



**ETEP - FACULDADE DE TECNOLOGIA
DE SÃO JOSÉ DOS CAMPOS**

**PROPOSTA DE SISTEMA DE CATÁLOGO DIGITAL UTILIZANDO
METODOLOGIA BASEADA EM TESTES**

João Paulo Gomes dos Santos

Trabalho de Conclusão de Curso de Bacharelado em Engenharia da Computação, orientado pelo Prof. Edizon Basseto Júnior.

ETEP Faculdades
São José dos Campos

2011

ETEP - FACULDADE DE TECNOLOGIA DE SÃO JOSÉ DOS CAMPOS

**PROPOSTA DE SISTEMA DE CATÁLOGO DIGITAL UTILIZANDO
METODOLOGIA BASEADA EM TESTES**

João Paulo Gomes dos Santos

Edizon Basseto Júnior

ETEP Faculdades
São José dos Campos
2011

“A dúvida é o principio da sabedoria”.

Aristóteles

AGRADECIMENTOS

Agradeço principalmente a Deus que me concedeu a capacidade, força e perseverança para o desenvolvimento e conclusão deste projeto, pois sabemos que sem ele não somos nada, e nada, poderíamos fazer.

Também agradeço à minha namorada Ana Cláudia Narumi Kameda e aos meus familiares, que me apoiaram na elaboração deste projeto. Gostaria de exprimir os mais sinceros agradecimentos e aqui reconhecer a sua importante contribuição.

Aos meus amigos de classe, pelo apoio incondicional e pela grande amizade que têm dedicado e que nunca pouparam esforços para me ajudar. Aos professores, em especial ao professor Edizon Basseto Júnior pela disponibilidade e força de vontade de orientar, sendo os educadores da ETEP Faculdades os que sempre me incentivaram.

Enfim, agradeço as todas as pessoas que contribuíram de forma direta ou indireta para realização do meu trabalho e acreditando na minha capacidade de realizar esse projeto.

RESUMO

Nos últimos anos com a crescente demanda tecnológica o hardware deixou de ser alvo exclusivo de pesquisas e aprimoramento, sendo a interação homem máquina o mais valioso item de um produto. Isso é fácil de verificar no mercado com a crescente revolução dos dispositivos portáteis e seus sistemas operacionais. Com isso a importância do mercado de desenvolvimento de software vem aumentando assim como a concorrência entre as empresas que desenvolvem os sistemas e aplicativos.. Essas empresas buscam entregar software com qualidade e com menores custos para seus clientes. Para atingir tal objetivo foram criadas práticas para otimização do processo de desenvolvimento de software, tendo como objetivo a redução de custos e aumento da qualidade do produto final. Este trabalho mostra uma maneira de combinar princípios ágeis de desenvolvimento de software de modo a se obter uma melhoria no processo de desenvolvimento de software. O trabalho utiliza como base o pensamento *Lean* e práticas de *Extreme Programming* como o *Test Driven Development*.

Palavras Chave: *Lean Thinking; Extreme Programming; Test Driven Development;*

SUMÁRIO

	<u>Pág.</u>
LISTA DE FIGURAS.....	8
LISTA DE SÍMBOLOS	10
1 INTRODUÇÃO	11
1.1 ESTUDO DE CASO	12
1.2 OBJETIVO DO TRABALHO.....	13
1.3 JUSTIFICATIVA	13
1.4 ESTRUTURA DO TRABALHO	14
1.4.1 FUNDAMENTAÇÃO TEÓRICA	14
1.4.2 METODOLOGIA.....	14
1.4.3 RESULTADOS	14
2 FUNDAMENTAÇÃO TEÓRICA.....	15
2.1 TESTES DE SOFTWARE	15
2.1.1 TESTES DE UNIDADE	16
2.1.2 TESTES DE INTEGRAÇÃO.....	16
2.2 TESTES AUTOMATIZADOS	17
2.3 LINGUAGEM DE PROGRAMAÇÃO RUBY	18
2.4 PADRÃO ARQUITETURAL MVC.....	19
2.5 FRAMEWORK DE DESENVOLVIMENTO RAILS.....	20
2.6 DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS.....	21
2.6.1 JAVA MICRO EDITION (J2ME)	21
2.6.2 IPHONE.....	22
2.6.3 ANDROID.....	23
2.7 METODOLOGIAS DE DESENVOLVIMENTO DE SOFTWARE.....	25
2.7.1 CASCATA.....	25
2.7.2 RATIONAL UNIFIED PROCESS (RUP).....	26
2.7.3 PROGRAMAÇÃO EXTREMA (XP)	27
2.7.4 LEAN.....	28

2.7.5 TEST DRIVEN DEVELOPMENT (TDD)	30
3 METODOLOGIA.....	32
3.1 METODOLOGIA UTILIZADA NO DESENVOLVIMENTO.....	33
3.2 DESENVOLVIMENTO DO SISTEMA WEB	34
3.2.1 TECNOLOGIAS UTILIZADAS	35
3.2.2 DESENVOLVIMENTO.....	38
3.3 DESENVOLVIMENTO DO SISTEMA MOBILE	41
3.3.1 TECNOLOGIAS UTILIZADAS	42
3.3.2 DESENVOLVIMENTO.....	42
4 RESULTADOS	45
4.1 SISTEMA WEB	45
4.2 SISTEMA MÓVEL.....	46
5 CONCLUSÃO.....	47

LISTA DE FIGURAS

	<u>Pág.</u>
Figura 2.1 - Arquitetura da Plataforma Android.	23
Figura 2.2 – Ciclo básico para TDD.....	31
Figura 3.1 – Diagrama em blocos do sistema.....	32
Figura 3.2 – Diagrama de Casos de Uso do Sistema Web.	34
Figura 3.3 – Trecho de um código escrito usando RSpec.	36
Figura 3.4 – Resultado da execução do Autotest.....	37
Figura 3.5 – Classe de teste da classe Categoria	39
Figura 3.6 – Classe Categoria.....	40
Figura 3.7 – Diagrama de Casos de Uso do Sistema Mobile	41
Figura 3.8 – Trecho de código de teste.....	43
Figura 4.1 – Relatório de cobertura do sistema Web	45
Figura 4.2 – Relatório de cobertura do sistema móvel.....	46

LISTA DE SÍMBOLOS

TDD - Test Driven Development

SUCESU - Sociedade dos Usuários de Informática e Telecomunicações

HTTP - Hypertext Transfer Protocol

HTML - Hypertext Markup Language

XML - Extensible Markup Language

JSON - Javascript Object Notation

J2ME - Java Micro Edition

RUP - Rational Unified Process

UP - Unified Process

XP - Extreme Programming

MVC - Model View Controller

AWS - Amazon Web Service

API - Application Programming Interface

1 INTRODUÇÃO

A Toyota foi o berço para o surgimento do pensamento *Lean*. Logo após a segunda guerra mundial, o Japão estava destruído e as empresas, precisando se reerguer, tinham uma produtividade muito baixa e os recursos para produção eram escassos. Em frente a esse cenário Taiichi Ohno e Shigeo Shingo desenvolveram o pensamento *Lean* (*Lean Thinking*), justamente para aumentar e otimizar a produção da empresa (OHNO, 1988).

O modelo utilizado na Toyota se tornou um sucesso, foi adaptado e adotado em muitos outros segmentos, virando referência quando se fala em redução de custos e em otimização da produção (OHNO, 1988).

Em 2003, Mary Poppendieck escreveu o livro *Lean Software Development: An Agile Toolkit* que mostra como o pensamento *Lean* pode ser adaptado para o desenvolvimento de software. Este livro pode ser considerado um marco para as empresas que desenvolvem software, pois com base nele muitas empresas puderam conhecer e aplicar os princípios do pensamento *Lean* e assim conseguiram aumentar seus desempenhos (POPPENDIECK, 2003).

Como um dos principais pilares do pensamento *Lean* temos a redução de desperdício, essa redução de desperdício pode ser alcançada com a adoção de algumas práticas de *Extreme Programming* (POPPENDIECK, 2003), conhecido pela sigla (XP), que terá suas características detalhadas nos próximos capítulos desse trabalho.

Extreme Programming contempla uma série de práticas para o sucesso no desenvolvimento de um software, uma dessas práticas é fazer com que o *feedback* do sistema seja rápido.

Para se obter um *feedback* suficientemente veloz podemos utilizar uma prática conhecida como *Test Driven Development* (**TDD**), que diz que os testes para uma determinada funcionalidade do software devem ser criados antes mesmo do desenvolvimento da mesma, assim pode-se ter resultados praticamente imediatos a respeito do que foi implementado para atender à funcionalidade em questão (FREEMAN, 2009).

Cada vez mais o mercado de desenvolvimento de software está valorizando práticas presentes no *Extreme Programming*, alguns dos principais motivos para essa valorização são o aumento da qualidade do resultado e a redução de custos durante o desenvolvimento.

A princípio, quando o **TDD** é incluído no processo de desenvolvimento de software, os desenvolvedores tendem a pensar que o ciclo de desenvolvimento foi aumentado e com

isso o tempo gasto para desenvolver uma funcionalidade aumenta também, segundo (FREEMAN, 2009) o que ocorre é exatamente o contrário. Quando criamos testes para uma funcionalidade somos obrigados a pensar como ela será implementada e assim conseguimos descobrir problemas do nosso design antecipadamente, reduzindo o tempo gasto na correção de defeitos.

Outra vantagem do uso de **TDD** é a possibilidade de agrupar os testes gerados por todas as funcionalidades já implementadas e executá-los em conjunto, garantindo que nenhuma funcionalidade existente seja afetada por novas implementações (FREEMAN, 2009).

