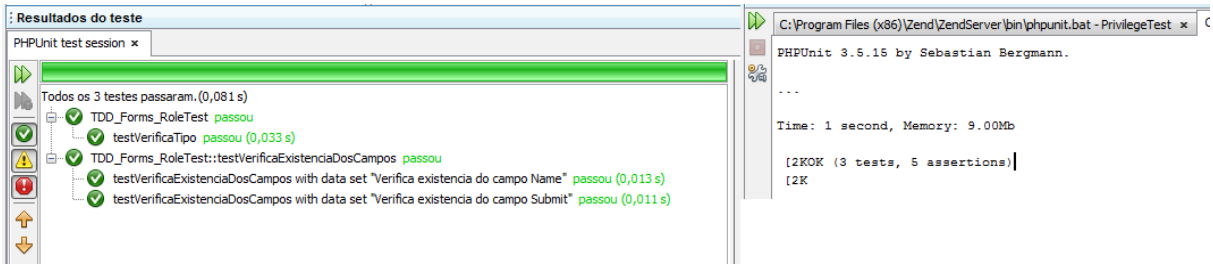

Figura 23 - Testes TDD_Forms_RoleTest

Além de mostrar uma barra que determina o percentual de abrangência dos testes feito na referida classe e uma listagem de classes e métodos testados, bem como o tempo de processamento ao realizar o teste, o software tem uma área de saída de dados, onde o desenvolvedor pode imprimir em tela o valor de diversas variáveis para conferir se estão em conformidade com o planejado.

Cada classe é testada separadamente do código fonte da aplicação. Os testes podem ser feitos em qualquer momento, sem que interfira na produção do programa e são automatizados, sendo executados continuamente durante o ciclo de desenvolvimento. Eles se baseiam em afirmações, onde se define os valores esperados e verifica-se se o resultado procede. Funções que alimentam os testes com valores randômicos podem ser criadas, aumentando assim a assertividade do resultado. São testadas inclusive as exceções lançadas pela aplicação, validando cada caso.

As funcionalidades exibidas na figura 24 são algumas das funcionalidades do sistema testadas e verificadas, as linhas tracejadas são de testes que tiveram alguma falha, portanto não corresponderam à saída que foi definida no escopo do projeto, assim o desenvolvedor poderá verificar o motivo do teste ter falhado e consultar possíveis erros inseridos no sistema. Este tipo de verificação ajuda o desenvolvedor a detectar erros inseridos em novas funcionalidades do sistema, assim caso uma nova funcionalidade seja inserida e os testes demonstrem algum tipo de falha, o desenvolvedor saberá que o código está errado e poderá corrigi-lo antes que as novas funcionalidades cheguem até o produto final.

Acl

- Verifica tipo
- Verifica carregamento de roles
- Verifica carregamento de resources
- Verifica carregamento dos privilegios

Privilege

- Verifica tipo do model
- Deve inserir nova privilege
- Deve falhar ao inserir um nome repetido
- Deve retornar privilege
- Deve editar privilege
- Deve remover privilege
- Deve retornar em pairs

Resource

- Verifica tipo do model
- Deve inserir nova resource
- Deve falhar ao inserir um nome repetido
- Deve retornar resource
- Deve editar resource
- Deve remover resource
- Deve retornar em pairs

Role

- Verifica tipo do model
- Deve inserir nova role
- Deve falhar ao inserir um nome repetido
- Deve retornar role
- Deve editar role
- Deve remover role
- Deve retornar em pairs

PrivilegeController

- Verifica existencia de actions
- Verifica existencia do formulario
- Verifica insercao de um privilege
- Verifica insercao de um privilege vazio
- Verifica campo preenchido edit
- Verifica edicao de um privilege
- Verifica paginacao

ResourceController

- Verifica existencia de actions
- Verifica existencia do formulario
- Verifica insercao de um resource
- Verifica insercao de um resource vazio
- Verifica campo preenchido edit
- Verifica edicao de um resource
- Verifica paginacao

RoleController

- Verifica existencia de actions
- Verifica existencia do formulario
- Verifica insercao de um role
- Verifica insercao de um role vazio
- Verifica campo preenchido edit
- Verifica edicao de um role
- Verifica paginacao

TDD_Forms_Privilege

- Verifica tipo
- Verifica existencia dos campos

Figura 24 - Testes realizados

Na figura 25 é exibido um teste de verificação de inserção de um novo papel na aplicação desenvolvida. Para que o teste seja um sucesso o método “save” deve retornar o valor “1”, que seria o valor de retorno do banco de dados, o primeiro valor da chave primária da tabela “roles” inserido no banco de dados.