Este trabalho tem como objetivo mostrar o uso de **TDD** aplicado em um sistema de exemplo, onde se pode observar as vantagens e a velocidade do *feedback* proporcionado por essa metodologia de desenvolvimento.

Mais informações sobre o pensamento *Lean*, *Extreme Programming* e **TDD** serão apresentadas nos próximos capítulos.

Para tornar mais claros os benefícios do uso do pensamento *Lean* e de práticas de *Extreme Programming* para o desenvolvimento de software, será utilizado um estudo de caso, que será explicado com mais detalhes no próximo item deste trabalho.

1.1 ESTUDO DE CASO

Cada dia que passa as empresas tecnológicas estão investindo mais em tecnologias móveis.

Ultimamente, a interação homem máquina vem sendo mais valorizada que evoluções de hardware. Diante deste fato os *tablets* ganharam um lugar de destaque entre os dispositivos móveis. Os *tablets* mais modernos são praticamente computadores, fazem tudo que um computador convencional faz, apenas com uma diferença: são bem menores.

Inicialmente os *tablets* eram muito caros, impossibilitando a adoção pelo público. Com o passar do tempo o custo benefício dos aparelhos cresceu muito, fato que impulsionou a venda e criou muitas oportunidades de negócio.

Hoje existem várias empresas que são especialistas em desenvolvimento de software para dispositivos móveis, devido à grande demanda das empresas consumidoras destes aplicativos.

Este trabalho utilizará como estudo de caso um sistema de catálogo digital que tem seu funcionamento descrito abaixo.

A primeira parte do projeto é composta por um sistema web capaz de armazenar os dados do cliente. Depois que o cliente realizar o cadastro de seus dados ele poderá visualizá-los em *tablets*. Os *tablets* serão capazes de sincronizar seus dados com os dados cadastrados no sistema web.

Um exemplo de aplicação deste projeto poderia ser um restaurante, onde os *tablets* podem ser usados como cardápios. Com base nos dados que foram cadastrados no sistema web os *tablets* seriam atualizados, reduzindo o custo e o tempo para a atualização dos cardápios.

1.2 OBJETIVO DO TRABALHO

O primeiro objetivo do trabalho é propor uma metodologia para desenvolvimento de software utilizando uma abordagem baseada em testes automatizados, mostrando ferramentas para testes e vantagens do uso de testes no desenvolvimento de software.

O segundo objetivo do trabalho é mostrar a integração de sistemas web com sistemas desenvolvidos para dispositivos móveis.

1.3 JUSTIFICATIVA

Este tema foi escolhido pela necessidade de aumento de qualidade que existe no processo de desenvolvimento de software, principalmente no Brasil. Segundo (SUCESU-RS, 2011) em 2008 o mercado de desenvolvimento de software global movimentou cerca de 1,23 trilhões de dólares, com um crescimento médio de 3% ao ano. Apesar desta grande movimentação financeira em torno de projetos de software, as empresas investem pouco na qualidade de seus produtos. Qualquer otimização gerada neste processo de desenvolvimento pode ser vista como economia.

1.4 ESTRUTURA DO TRABALHO

Este trabalho foi dividido nas seguintes partes:

- Fundamentação Teórica;
- Metodologia;
- Resultados;
- Conclusão;

1.4.1 FUNDAMENTAÇÃO TEÓRICA

Na fundamentação teórica deste trabalho serão apresentados todos os conceitos que servem como base para o trabalho.

Alguns tópicos abordados nesta fase estão listados abaixo:

- A importância dos testes de software;
- Os tipos de testes de software;
- A linguagem de programação Ruby e suas ferramentas;
- Desenvolvimento para dispositivos móveis nas plataformas Android, iPhone e J2ME;
- Metodologias para desenvolvimento de software;
- *Test Driven Development*

1.4.2 METODOLOGIA

Na metodologia será mostrado como o software foi desenvolvido e como o *Test Driven Development* foi empregado.

1.4.3 RESULTADOS

Na etapa de resultados serão mostrados relatórios de cobertura de testes obtidos após o desenvolvimento do sistema, comprovando a eficácia da metodologia.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 TESTES DE SOFTWARE

Como em qualquer área de desenvolvimento, os testes são essenciais para aumentar a qualidade do produto final, no desenvolvimento de software não é diferente.

A definição de teste de software segundo (MYERS, 2004) diz que teste de software é o processo de executar um software com o intuito de encontrar erros.

Devido à natureza dinâmica dos softwares não é possível garantir que eles fiquem livres de erros (MYERS, 2004), mesmo passando por exaustivas baterias de testes.

Segundo (MCGREGOR, 2001) e (GUERRA, 2005), existem vários tipos de testes de software, cada um com um propósito específico, alguns dos principais são:

- **Teste de unidade:** O teste deve ser feito sobre uma unidade do software, ou seja, sobre uma classe ou método. Testes de unidade ajudam a garantir que a unidade a ser testada está desacoplada das outras unidades e coesa. Seu principal objetivo é verificar se a unidade possui o comportamento esperado.
- **Teste de Interação:** O teste deve ser feito sobre um grupo de classes do software e devem verificar se a troca de mensagens entre as classes ocorre corretamente.
- **Teste de Componente:** O teste deve ser feito sobre um grupo de classes e devem ser verificadas as entradas e as saídas do componente.
- **Teste de Integração:** Deve ser feito sobre um grupo de componentes.
- **Teste de Sistema:** Deve ser realizado utilizando o software como um todo, usando a visão do usuário final. Este teste ajuda na verificação dos requisitos do software.
- **Teste de Validação:** Assim como o teste de sistema, este teste deve utilizar o software como um todo, tendo como única diferença a necessidade de acompanhamento do cliente durante os testes.

A execução dos testes de forma manual é muito custosa para grandes softwares e assim a execução de todos os testes a cada nova versão do software acaba se tornando inviável para softwares de grande porte (GUERRA, 2005).

A qualidade do software diminui à medida que a qualidade dos testes diminui, ou seja, quanto menor a eficácia dos testes do software menor será a qualidade do produto final, já que a garantia de que tudo esteja funcionando corretamente irá diminuir (GUERRA, 2005).

Nos próximos capítulos serão detalhados os principais tipos de teste utilizados neste trabalho.

2.1.1 TESTES DE UNIDADE

A unidade fundamental de um sistema orientado a objetos é denominada classe, o teste de unidade de uma classe tem como objetivo principal verificar se a implementação da classe corresponde ao que foi especificado para a mesma (MCGREGOR, 2001).

Garantindo que todas as classes estão cobertas por testes de unidade, seguindo suas especificações, haverá uma grande chance de que qualquer problema no sistema desenvolvido foi causado por erros de integração entre as unidades.

Segundo (MCGREGOR, 2001), o tempo gasto com correção de problemas no sistema é reduzido drasticamente, já que as unidades são coesas e testadas corretamente.

Pode-se perceber uma grande redução no tempo gasto com correção de defeitos no sistema em que os testes unitários são adotados. Outra vantagem é a melhoria no design do sistema, ou seja, os componentes do sistema deve ser coesos e estar desacoplados uns dos outros para que seja possível a criação de testes de unidade (HUNT, 2003).

2.1.2 TESTES DE INTEGRAÇÃO

Um sistema orientado a objetos é composto por um conjunto de objetos que colaboram entre si para atingir um objetivo. O modo com que estes objetos interagem entre si determina o resultado do sistema (MCGREGOR, 2001).

Um exemplo seria o caso de um sistema em que todas as unidades tem o comportamento correto quando isoladas, mas quando colocadas para interagirem entre si o objetivo do sistema não é atingido, ou seja, existe um problema de integração entre as unidades (MCGREGOR, 2001).

Segundo (MCGREGOR, 2001) o principal objetivo de um teste de integração é garantir que as mensagens enviadas de um objeto para outro sejam executadas de maneira correta.

Antes de realizar testes de integração deve-se garantir que as unidades participantes do teste estão cobertas por testes de unidade.

A segurança para fazer uma atualização em um sistema aumenta proporcionalmente a medida que a cobertura de testes de integração aumenta, ou seja, os testes de integração ajudam a garantir que as funcionalidades do sistema estão de acordo com os requisitos.

Outra vantagem dos testes de integração é a garantia de que uma alteração não danificou alguma funcionalidade que estava correta.

2.2 TESTES AUTOMATIZADOS

Os testes são parte importante do desenvolvimento de um sistema, normalmente eles são executados de forma manual, ou seja, dependem da interação humana para serem realizados. A execução dos testes desta maneira é muito custosa, já que consome muito tempo durante sua execução.

Depois de desenvolver os testes alguém precisa disparar a execução do conjunto de testes que foi desenvolvido, deixando o *feedback* gerado pela execução dos testes dependente de alguém para ser gerado.

Uma alternativa para reduzir os custos dos testes de software é a automatização da execução dos mesmos, de forma que não seja necessária a interação humana em nenhuma parte deste processo.

Executar uma bateria de testes manuais a cada iteração do sistema se torna inviável à medida que o software cresce (GUERRA, 2005), com o intuito de contornar este problema surgiu o conceito de automação de testes.

Automatizar o processo de testes aumenta a confiabilidade do software a ser desenvolvido e garante maior agilidade no ciclo de desenvolvimento, já que os testes de regressão podem ser executados a cada iteração do ciclo de desenvolvimento (DUSTIN, 2002).

Para facilitar o processo de execução automatizada dos testes, existem ferramentas que simulam ambientes para execução dos testes continuamente, essas ferramentas são mais conhecidas como ambientes de integração contínua.

Com a execução dos testes de regressão de maneira automatizada pode-se saber rapidamente se qualquer uma das modificações efetuadas causou algum impacto indesejado em outras partes do software (DUSTIN, 2002).

Recomenda-se que a cada alteração no sistema seja criado um caso teste para cobrir a nova situação criada com a alteração em questão, depois de elaborar o caso de teste é necessário adicioná-lo ao conjunto de testes do sistema em desenvolvimento, fazendo com que o conjunto de testes automatizados esteja sempre atualizado.

2.3 LINGUAGEM DE PROGRAMAÇÃO RUBY

A linguagem de programação Ruby foi criada por *Yukihiro Matsumoto*, no Japão, no ano de 1995, desde então vem se tornando uma linguagem robusta o suficiente para ser utilizada em sistemas de qualquer natureza (MATSUMOTO, 2001).

Ruby é uma linguagem de programação totalmente orientada a objetos, já que tudo em Ruby é um objeto, sem exceções.

Muitas linguagens de programação incorporaram aspectos de programação orientada a objetos, mas poucas conseguem ser totalmente orientadas a objetos assim como Ruby (MATSUMOTO, 2001).

Como exemplo pode-se citar a linguagem de programação Java, ela é classificada como uma linguagem de programação orientada a objetos, mas existem representações de tipos primitivos (integer, double, byte, char, etc) em Java, ou seja, esses tipos primitivos não são objetos, sendo assim Java não é uma linguagem totalmente orientada a objetos (SIERRA, 2008). Em Ruby até os inteiros são objetos da classe FixNum.

Segundo (MATSUMOTO, 2001), quando a linguagem Ruby foi desenvolvida o principal foco de seu criador era gerar uma linguagem que pudesse aumentar a produtividade dos desenvolvedores de forma fácil, com base nessa necessidade do criador da linguagem, Ruby adquiriu algumas características, descritas abaixo (MATSUMOTO, 2001):

- Programação Interativa: Ruby é uma linguagem de script, ou seja, não é necessário compilar o código. Existe um interpretador para facilitar o desenvolvimento.
- Programação Dinâmica: Praticamente tudo que é feito em Ruby é feito em tempo de execução. Os tipos das variáveis, expressões, classes e definições de métodos são determinados em tempo de execução. Uma característica interessante é a possibilidade de alteração de suas classes em tempo de execução de maneira totalmente dinâmica, garantindo grande flexibilidade no desenvolvimento.

- **Sintaxe Familiar:** A sintaxe da linguagem Ruby é muito parecida com a sintaxe de linguagens renomadas, como por exemplo, Java, Perl, Python, C/C++, essa característica ajuda na disseminação da linguagem.
- **Bibliotecas de classes:** Ruby possui uma grande quantidade de bibliotecas que já vem com a distribuição padrão e cobrem um vasto domínio de necessidades, começando pelos tipos básicos (strings, arrays, hashes) e indo até tópicos mais avançados, como programação para recursos de rede e threads. Mesmo possuindo uma grande quantidade de bibliotecas nativas, é possível adicionar novas bibliotecas desenvolvidas por terceiros.
- **Portabilidade:** Programas escritos em Ruby podem ser rodados em qualquer ambiente computacional que possua um interpretador Ruby, ou seja, é possível criar programas Ruby em uma plataforma e migrá-los para outra sem a necessidade de nenhuma modificação.

2.4 PADRÃO ARQUITETURAL MVC

Em 1979, *Trygve Reenskaug* desenvolveu um padrão arquitetural para desenvolvimento de aplicativos, padrão conhecido como Modelo – Visão – Controlador (MVC). Como o próprio nome sugere esse padrão arquitetural divide a arquitetura das aplicações em 3 camadas básicas (RUBY, THOMAS e HANSSON, 2010):

- **Modelo:** O modelo é responsável por representar os estados dos objetos da aplicação. O estado dos objetos pode ser considerado transiente, quando o estado é mantido apenas por algumas interações, ou persistente, quando o estado é gravado em um mecanismo de persistência, como os banco de dados. O modelo é muito mais que apenas dados, ele assegura algumas regras de negócio relacionadas com os dados representados por ele.
- **Visão:** A visão é responsável por gerar a interface com o usuário, baseando-se nos dados do modelo.
- **Controlador:** Os controladores são responsáveis por coordenar o funcionamento da aplicação, eles recebem os eventos gerados pela visão, interagem com os modelos e respondem aos eventos delegando o fluxo para outras visões.

2.5 FRAMEWORK DE DESENVOLVIMENTO RAILS

Rails é um framework para desenvolvimento de aplicações para web escrito em Ruby. Criado por *David Heinemeier Hansson*, tem como seu principal objetivo facilitar o desenvolvimento de aplicações para web, levando em consideração alguns aspectos que os desenvolvedores precisam conhecer para começar a desenvolver uma aplicação (RUBY, THOMAS e HANSSON, 2010).

Os principais conceitos utilizados na criação do Rails foram:

- Convenção ao invés de configuração: Principal fundamento do framework, para tudo no Rails existe uma convenção, como por exemplo:
 - Estrutura de diretórios da aplicação;
 - Padrão de nomes dos controladores, sempre seguindo o nome dos modelos;
 - Padrão de nomes das visões, sempre seguindo os nomes das ações dos controladores;
 - Padrão de nomes de tabelas do banco de dados, sempre o nome do modelo no plural;Conhecer e obedecer as convenções é muito importante para aproveitar todas as facilidades do framework.
- Não se repita: Grande facilidade para reuso de código através de *plugins*. Alguns dos *plugins* mais populares para o desenvolvimento com Rails são:
 - Devise: É capaz de criar uma estrutura completa para autenticação de usuários;
 - Will Paginate: Serve para paginar os itens nas listagens;
 - PaperClip: Serve para controlar upload de arquivos;

A estrutura interna do Rails é dividida nos seguintes componentes (FERNANDEZ, 2010):

- *Action Controller*: Cuida do gerenciamento dos controles das aplicações. Processa as requisições HTTP, extrai os parâmetros e faz o encaminhamento para a ação desejada. Outros serviços provenientes do *Action Controller* são:
 - Gerenciamento de sessões HTTP;

- Renderização de templates;
- Redirecionamentos.
- *Action Dispatch*: É responsável pelo roteamento das requisições HTTP.
- *Action View*: É responsável pela geração das respostas para as requisições feitas, por padrão Rails tem suporte a html, xml e json.
- *Action Mailer*: É utilizado para gerenciar o envio e recebimento de e-mails.
- *Active Record*: É a base para os modelos nas aplicações Rails. Algumas de suas características são:
 - Possibilita a independência de uma banco de dados específico;
 - É capaz de fazer buscas avançadas;
 - Possibilita a criação de relacionamentos entre os modelos.
- *Active Support*: Classes utilitárias usadas por todo o framework Rails.

2.6 DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Segundo (FLING, 2009), o telefone foi uma das maiores invenções da humanidade, ele revolucionou as comunicações, permitindo que as pessoas, mesmo distantes, conseguissem se comunicar de maneira rápida e fácil.

Hoje em dia os telefones representam dispositivos com muitas funcionalidades, com um telefone é possível fazer ligações, mandar mensagens de texto, navegar na internet, jogar, acessar mapas, ouvir músicas, assistir vídeos e utilizar aplicações em geral.

Graças à evolução do hardware para dispositivos móveis é possível, cada vez mais, adicionar funcionalidades aos telefones atuais.

Com essa evolução dos telefones, o desenvolvimento de aplicativos para plataformas móveis vem se popularizando, algumas das principais tecnologias utilizadas para desenvolvimento de aplicações móveis serão introduzidas a seguir.

2.6.1 JAVA MICRO EDITION (J2ME)

Java Micro Edition (**J2ME**) é um conjunto de tecnologias e especificações que têm como principal objetivo a criação de uma máquina virtual Java capaz de ser executada em dispositivos com recursos de hardware limitados, ideal para dispositivos móveis.

Aplicativos escritos usando J2ME podem ser executados em qualquer dispositivo que possua uma máquina virtual Java, como os dispositivos listados abaixo:

- Celular;
- Palm;
- Pager;
- Tablets;

A linguagem de desenvolvimento é puramente Java com limitações em algumas bibliotecas. O fato de ser semelhante à linguagem Java facilita o crescimento da comunidade que desenvolve para esta plataforma (RISCHPATER, 2008).

2.6.2 IPHONE

O iPhone é um dispositivo criado pela empresa Apple, é considerado um dos telefones mais sofisticados do mundo, sempre em busca da simplicidade e praticidade durante seu uso.

O desenvolvimento para iPhone é feito utilizando uma linguagem de programação chamada Objective-C e ferramentas feitas para rodarem apenas em máquinas Apple, ou seja, o desenvolvimento é mais restrito, ao contrário do desenvolvimento para J2ME que é aberto para qualquer plataforma (NEUBURG, 2011).

Segundo (PILONE, 2010), as ferramentas necessárias para o desenvolvimento de aplicações para iPhone são:

- *XCode*: Ferramenta para gerenciamento e edição de código para iPhone, com ele é possível navegar pelos diretórios do aplicativo, editar o código e depurar o código. É um ambiente de desenvolvimento completo e se integra completamente com os outros componentes necessários para o desenvolvimento de aplicativos para iPhone.
- *Interface Builder*: Ferramenta para a criação das interfaces visuais da aplicação, você pode arrastar e soltar os componentes para criar suas interfaces visuais, facilitando muito o desenvolvimento.
- *Instruments*: Ferramenta utilizada para a coleta de dados de aplicações e otimização das mesmas.
- *iPhone Simulator*: Simulador para testar o aplicativo a ser desenvolvido sem a necessidade de ter um iPhone em mãos para efetuar os testes, reduzindo o tempo de desenvolvimento.