O método “assertEquals” faz essa verificação, com o primeiro parâmetro sendo a saída esperada e o segundo a entrada, neste caso o valor retornado do banco de dados.

```
1  public function testDeveInserirNovaRole() {
2      $result = $this->_role->save($this->_data[0]);
3      $this->assertEquals(1, $result);
4  }
```

Figura 25 - Teste deve inserir novo papel

Na primeira etapa do TDD deve-se escrever um teste que falhe, na figura 25 o teste para testar o método “save”, que insere um dado no banco de dados foi desenvolvido. Na figura 26 é exibido o código do método que será testado.

```
1  <?php
2
3  class Application_Model_Role {
4
5      public function save() {
6
7      }
8  }
```

Figura 26 - Método save

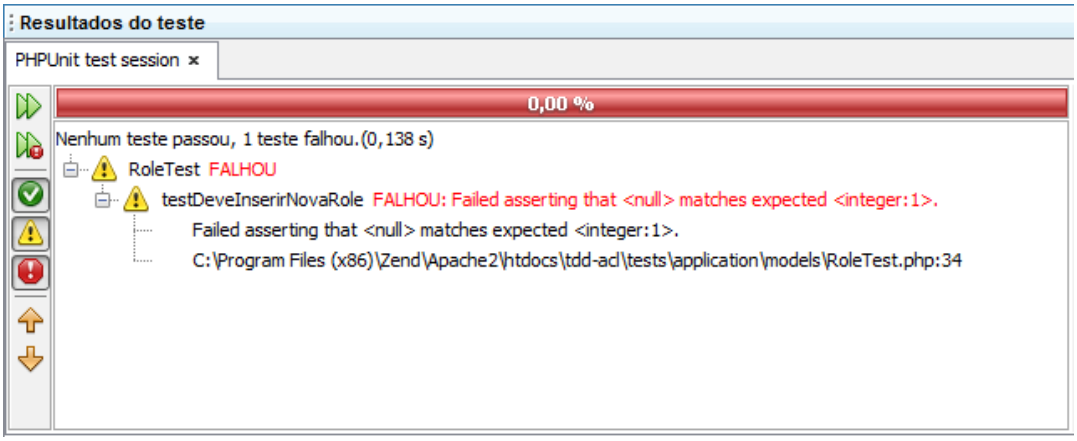

Figura 27 - Etapa 1, falha no teste

A figura 27 exibe a falha no teste, pois o método que deveria retornar o valor 1 retornou nulo, o código do método save foi desenvolvido, porém não houve retorno, assim o resultado do teste é uma falha, concluindo assim a primeira etapa do TDD.

A próxima etapa do TDD é demonstrada na figura 28, onde o método é desenvolvido para que o teste tenha sucesso da maneira mais simples possível.

```php
<?php

class Application_Model_Role {

    public function save(array $data) {
        $dt = new Zend_Date();
        $dt = $dt->toString("yyyy-MM-dd HH:mm:s");
        $data['created_at'] = $dt;
        $data['updated_at'] = $dt;

        $dbTable = new Application_Model_DbTable_Role();
        return $dbTable->insert($data);
    }
}
```