2.6.3 ANDROID

A plataforma Android foi criada sobre o *kernel* do Linux, ou seja, foi criada baseada num sistema operacional sólido e estável. O Android tem sua arquitetura de sistema como é mostrado na FIGURA 2.1:

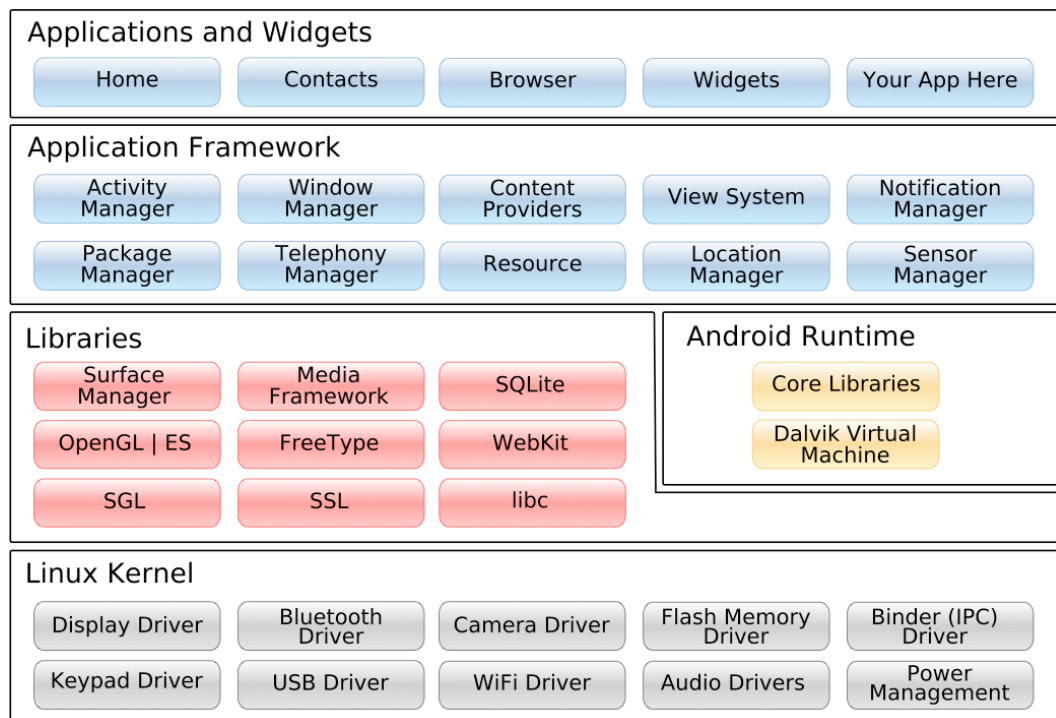


Figura 2.1 - Arquitetura da Plataforma Android.

FONTE (BURNETTE, 2010)

Internamente o Android utiliza-se de vários recursos do *kernel* do Linux:

- Gerenciamento de memória;
- Gerenciamento de processos;
- Gerenciamento de Energia;
- Acesso a rede;
- Driver de comunicação com câmera;
- Driver de comunicação com redes sem fio;

A próxima camada acima do *kernel* é composta por bibliotecas nativas do sistema, essas bibliotecas são responsáveis por executar operações básicas para as camadas superiores. Alguns exemplos de componentes presentes na camada de bibliotecas são mostrados a seguir:

- *Surface Manager*: Responsável pelo gerenciamento de janelas, possibilita a criação de diversos efeitos durante o uso do dispositivo.
- Gráficos 2D e 3D: Motor de renderização capaz de gerar gráficos em 2D e 3D em uma única interface visual. É possível utilizar recursos para renderização 3D caso o telefone utilizado seja compatível, possibilitando a execução de jogos com gráficos mais refinados.
- *Codecs* para vídeos e áudio: São responsáveis por interpretar arquivos de áudio e vídeo em diversos formatos, alguns dos mais conhecidos são: AAC, AVC (H.264), H.263, MP3 e MPEG-4.
- Banco de dados interno: Banco de dados SQLite embutido, utilizado para armazenamento persistente de dados.
- Mecanismo para exibição de conteúdo HTML: Para garantir um bom desempenho na exibição de páginas HTML, o Android usa uma biblioteca chamada WebKit, mesma biblioteca utilizada no browser GoogleChrome, Safari, iPhone e celulares Nokia da linha S60.

Juntamente com a camada de bibliotecas existe a camada responsável pela execução do Android, essa camada é composta por uma máquina virtual Java chamada Dalvik, desenvolvida pela Google e otimizada para dispositivos móveis. Todo o código escrito para Android é feito em Java.

Pelo fato da execução das aplicações ocorrer dentro de uma máquina virtual Java é possível acessar grande parte das bibliotecas disponíveis no Java, garantindo um grande ganho de velocidade durante o desenvolvimento.

A próxima camada, logo após a camada de execução, é a camada que possui aplicativos básicos para o funcionamento do telefone, como por exemplo:

- Gerenciador de atividades: Gerencia o ciclo de vida das aplicações, trata de eventos como os que ocorrem quando o usuário recebe uma mensagem de texto ou recebe uma ligação.
- Provedor de conteúdo: Armazena dados que devem ser compartilhados entre os aplicativos, como por exemplo os contatos do telefone do usuário.
- Gerenciador de recursos: Gerencia tudo que é distribuído com os aplicativos que não seja código, como por exemplo imagens, ícones, sons e vídeos.

- Gerenciador de localização: Responsável por capturar e disponibilizar, para as aplicações, dados sobre a localização do dispositivo.
- Gerenciador de notificações: Responsável por exibir alertas disparados pelo o sistema para o usuário, serve como central de comunicação entre o sistema e o usuário.

A última camada na arquitetura de aplicações Android é a camada dos aplicativos que rodam no Android. Essa é a camada pela qual o usuário vai interagir com o sistema, enviando eventos e requisições para os aplicativos. Alguns dos aplicativos que já vem instalados por padrão num sistema Android são:

- Aplicativo para discagem de números de telefone;
- Leitor de email;
- Gerenciador de contatos;
- Navegador de internet;

Através de um aplicativos chamado Android Market é possível instalar dezenas de outros aplicativos no Android.

2.7 METODOLOGIAS DE DESENVOLVIMENTO DE SOFTWARE

Nesta etapa serão apresentadas algumas metodologias de desenvolvimento mais conhecidas.

2.7.1 CASCATA

Foi a primeira metodologia para desenvolvimento de software amplamente utilizada pelas empresas. Por volta de 1970 a indústria de software descobriu que o processo de desenvolvimento de software precisava ter uma organização maior, com base nesta necessidade Royce desenvolveu uma metodologia linear para desenvolvimento de software (ROYCE, 1970).

Nesta metodologia o desenvolvimento segue algumas etapas previamente definidas. Segundo (PRESSMAN, 2009), as etapas existentes são:

- Engenharia do Sistema: Etapa inicial dedicada à análise de requisitos de forma ampla, ideal para definir restrições tecnológicas.
- Análise de Requisitos: Levantamento das funcionalidades necessárias para o software. Nesta fase é criado um conhecimento profundo sobre o domínio de

negócios do software. Tudo que for levantando deve ser documentado e revisto pelo cliente.

- **Projeto:** Nesta etapa são tomadas decisões de arquitetura e modelagem respeitando os requisitos levantados nas fases anteriores. O resultado é um documento que representa detalhadamente como cada parte do sistema deve ser implementada.
- **Geração de Código:** Etapa onde ocorre a codificação do que foi estabelecido na fase de projeto.
- **Testes:** Nesta etapa deve ser feita a validação de cada funcionalidade do sistema de modo a garantir que o que foi implementado está de acordo com o que foi projetado.
- **Manutenção:** Etapa destinada à correções nas funcionalidades já implementadas ou destinada à adição de novas implementações.

2.7.2 RATIONAL UNIFIED PROCESS (RUP)

Rational Unified Process (**RUP**) é a versão do Processo Unificado (**UP**), para desenvolvimento de software, criada pela Rational Corporation (JACOBSON, 1999).

Por muito tempo RUP foi largamente utilizado em grandes aplicações, assim ganhou a fama de que era um processo pesado. Com a crescente utilização de metodologias ágeis para desenvolvimento de software, a Rational Corporation tem seus esforços voltados para mostrar que o RUP é um processo adaptável e flexível (EVANS, 2003).

O processo unificado é dividido em quatro fases:

- **Concepção:** Etapa dedicada à coleta de requisitos, levantamento dos principais riscos e um levantamento de esforços.
- **Elaboração:** Os requisitos são estudados mais profundamente. A arquitetura do sistema é desenvolvida e testada com objetivo de reduzir riscos e obter uma estimativa mais precisa dos esforços. Esta etapa é subdividida em iterações que contêm: levantamento de requisitos, modelagem, programação e testes.
- **Construção:** Com base na estrutura criada na etapa de elaboração, as implementações do sistemas são iniciadas. Essas implementações são divididas em iterações que incluem codificação e testes de aceitação com os clientes.

- Transição: As implementações estão finalizadas e o sistema pode ser entregue para o cliente. Alguns ajustes no sistema são aceitáveis antes da entrega final para o cliente.

2.7.3 PROGRAMAÇÃO EXTREMA (XP)

Depois de muitos anos trabalhando com desenvolvimento de software, *Kent Beck* propôs uma metodologia para desenvolvimento de software conhecida como Programação Extrema (**XP**), também conhecida como *Extreme Programming*.