Figura 28 - Código método save

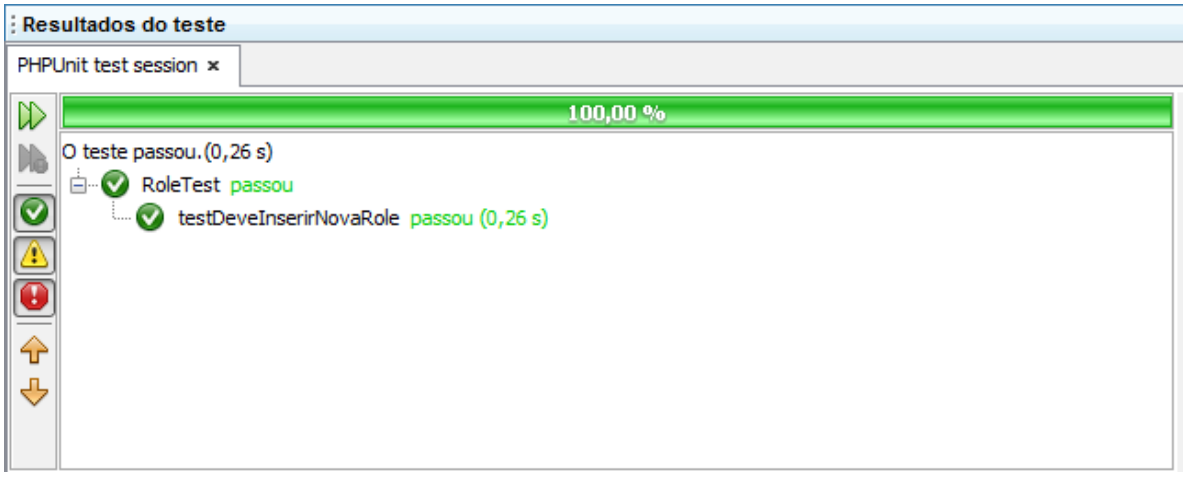

Figura 29 - Resultado teste método save

A figura 29 exibe o resultado do teste do método "*save*" após o desenvolvimento da segunda etapa do TDD, com essa modificação a etapa em questão esta concluída.

A próxima etapa do TDD é o *refactoring*, a partir desta etapa toda modificação realizada no código fonte do método "*save*" deverá resultar em um teste com sucesso, pois se anteriormente o teste verificava a saída desejada corretamente, a partir de agora qualquer modificação também deverá retornar a saída desejada, neste caso o valor 1 referente à inserção do papel no banco de dados.

Após várias refatorações no código da aplicação, o código final obtido foi o desenvolvimento de uma classe abstrata onde o método "*save*" será utilizado por toda aplicação, não somente ao módulo "Role", assim não será necessário o desenvolvimento de um método "*save*" para cada módulo do sistema, todos os módulos utilizarão a classe abstrata para diversas funcionalidades, como inserção (através do método "*save*"), atualização de dados, busca, etc.

A figura 30 exibe o módulo "Role" refatorado, este módulo agora estenda da nova classe abstrata, para que os métodos desenvolvidos na classe abstrata possam ser utilizados pelo módulo.

```php
<?php

class Application_Model_Role extends Application_Model_Abstract {

    public function __construct() {
        $this->_dbTable = new Application_Model_DbTable_Role();
        $this->_table = "roles";
    }
}
```

Figura 30 - Model Role refatorado

```php
abstract class Application_Model_Abstract {

    protected $_dbTable;
    protected $_table;

    public function save(array $data) {
        $dt = new Zend_Date();
        $dt = $dt->toString("yyyy-MM-dd HH:mm:s");
        $data['created_at'] = $dt;
        $data['updated_at'] = $dt;

        if (isset($data['id'])) {
            return $this->_update($data);
        } else {
            return $this->_insert($data);
        }
    }
```

Figura 31 - Método save classe abstrata

A figura 31 exibe o método “*save*” na classe abstrata após a refatoração do sistema, além de inserir no banco de dados o método refatorado identifica ainda se o usuário efetuará uma inserção no banco de dados, ou uma atualização através do campo “id” do array data.

Após todas essas modificações que influenciaram na refatoração de grande parte da aplicação o teste de inserir novo papel ainda deverá ter sucesso, pois como a saída esperada já foi adquirida, a partir das modificações realizadas a saída ainda deverá ser a mesma, caso a saída dos testes seja diferente da anterior o teste falhará, assim o desenvolvedor identificará que um erro foi inserido no código, podendo assim resolve-lo rapidamente.

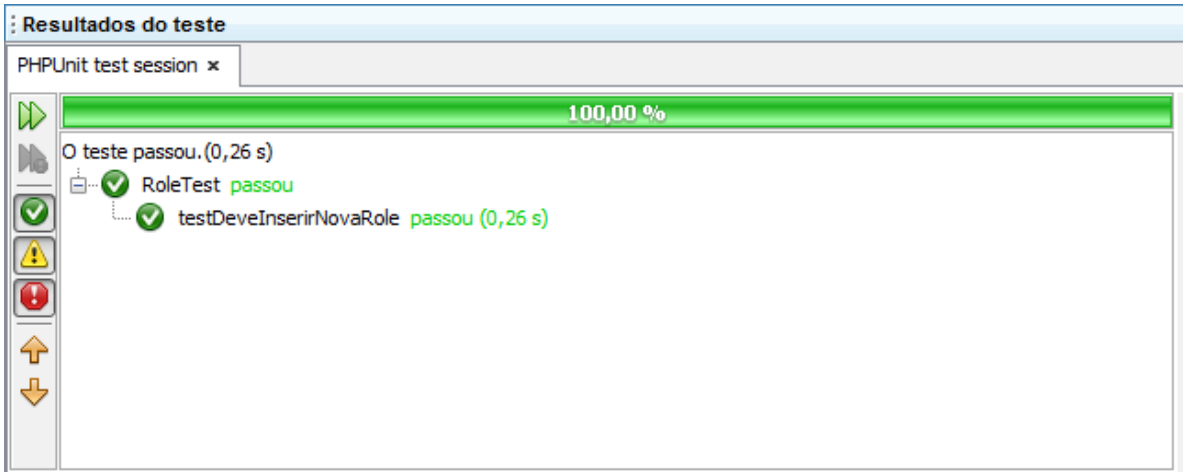

Figura 32 - Resultado teste após refatoração

A figura 32 exibe o resultado do teste “testDeveInserirNovaRole” após a refatoração do código, como o teste resultou na saída desejada, que também foi obtida antes da refatoração, o mesmo se dá como válido, com esta verificação o desenvolvedor tem certeza que nenhum erro foi inserido no código, mesmo após as melhorias realizadas no mesmo, o TDD fornece um *feedback* que dá ao desenvolvedor uma maior segurança em relação a entrega desse código para um novo *release* da aplicação.

No início do desenvolvimento da aplicação vários testes foram realizados, alguns deles estão exibidos na figura 33:

```
53      /**
54       * @dataProvider existsProvider
55       */
56      public function testVerificaExistenciaDeActions($action) {
57          $this->dispatch("/admin/privilege/".$action);
58          $this->assertModule("admin");
59          $this->assertController("privilege");
60          $this->assertAction($action);
61      }
62
63      public function testVerificaExistenciaDoFormulario() {
64          $this->dispatch("/admin/privilege/new");
65          $this->assertContains("<form", $this->getResponse()->getBody());
66      }
67
68      public function testVerificaInsercaoDeUmPrivilege() {
69          $this->getRequest()->setPost($this->_data[0]);
70          $this->getRequest()->setMethod("POST");
71          $this->dispatch("/admin/privilege/new");
72
73          $privilege = $this->_privilege->find(1);
74          $this->assertEquals(1, $privilege->id);
75
76          $this->assertRedirect();
77      }
78
79      public function testVerificaInsercaoDeUmPrivilegeVazio() {
80          $this->getRequest()->setPost(array());
81          $this->getRequest()->setMethod("POST");
82          $this->dispatch("/admin/privilege/new");
83          $this->assertContains("errors", $this->getResponse()->getBody());
84      }
85
86      public function testVerificaCampoPreenchidoEdit() {
87          $this->_privilege->save(array('name'=>'Test','role_id'=>1,'resource_id'=>1));
88          $this->dispatch("/admin/privilege/edit/id/1");
89          $this->assertContains("Test", $this->getResponse()->getBody());
90      }
```

Figura 33 - Testes da aplicação

Quando o desenvolvedor está iniciando no desenvolvimento do *software* utilizando TDD ele desenvolve vários testes simples (testar tipos de campos, tipos de formulários, tipos de modelos, existência de actions, etc), essa prática acarreta em um maior tempo de desenvolvimento da aplicação, essa prática é chamada de *Baby Steps* (passos de bebe) (PRESSMANN, 2006).