Com base em sua experiência com *SmallTalk*, em 1996 Beck publicou seu livro sobre técnicas para programação (BECK, 1996). No mesmo ano da publicação do livro, *Beck* foi convidado para liderar um projeto muito importante na *Chrysler*, era um sistema para controlar a folha de pagamento da empresa, este sistema já estava com prazos e custos estourados e ainda não possuía resultados. Com a participação de *Martin Fowler* e *Ron Jeffries*, *Beck* conseguiu alta produtividade para a equipe e eles conseguiram entregar um sistema de excelente qualidade começando do zero e gastando menos tempo do que foi gasto nas tentativas anteriores.

Neste projeto da *Chrysler*, *Beck* decidiu utilizar uma série de práticas que foram consideradas eficientes em outros projetos, este foi o início da Programação Extrema. Algumas das práticas utilizadas, que foram impactantes para o aumento da qualidade do produto, são:

- Revisão de código;
- Testes;
- Integração rápida;
- Feedback do cliente;
- Design simples;

Além de aplicar as práticas descritas acima, Beck intensificou o uso destas práticas, como por exemplo:

- Intensificou a revisão de código através da programação em pares;
- Intensificou o uso de testes com testes automatizados;
- Intensificou o *feedback* do cliente, fazendo com que o cliente ficasse mais presente durante o desenvolvimento do software.

Com base nestes princípios e práticas utilizados, em 1999, Beck publicou um livro (BECK, 1999) que ajudou a popularizar a metodologia em questão. O livro aborda os principais valores da metodologia:

- Comunicação: é fundamental para conseguir atender as necessidades do cliente, a presença do cliente aumenta a frequência dos *feedbacks* a respeito do que foi desenvolvido e favorece o desenvolvimento de um produto com maior qualidade.
- Simplicidade: deve ser sempre utilizada para evitar que o tempo da equipe seja gasto com atividades que não precisam ser complexas. Além disso a complexidade aumenta a possibilidade de introdução de erros. Quanto mais simples for o código desenvolvido mais facilmente um desenvolvedor pode compreender o que foi feito e conseguir contribuir.
- Coragem: é comum existir uma resistência em relação às mudanças, mesmo assim as mudanças são necessárias, tendo em vista que durante o desenvolvimento do sistema corre-se o risco de não se tomar as melhores decisões. Para garantir que o sistema a ser desenvolvido está tomando o rumo correto deve-se ter coragem para realizar mudanças e inovações.
- *Feedback* do cliente: deve ser constante entre os participantes de um projeto para compartilhar soluções e problemas, aumentando o nível de entendimento de todos sobre o projeto. Quanto mais cedo um problema for detectado e compartilhado mais cedo ele será resolvido e todos saberão com lidar com ele posteriormente.
- Respeito: deve existir entre os participantes de um projeto para que os outros valores possam ser eficazes. A falta de respeito pode comprometer a comunicação e o *feedback* do cliente sobre o que foi desenvolvido, reduzindo a transparência entre os membros da equipe.

2.7.4 LEAN

Na década de 40 a Toyota ainda era uma pequena empresa, que percebeu a possibilidade de crescimento se conseguisse produzir veículos baratos e com qualidade. *Taiichi Ohno* e *Shigeo Shingo* receberam o desafio para reduzir os custos na linha de produção da Toyota, para atingir o objetivo proposto, eles propuseram a eliminação de todo

desperdício presente desde a fabricação até a entrega do produto. As mudanças para redução de custo foram base para o início do pensamento *Lean*. Os princípios elementares para a cultura *Lean* utilizada pela Toyota eram (OHNO, 1988):

- *Just in Time*: Eliminou a necessidade de estocar itens para a produção, eliminado custos e problemas de armazenamento.
- *Stop the line*: uma parada na linha de produção era forçada se um defeito fosse encontrado, evitando a produção de produtos com erros. A linha de produção só retomava o funcionamento normal quando o defeito fosse eliminado.

Mais tarde esses princípios foram estendidos para outras áreas da empresa.

Os princípios *Lean* são voltados para a redução de custos e podem ser aplicados no desenvolvimento de qualquer tipo de produto. Alguns princípios podem surgir dependendo do contexto do produto a ser desenvolvido, no desenvolvimento de software pode-se destacar os seguintes princípios elementares (POPPENDIECK, 2003):

- Elimine o desperdício: de acordo com o criador do pensamento *Lean*, desperdício é tudo aquilo que não acrescenta valor ao produto na percepção do cliente. Alguns tipos de desperdícios que devem ser evitados durante o desenvolvimento de software são listados abaixo:
 - Funcionalidades incompletas, poderiam ser evitadas através de um planejamento prévio;
 - Excesso de processos, normalmente não acrescentam valor para o cliente final;
 - Antecipação de funcionalidades, pois aumenta a complexidade do sistema desnecessariamente;
 - Troca de tarefas, pois o número excessivo de trocas de contexto reduz a produtividade;
 - Esperar por informações, ter que esperar em alguma etapa do ciclo de desenvolvimento atrasa todo o ciclo;
 - Troca de pessoas entre equipes causa desperdício em razão da perda de conhecimento;
 - Defeitos causam desperdício pois o tempo gasto costuma ser muito grande quando se comparado com o tempo utilizado na prevenção;

- Amplifique o aprendizado: lições devem ser extraídas dos problemas enfrentados e disseminadas entre os membros da equipe, criando uma base de conhecimento para ajudar na evolução e amadurecimento da equipe.
- Entregue rápido: deve-se entregar pequenas partes funcionais do sistema de forma incremental para aumentar a interação com o cliente e aumentar a frequência com que o cliente dá os *feedbacks* sobre o que foi desenvolvido.
- Valorize a equipe: as pessoas não devem ser tratadas apenas como recursos, os membros da equipe devem ter seu trabalho reconhecido para conseguirem motivação e assim contribuir para o desenvolvimento do produto de forma efetiva.
- Adicione segurança: deve-se implementar soluções que deixe a equipe segura de que o produto que está sendo desenvolvido tenha qualidade. Priorize a automatização de testes ao invés da detecção e correção de defeitos.

2.7.5 TEST DRIVEN DEVELOPMENT (TDD)

Segundo (FREEMAN, 2009), o desenvolvimento dirigido por testes (**TDD**) é uma ideia simples que diz: escreva testes para o seu código antes de escrever o próprio código. Utilizar TDD muda o papel que os testes tem no desenvolvimento de software, ao invés de utilizar os testes pra evitar que os clientes tenham problemas ao utilizar o sistema, o levantamento dos testes faz com que a equipe entenda a funcionalidade antes de implementá-la, evitando que os erros cheguem aos clientes.

TDD é uma prática muito usada em abordagens ágeis para desenvolvimento de software, é umas das principais práticas da Programação Extrema.

O ciclo básico para quem usa TDD consiste em:

- Escrever um teste;
- Escrever o código para fazer o teste passar;
- Melhorar o código garantindo que os testes ainda estão passando;

Este ciclo deve ser repetido a cada mudança que for feita no sistema. A FIGURA 2.2 representa o ciclo descrito anteriormente.

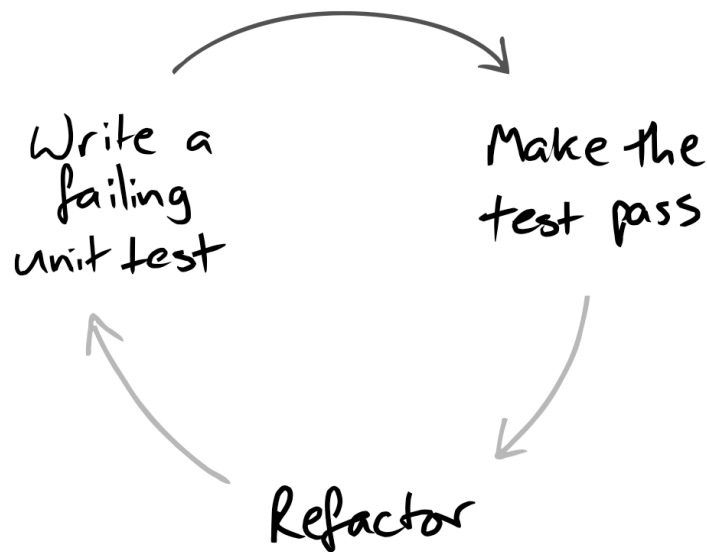


Figura 2.2 – Ciclo básico para TDD.

FONTE (FREEMAN, 2009)

A medida que o sistema vai sendo desenvolvido *feedbacks* relativos à implementação e design do sistema são recebidos.

Quando escreve-se testes tem-se os seguintes benefícios:

- Ajuda a ter uma ideia clara de qual deve ser o próximo passo para o desenvolvimento, o design tende a ser melhor;
- O desenvolvimento de componentes desacoplados é favorecido, tendo em vista que componentes desacoplados são mais fáceis de se testar;
- O código de teste deve ser como uma especificação executável do código desenvolvido;
- Ganha-se um conjunto de testes de regressão.

Quando os testes escritos anteriormente são executados obtém-se os seguintes benefícios:

- Facilidade na detecção de erros, o erro é detectado de forma instantânea, assim que o conjunto de testes é executado;
- Ajuda a saber quando a funcionalidade está terminada, evitando a criação de partes de código desnecessárias para a funcionalidade em questão.

3 METODOLOGIA

Neste capítulo, será apresentado como o sistema foi desenvolvido, as ferramentas escolhidas e mais detalhes sobre o estudo de caso.

Como estudo de caso foi desenvolvido um sistema de catálogo digital que possibilita ao usuário cadastrar seus itens e, em seguida, visualizá-los em um dispositivo móvel, celular ou tablet.