No início do desenvolvimento da aplicação do projeto a prática de *baby steps* também foi realizada, porém com o andamento do projeto essa prática não foi mais necessária, pois a confiança nos resultados do teste fornece ao desenvolvedor a segurança de que ele pode parar de efetuar alguns testes taxados como desnecessários e que só acarretariam em uma demanda maior de tempo de desenvolvimento. Com a prática no desenvolvimento utilizando TDD, o desenvolvedor não necessita mais desenvolvedor esses testes, pois a confiança que ele tem com o *feedback* dos testes fornece ao desenvolvedor essa segurança de que todo o sistema está funcionando conforme o escopo do projeto.

Alguns outros testes efetuados na aplicação foram:

- Formulários de Papéis, Recursos e Privilégios (verificação de tipos de campos, campos vazios, existência de campos, verificação de documentos html que devem existir nos formulários)

- Models de papéis, recursos e privilégios (verificação de carregamento dos dados)

Controllers de papéis, recursos e privilégios (verificação de inserção, edição e deleção, verificação do tipo de modelo, verificação de inserções repetidas).

A figura 34 exibe o resultado dos 63 testes realizados na aplicação, todos os testes foram validados pelo PHPUnit, demonstrando assim que o código fonte da aplicação está testado e validado, todas as saídas desejadas foram obtidas com sucesso conforme os testes desenvolvidos.

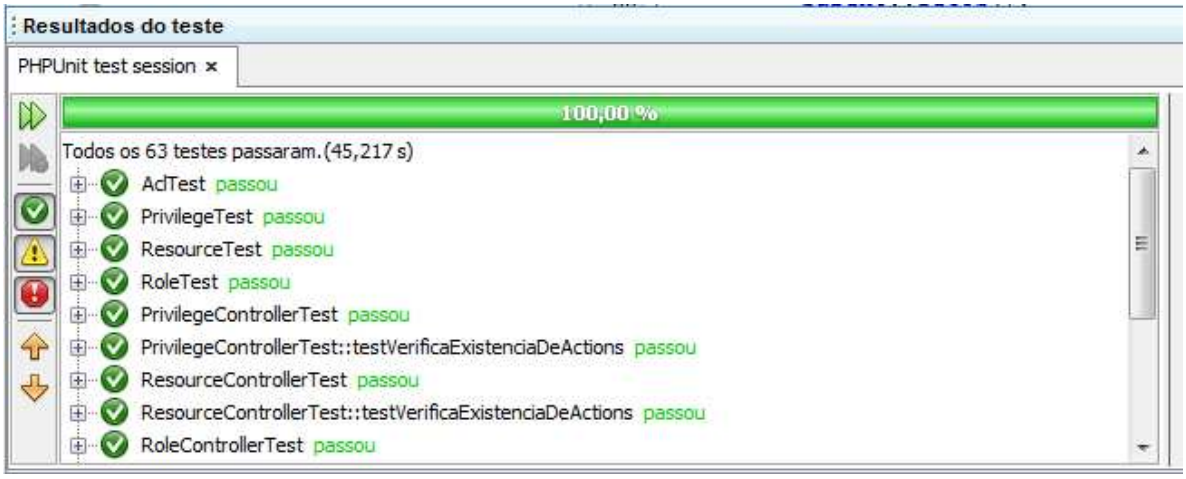

Figura 34 - Resultado todos os testes da aplicação

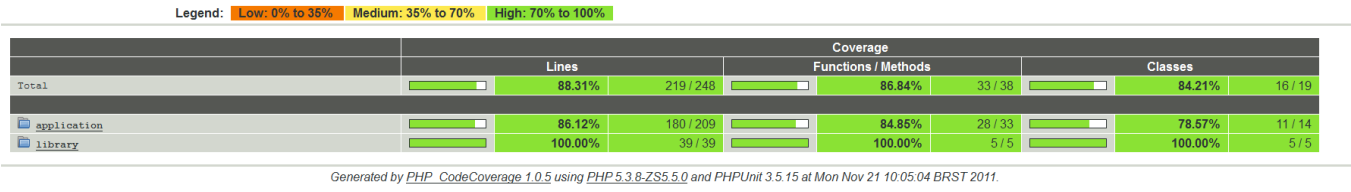

Figura 35 - Total de testes

Uma ferramenta de auxílio para o PHP, chamada XDebug, auxilia o desenvolvedor no *debug* de seu código de uma forma simples, em conjunto com o PHPUnit o XDebug informa ao desenvolvedor a porcentagem de quanto seu código está testado. A figura 35 exibe o valor