Na opinião do autor, catálogo digital é um termo muito amplo e pode ser utilizado em diversos contextos, como:

- Catálogo de Roupas;
- Catálogo de Remédios;
- Catálogo de Imóveis;
- Catálogos de Automóveis;

Pode-se ver que existem inúmeras aplicações para este tipo de sistema baseado em um catálogo digital, este trabalho aborda a situação em que o catálogo digital é utilizado por um restaurante.

O sistema pode ser dividido em dois blocos básicos, de acordo com a FIGURA 3.1:

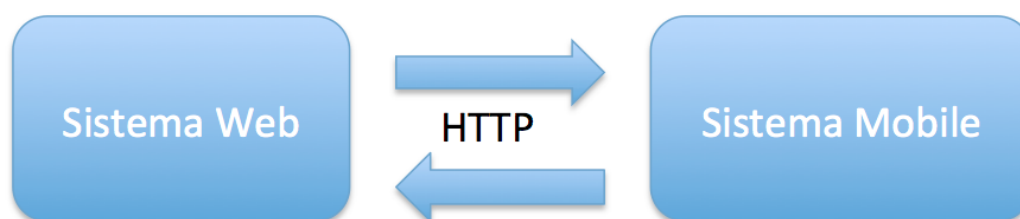


Figura 3.1 – Diagrama em blocos do sistema.

O sistema Web que está representado na FIGURA 3.1 é responsável pelos seguintes itens:

- Controle de usuários;
- Gerenciamento de categorias dos itens;
- Gerenciamento dos itens;
- Prover interface HTTP para comunicação com o sistema mobile;
- Disponibilizar o aplicativos mobile para download;

O sistema Mobile que está representado na FIGURA 3.1 é responsável pelos seguintes itens:

- Fazer a comunicação com o sistema Web para atualizar as categorias e os itens de um usuário;
- Exibir os itens separados por categoria;

O sistema mobile se comunica com o sistema Web através de requisições HTTP para sincronizar os dados cadastrados no sistema.

3.1 METODOLOGIA UTILIZADA NO DESENVOLVIMENTO

A metodologia proposta consiste em uma união dos princípios do pensamento *Lean* e práticas do *Extreme Programming*.

Alguns dos princípios do pensamento *Lean* que foram utilizados são:

- Elimine o desperdício: Este item tem como foco evitar a geração de defeitos no software, já que defeitos geram um desperdício muito grande quando são corrigidos. Neste caso deve-se favorecer o trabalho preventivo utilizando uma abordagem amparada em testes automatizados.
- Amplifique o aprendizado: Através da participação de grupos de discussão sobre as tecnologias utilizadas é possível economizar muito tempo durante a fase de aprendizado de uma tecnologia.
- Adicione Segurança: Este item está relacionado com a segurança que se deve ter ao realizar modificações durante o desenvolvimento do software, um item imprescindível para aumentar a segurança durante o desenvolvimento é a execução automatizada dos testes desenvolvidos.

Além dos princípios *Lean* algumas práticas de *Extreme Programming* também foram adotadas:

- *Test Driven Development*: Sem dúvida esta prática é a mais valiosa, pois:
 - Aumenta a frequência de *feedback* do código desenvolvido;
 - Ajuda a aumentar a coragem para mudanças, graças ao aumento de *feedback* e consequentemente com o aumento da segurança para os desenvolvedores;
 - Faz com que o código seja criado da forma mais simples possível, evitando a criação de código desnecessário;

- Refatoração: É uma prática bem interessante que ajuda a melhoria do código existente. Refatorar é o ato de mudar código existente, o ideal é executar os testes do código modificado e verificar se o que foi alterado não causou impacto e tudo continua funcionando corretamente. Para-se usar refatoração é recomendado o uso de *Test Driven Development*, garantindo assim maior segurança durante o desenvolvimento.

3.2 DESENVOLVIMENTO DO SISTEMA WEB

O sistema web pode ser dividido nos seguintes casos de uso, segundo a FIGURA 3.2:

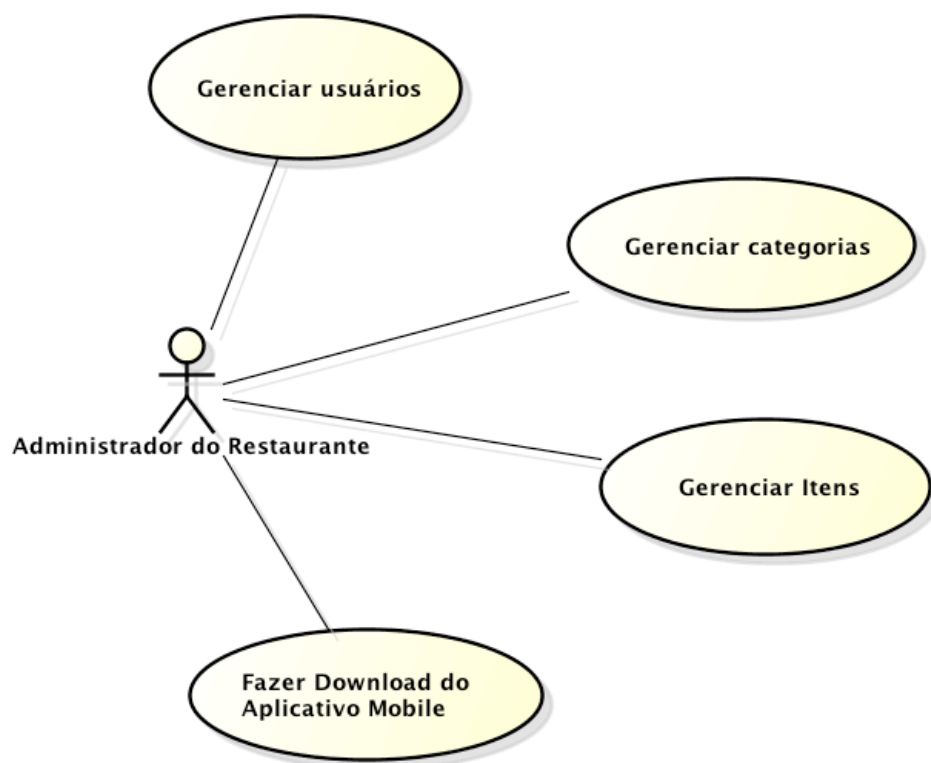


Figura 3.2 – Diagrama de Casos de Uso do Sistema Web.

O caso de uso “Gerenciar usuários” contempla todas as funcionalidades relacionadas aos usuários do sistema, listadas a seguir:

- Criação de uma nova conta de usuário;
- Alteração dos dados de um usuário;
- Cancelamento da conta de um usuário;

Assim que o Administrador do Restaurante se cadastrar para utilizar o sistema, ele poderá cadastrar as categorias em que seus itens serão divididos, por meio do caso de uso “Gerenciar Categorias”, com este caso de uso o usuário poderá:

- Cadastrar novas categorias para seus itens;
- Alterar suas categorias já cadastradas;
- Excluir as categorias;
- Visualizar todas as categorias cadastradas;

Após o cadastro das categorias o usuário poderá cadastrar seus itens que devem ser exibidos no dispositivo móvel. Através do caso de uso “Gerenciar Itens” o usuário poderá:

- Cadastrar novos itens;
- Alterar itens;
- Excluir itens;
- Visualizar os itens já cadastrados;

O último caso de uso relacionado com o sistema Web é o caso de uso “Fazer Download do Aplicativo Mobile”, que consiste basicamente em disponibilizar um executável para ser instalado no dispositivo móvel que será utilizado como catálogo.

3.2.1 TECNOLOGIAS UTILIZADAS

O Sistema Web foi desenvolvido em Ruby, utilizando Rails como framework MVC. A escolha dessa combinação foi motivada pela facilidade e velocidade que ela dá ao desenvolvimento de aplicações guiadas por testes, já que existem muitos plugins e extensões construídas em Ruby para facilitar o desenvolvimento de aplicações que usam práticas como o *Test Driven Development*.

Uma ferramenta chamada RSpec foi utilizada para contribuir com a legibilidade do código de teste. O RSpec é um plugin para Rails que provê muitos facilitadores para serem usados na criação de testes de unidade e de integração.

A FIGURA 3.3 exemplifica um trecho de um código de teste escrito usando RSpec:

```

describe "ao salvar um item" do
  def criar_item
    categoria = Factory(:categoria)
    post :create, :item => Factory.attributes_for(:item, :categoria => categoria)
  end

  it "deve carregar a tela de novo item" do
    get :new
    response.should render_template(:action => "new")
  end

  it "deve redirecionar o usuario para a tela de listagem" do
    criar_item
    response.should redirect_to(itens_url)
  end

  it "deve redirecionar para a tela de criacao quando ocorrer um erro durante a criacao" do
    post :create, :item => Item.new
    response.should render_template(:action => "new")
  end
end

```

Figura 3.3 – Trecho de um código escrito usando RSpec.

Pode-se observar na FIGURA 3.3 que os testes são organizados por funcionalidade e dentro de cada funcionalidade estão listados os casos de testes. No exemplo da FIGURA 3.3 a funcionalidade em teste é a ação de salvar um item e os casos de teste descritos são:

- Testar se a tela de criação de um novo item é carregada quando o usuário requisitar a tela;
- Testar se o usuário é redirecionado para a tela de listagem quando um item for cadastrado;
- Testar se o usuário permanece na tela de criação de item caso ocorra um erro ao tentar criar um item inválido.

Além da legibilidade e facilidade do uso do RSpec, quando ele é utilizado pode-se construir um conjunto de testes e executá-lo de maneira automatizada com o uso de uma ferramenta chamada Autotest, que é capaz de detectar mudanças no código e iniciar a execução dos testes relacionados ao código que foi modificado. Além de executar os testes relacionados ao código que foi modificado, periodicamente, o Autotest dispara a execução de todos os testes do sistema, garantindo a qualidade do código por completo, não se limitando apenas ao código que foi alterado.