da porcentagem para este projeto, no total a pasta “*application*” teve 86,12% de seu código fonte testado, já a pasta “*library”* teve 100% de seu código fonte testado, ao final todo o projeto teve 88,31% de seu código fonte testado e aprovado.

Essa ferramenta pode ser utilizada como um relatório para os gestores do projeto, informando como está o andamento do projeto e como está a parte de testes do mesmo.

Um código 100% testado não significa que estará sem erros, isso significa que todos os testes realizados estão sem erros, mas o sistema como um todo ainda pode conter erros, pois se algo que estava no escopo não foi desenvolvido ainda significa que o sistema está errado, portanto cabe ao desenvolvedor verificar se tudo o que estava no escopo do projeto foi devidamente testado e desenvolvido.

Sem dúvida a realização de testes é uma prática muito bem vista, que produz resultados em curto, médio e longo prazo. Evita surpresas na execução dos sistemas; reduz o tempo gasto com *debug*; auxilia em um possível processo de reengenharia e adaptação do *software*; aumenta a qualidade do produto final; e muitos outros benefícios que são computados no fim do processo de desenvolvimento.

CONCLUSÕES

Utilizando técnicas como refactoring, programação em par, integração contínua, testes unitários e outras, o TDD dá como sucesso projetos que a utilizam como metodologia de desenvolvimento. Diversas empresas já adotaram o TDD como padrão no desenvolvimento de software, pois aumentaram o nível de compreensão do código gerado economizando tempo de manutenção.

Com o uso desta técnica é possível reduzir a complexidade do software, aumentando a manutenibilidade do mesmo. Falhas são facilmente identificadas ainda na etapa de desenvolvimento graças ao contínuo *feedback* dado ao programador. As unidades de teste criadas permitem avaliar mais facilmente novos defeitos que podem ter sido inseridos no software facilitando o desenvolvimento de sucessivos releases. A prática TDD também integra as diferentes fases do desenvolvimento: projeto, implementação e teste. É dada maior ênfase em como os elementos necessitam ser implementados e menor ênfase nas estruturas lógicas. Portanto, se adotado fielmente, o TDD pode resultar na falta do esboço do sistema. Decisões importantes de projeto podem ser perdidas devido à falta da documentação formal do projeto.

Além disso todos os testes efetuados no sistema servem como documentação para o mesmo, o próximo desenvolvedor que efetuar mudanças no sistema terá uma ampla documentação sobre o sistema, assim terá confiança em efetuar tais mudanças com um feedback real de possíveis erros inseridos em outras partes do software, o tempo em que ele efetuará a mudança será muito menor, pois com o sistema todo documentado e bem organizado, basta o desenvolvedor efetuar suas mudanças e rodar os testes novamente, assim ele terá um real feedback do sistema, podendo assim soltar um novo release do software.

O uso do TDD diminui muito o custo com a manutenção do software após o mesmo ser entregue para o cliente, correções de bugs, inserção de novas funcionalidades serão realizados em um tempo muito menor.

O tempo de desenvolvimento utilizando o TDD para um desenvolvedor novo na metodologia será maior, porém após um tempo utilizando a metodologia o desenvolvedor ficará bem mais ágil, aumentando assim sua produtividade e a qualidade de seus testes, ocasionando assim o aumento da qualidade do software final.

A entrega contínua garante uma maior satisfação por parte do cliente e do gestor do projeto, pois com ela a empresa estará presente junto ao cliente no processo de desenvolvimento, o cliente terá uma maior tranqüilidade quanto a prazo de entrega e confiança junto à empresa que está sempre em contato lançando novos módulos do sistema em diversos momentos.

No final do projeto o uso do TDD se dá como muito válido, pois o tempo com manutenção será bem menor, satisfação do cliente a cada entrega de módulos (devido à entrega contínua) será maior, além da maior confiança dos desenvolvedores em seu código dão ao software final uma qualidade muito mais significativa do que um software que não foi desenvolvido utilizando o TDD.

REFERÊNCIAS

AGILE DATA. Disponível em http://www.agiledata.org/essays/tdd.html

B. GEORGE, L. WILLIAMS, An Initial Investigation of Test Driven Development In Industry. Disponível em http://http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf

BERGMANN, Sebastian. (2011) "PHPUnit site oficial", http://www.phpunit.de , Fevereiro 2011.

BECK, K. "eXtreme Programming Explained", Addison Wesley, 2000.

BECK, K. Test-Driven Development By Example. Addison Wesley, 2002. Estados Unidos.

BORGES, Eduardo N. Conceitos e Benefícios do Test Driven Development. Universidade Federal do Rio Grande do Sul / Instituto de Informática, 2006.

C. G. JONES, Test-Driven Development Goes to School, 2004. Disponível em http://portal.acm.org/citation.cfm?id=1040261&dl=ACM&coll=GUIDE&CFID=15151515&CFTOKEN=6184618

FEITOSA, Daniela Soares. Um estudo sobre o impacto do uso de desenvolvimento

orientado por testes na melhoria da qualidade de software. Universidade Federal da

Bahia / Instituto de Matemática, 2007.

FOWLER, Martin (2010) "Martin Fowler site oficial", http:// http://martinfowler.com/refactoring/, Dezembro 2010.

GASPARETO, Otavio. Test Driven Development. Universidade Federal do Rio Grande do Sul / Instituto de Informática, 2006.

GOLDMAN, Dan B, Refactoring Tools for Extreme Programming: An Overview, 2002. Disponível em http://www.cs.washington.edu/education/courses/cse503/02wi/papers/goldman.pdf

H. BAUMEISTER, M. WIRSING, Applying Test-First Programming and Iterative Development in Building an E-Business Application. Disponível em http://www.pst.informatik.uni-muenchen.de/ projekte/caruso/ssgrr2002w.pdf

J. LANGR, Evolution of Test and Code Via Test-First Design. Disponível em http://www.objectmentor.com/resources/articles/tfd.pdf

M. Müller, F. Padberg, About the Return on Investment of Test-Driven Development. Disponível em http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf.

MINETTO, Elton Luís. Frameworks para Desenvolvimento em PHP. Chapecó : Novatec Editora Ltda., 2007.

PÁDUA Filho, Wilson. Engenharia de Software: Fundamentos, Métodos e Padrões. 2ª.

ed. Rio de Janeiro: LTC-Livros Técnicos e Científicos Editora S.A., 2003.

PEDRYCZ, Witold; PETERS, J. F. Engenharia De Software. 3ª. ed. Rio de Janeiro: Elsevier, 2001.

PRESSMAN, R. S. Engenharia de Software. 6ª. ed. São Paulo: MacGrow-Hill, 2006.

R. KAUFMANN, D. JANZEN, Implications of Test Driven Development A Pilot Study, 2003. Disponível em http://people.bethelks.edu/djanzen/Papers/pos12-kaufmann.pdf

R. MUGRIDE, Test Driven Development and the Specifc Method, 2003. Disponível em http://www.agiledevelopmentconference.com/2003/files/P6Paper.pdf

SANTOS (2009) “USantos site oficial”, http://usantos.wordpress.com, Setembro 2009.

S. FREEMAN, T. MACKINNON, N. PRYCE, Mock Roles, Not Objects. Disponível em

http://www.jmock.org/oopsla2004.pdf

W. MARRERO, A. SETTLE, Testing First: Emphasizing Testing in Early Programming Courses (2005). Disponível em http://portal.acm.org/ft_gateway.cfm?id=1067451&type=pdf&coll=GUIDE&dl=GUIDE&CFID=69236573&CFTOKEN=4183298.