A FIGURA 3.4 exemplifica uma execução do Autotest:

```
.....  
Finished in 1.75 seconds  
65 examples, 0 failures  
Coverage report generated for RSpec to /Users/jpjcjbr/dev/ruby/cardapio_na_mao/coverage
```

Figura 3.4 – Resultado da execução do Autotest

De acordo com a FIGURA 3.4 pode-se ver que cada teste executado é representado por um ponto verde no terminal. Depois que a execução de todos os testes é finalizada um resumo é apresentado ao usuário mostrando informações como:

- Tempo gasto durante a execução dos testes;
- Número de testes executados com sucesso;
- Número de falhas nos testes executados;

Combinando o RSpec com o Autotest pode-se garantir que nossos testes estão bem escritos, legíveis e que estão sendo executados continuamente.

Além dos plugins destinados para uso durante o desenvolvimento dos testes, existem plugins para facilitar o desenvolvimento de funcionalidade rotineiras, como por exemplo:

- **Devise:** Plugin criado por um brasileiro, chamado José Valim. Esse plugin é destinado ao controle do fluxo de autenticação de usuários nas aplicações. Além do controle do processo de autenticação ele também é capaz de criar várias funcionalidades de maneira simples, entre elas:
 - Enviar e-mails de confirmação durante a criação de um novo usuário no sistema;
 - Habilitar rotinas para travar o acesso de um usuário ao sistema, podendo levar em consideração o número de tentativas sem sucesso.
 - Habilitar rotinas para reenviar a senha para os usuários em caso de perda.
- **PaperClip:** Plugin criado pela empresa Thoughtbot. O PaperClip é um plugin criado para resolver o problema de upload de imagens. Além de tratar o processo de upload de imagens, ele possui algumas funcionalidades interessantes:
 - Define tamanhos de imagem que devem ser aceitos;
 - Redimensiona automaticamente as imagens, evitando a codificação de algoritmos para gerar miniaturas das imagens.
 - Pode ser plugado em serviços de armazenamento de arquivos, como o s3 AWS da Amazon.

- Will Paginate: Plugin especializado na preparação de registros para serem paginados na camada de visualização. Possui interface fluente para facilitar a manipulação dos resultados.

3.2.2 DESENVOLVIMENTO

Foram levantadas quais funcionalidades seriam desenvolvidas, depois de ter as atividades em mãos elas foram priorizadas de acordo com as dependências existentes entre elas.

As primeiras funcionalidades que foram desenvolvidas foram as relacionadas com o gerenciamento de categorias.

Antes de iniciar o desenvolvimento alguns testes de aceitação foram criados para garantir que o resultado do desenvolvimento pudesse atender os requisitos iniciais, evitando desenvolver funcionalidades desnecessárias. Exemplos:

- A tela de cadastro e edição de categoria devem conter uma caixa de texto para entrar com o nome da categoria.
- Quando uma categoria for cadastrada o usuário deve ser redirecionado para a tela de listagem de categorias.
- A partir da tela de listagem de categorias deve-se ter acesso aos links para edição e exclusão das categorias.
- O modelo de Categoria deve ser validado exigindo a presença do nome da categoria antes de qualquer inserção ou alteração.
- Não devem existir categorias repetidas para um usuário.

Depois de definidos os testes de aceitação deve-se iniciar o desenvolvimento. Como este sistema é baseado em testes, deve-se criar o teste referente a cada parte do sistema para validar gradativamente a evolução do sistema.

Neste caso foram criados os seguintes testes:

- Testes para validar se as telas estavam de acordo com o que foi especificado;
- Testes para validar se o modelo estava com as validações corretas;
- Testes para validar se o controlador estava fazendo seu papel e redirecionando as requisições para os locais corretos;

Com base no que foi mostrado anteriormente consegue-se uma cobertura de testes, abrangendo todas as camadas do sistema, garantindo que todo o fluxo está coberto por testes.

Para exemplificar o que foi desenvolvido pode-se usar como base a FIGURA 3.5, que representa a classe de teste da classe Categoria que foi desenvolvida, e a FIGURA 3.6, que representa a classe de modelo Categoria.

```
1 require 'spec_helper'
2
3 describe Categoria do
4
5   subject { Factory(:categoria) }
6
7   it { should belong_to(:user) }
8   it { should have_many(:itens) }
9   it { should validate_presence_of(:nome) }
10  it { should validate_presence_of(:user) }
11
12  it "deve possuir um nome unico no escopo de usuario" do
13    categoria1 = Factory(:categoria, :nome => 'Categoria1')
14
15    categoria_invalida = Categoria.new(:nome => 'categoria1')
16    categoria_invalida.user = categoria1.user
17
18    lambda{
19      categoria_invalida.save
20    }.should_not change(Categoria, :count)
21  end
22
23  it "pode possuir o mesmo nome para usuarios diferentes" do
24
25    categoria1 = Factory(:categoria, :nome => 'categoria1')
26
27    nova_categoria = Categoria.new(:nome => 'categoria1')
28    nova_categoria.user = Factory(:user, :email => 'jpjcjbr2@gmail.com')
29
30    lambda{
31      nova_categoria.save
32    }.should change(Categoria, :count).by 1
33  end
34 end
```

Figura 3.5 – Classe de teste da classe Categoria

De acordo com a FIGURA 3.5 pode-se ver que existem testes validando o seguintes pontos:

1. Validando se a classe Categoria pertence à classe User, na linha 7;
2. Validando se a classe Categoria possui muitos itens, na linha 8;
3. Validando a obrigatoriedade dos atributos user e nome da classe Categoria, nas linhas 9 e 10;
4. Validando se não existem categorias com nomes repetidos para o mesmo usuário, da linha 12 até a linha 21;
5. Validando a possibilidade de existirem categorias com nomes iguais para usuários diferentes, da linha 23 até a linha 33.

Com base na FIGURA 3.5 pode-se ver que os testes criados com RSpec são altamente legíveis e fáceis de entender.

Os testes foram numerados para facilitar a explicação da FIGURA 3.6.

```
1 class Categoria < ActiveRecord::Base
2   belongs_to :user
3   has_many :itens, :dependent => :destroy
4   validates :nome, :length => { :maximum => 30 }
5   validates_presence_of :nome
6
7   validates_presence_of :user
8   validates_associated :user
9
10  validates_uniqueness_of :nome, :case_sensitive => false, :scope => [:user_id]
11 end
```

Figura 3.6 – Classe Categoria

De acordo com a FIGURA 3.6 pode-se ver que o código da classe Categoria foi implementado respeitando os testes criados em sua classe de teste.

Pode-se ver que a linha 2 faz com que o teste 1 seja executado com sucesso, garantindo a associação da Categoria com o User.

Na linha 3 foi implementado o código para satisfazer o teste 2, assegurando que a classe Categoria esteja associada com muitos itens.

Da linha 4 até a linha 8 está o código referente ao teste 3, que garante que a classe Categoria possua um user e possua um nome.

E por fim, o código da linha 10 satisfaz os testes 4 e 5 de uma vez, criando um escopo para buscas que leva em consideração o atributo user_id da classe Categoria, fazendo com que não existam categorias com nomes iguais para um usuário.

De forma análoga foram desenvolvidas as funcionalidades relacionadas aos itens, usuários e download do executável para o dispositivo móvel.

3.3 DESENVOLVIMENTO DO SISTEMA MOBILE

O sistema mobile pode ser dividido nos seguintes casos de uso, de acordo com a FIGURA 3.7:

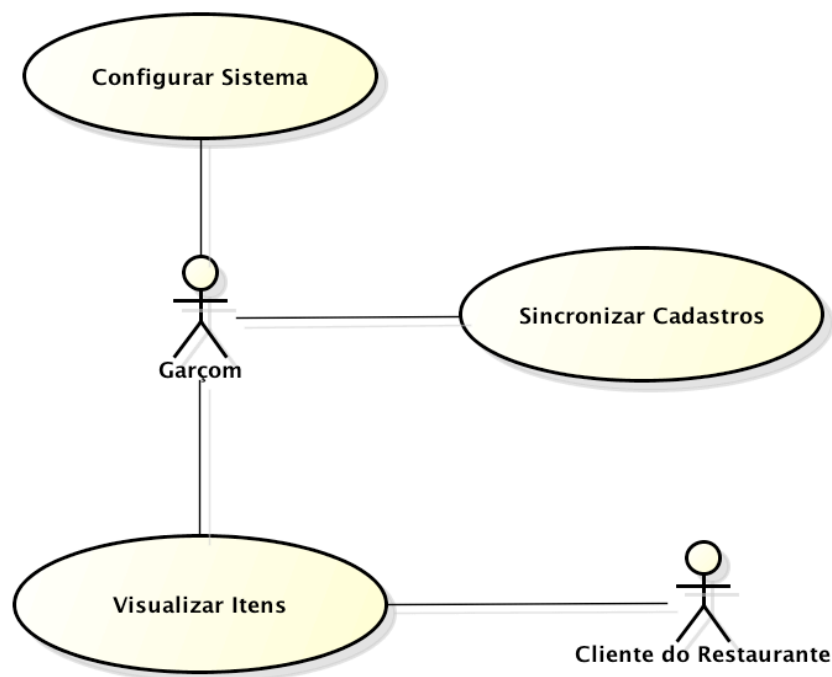


Figura 3.7 – Diagrama de Casos de Uso do Sistema Mobile

O caso de uso “Configurar Sistema” será executado apenas pelo Garçom do Restaurante, este caso de uso contempla a inserção dos dados para que o sistema mobile consiga se comunicar com o sistema web, podendo assim, buscar os dados cadastrados.

Depois de realizar a configuração básica do sistema, o usuário pode iniciar o processo de sincronização dos dados entre o dispositivo móvel e o servidor através do caso de uso “Sincronizar Cadastros”.

O último caso de uso, que possui o nome “Visualizar Itens”, possibilita que o usuário visualize os itens que foram, previamente, cadastrados no sistema web e sincronizados com o sistema mobile.

3.3.1 TECNOLOGIAS UTILIZADAS

O sistema mobile foi desenvolvido utilizando a plataforma Android como base. A escolha da plataforma Android para o desenvolvimento do sistema mobile se deu pelos seguintes motivos:

- Plataforma aberta para desenvolvimento;
- Sem a necessidade de comprar um computador específico para desenvolver, ao contrário do iPhone;
- Sem a necessidade comprar uma licença de desenvolvedor para testar o aplicativo no dispositivo móvel;
- Facilidade para realizar testes no sistema desenvolvido.

Existem frameworks para facilitar o desenvolvimento com Android, mas o uso desses frameworks foi eliminado em razão do acoplamento resultante entre o sistema que foi desenvolvido e os frameworks, dificultando a prática do TDD.

O banco de dados SQLite foi utilizado para armazenar os dados do sistema, a sua escolha se deu pelo fato de ser um banco de dados nativo da plataforma Android.

Para fazer a comunicação com o sistema web foi desenvolvido um cliente HTTP com base na API *java.net*.

3.3.2 DESENVOLVIMENTO

Durante todo o desenvolvimento do sistema móvel foram desenvolvidos testes de integração para assegurar o funcionamento do mesmo.

Por ser uma tecnologia relativamente nova, a plataforma Android ainda não possui muitas ferramentas maduras para o ciclo de desenvolvimento completo com TDD.

Neste caso não foram criados testes de unidade, apenas testes de integração, essa decisão tem como principal motivo a comparação do nível de cobertura de testes obtida quando se desenvolve com e sem testes de unidade.

Outro motivo para evitar o uso dos testes de unidade é que no sistema mobile existe uma dificuldade em separar os componentes que são responsáveis por controlar as regras da aplicação dos componentes que são responsáveis por controlar as interações com as interfaces de usuário, dificultando a criação dos testes unitários do sistema.

Para contornar esta dificuldade o sistema foi desenvolvido sempre pensando na separação entre os componentes de visualização e os componentes que fazem parte do domínio da aplicação.

Alguns testes que foram criados estão descritos abaixo:

- Verificar se as configurações do sistemas são atualizadas quando o botão de confirmação for acionado;
- Verificar se uma mensagem de sucesso é exibida quando os cadastros são sincronizados com sucesso;
- Verificar se o detalhe de um item é exibido ao usuário.
- Verificar se os menus foram criados corretamente.

A FIGURA 3.8 exemplifica o código de teste gerado para a tela de configuração do sistema móvel:

```
29 public void testDeveExibirAUrlDoServidorConfigurada()
30 {
31     assertTrue(solo.searchEditText("http://www.cardapionamao2.com.br"));
32 }
33
34 public void testDeveConterUmLabelParaUrlDoServidor()
35 {
36     assertTrue(solo.searchText("Endereço Servidor"));
37 }
38
39 public void testDeveExibirOEmailDoUsuarioConfigurado()
40 {
41     assertTrue(solo.searchEditText("jpjcjbr@gmail2.com"));
42 }
43
44 public void testDeveConterUmLabelParaOEmailDoUsuario()
45 {
46     assertTrue(solo.searchText("E-mail Usuário"));
47 }
48
49 public void testDeveConterUmBotaoParaSalvar()
50 {
51     assertTrue(solo.searchButton("Salvar"));
52 }
```

Figura 3.8 – Trecho de código de teste

De acordo com a FIGURA 3.8 pode-se ver que os seguintes testes foram criados para validar os itens presentes na tela de configuração o sistema móvel:

- Deve exibir uma caixa de entrada com a url configurada para buscar os dados cadastrados;
- Deve exibir um label para identificar a caixa de texto referente a url do servidor;
- Deve exibir uma caixa de entrada com o email do usuário configurado para buscar os dados;
- Deve exibir um label para identificar a caixa de texto referente ao email do usuário;
- Deve exibir um botão para salvar os dados do formulário.

4 RESULTADOS

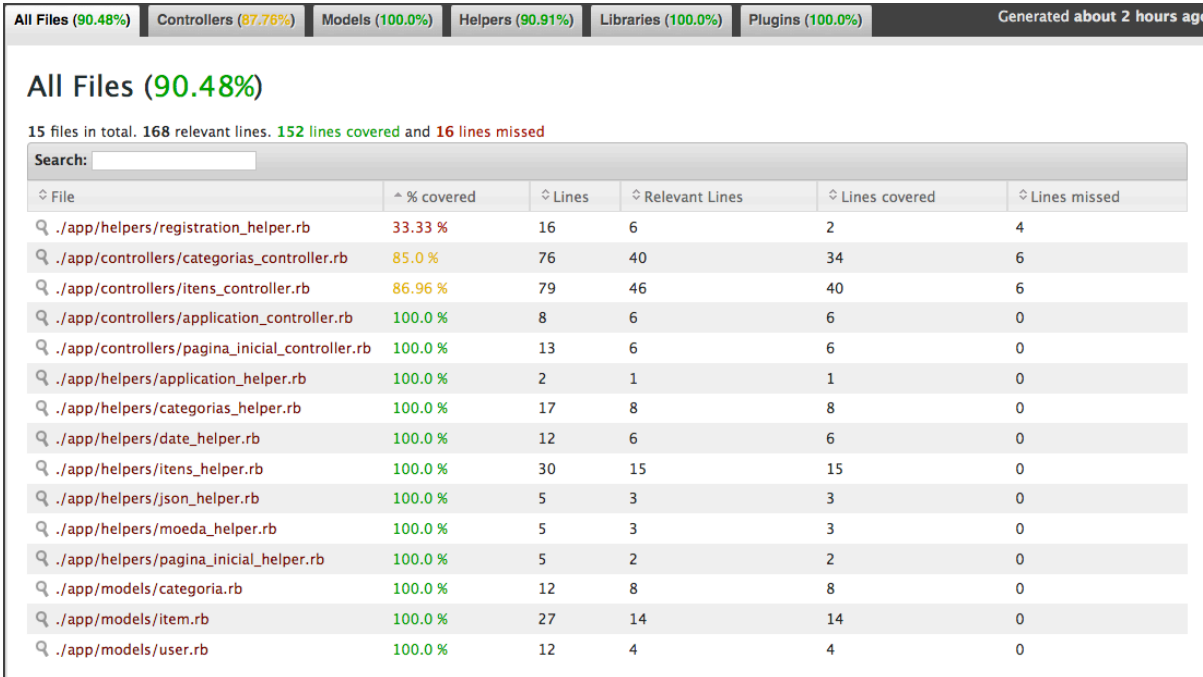
Para poder mostrar os resultados obtidos durante o desenvolvimento deste trabalho utilizou-se ferramentas para medir a cobertura de testes do código produzido.

Essas ferramentas que medem a cobertura dos testes verificam as linhas de código que foram percorridas durante a execução dos testes e fazem o cálculo percentual da quantidade de linhas de código cobertas por testes.

Foram utilizadas duas ferramentas para avaliar a cobertura de testes dos sistemas, uma para o sistema web e outra para o sistema móvel.

4.1 SISTEMA WEB

Para o sistema Web foi utilizado um plugin de Ruby chamado de SimpleCov, com o uso deste plugin é possível gerar o relatório de cobertura a cada execução da bateria de testes automaticamente. Um exemplo do relatório de cobertura, obtido através do plugin SimpleCov, é mostrado na FIGURA 4.1:



All Files (90.48%)	Controllers (87.76%)	Models (100.0%)	Helpers (90.91%)	Libraries (100.0%)	Plugins (100.0%)	Generated about 2 hours ago
All Files (90.48%)						
15 files in total. 168 relevant lines. 152 lines covered and 16 lines missed						
Search: <input type="text"/>						
File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	
./app/helpers/registration_helper.rb	33.33 %	16	6	2	4	
./app/controllers/categorias_controller.rb	85.0 %	76	40	34	6	
./app/controllers/itens_controller.rb	86.96 %	79	46	40	6	
./app/controllers/application_controller.rb	100.0 %	8	6	6	0	
./app/controllers/pagina_inicial_controller.rb	100.0 %	13	6	6	0	
./app/helpers/application_helper.rb	100.0 %	2	1	1	0	
./app/helpers/categorias_helper.rb	100.0 %	17	8	8	0	
./app/helpers/date_helper.rb	100.0 %	12	6	6	0	
./app/helpers/itens_helper.rb	100.0 %	30	15	15	0	
./app/helpers/json_helper.rb	100.0 %	5	3	3	0	
./app/helpers/moeda_helper.rb	100.0 %	5	3	3	0	
./app/helpers/pagina_inicial_helper.rb	100.0 %	5	2	2	0	
./app/models/categoria.rb	100.0 %	12	8	8	0	
./app/models/item.rb	100.0 %	27	14	14	0	
./app/models/user.rb	100.0 %	12	4	4	0	

Figura 4.1 – Relatório de cobertura do sistema Web

Com base na FIGURA 4.1 pode-se ver que 90.48% das linhas de código estão cobertas por testes. Além disso pode-se ver detalhadamente a taxa de cobertura separada por arquivo e por categoria de arquivo.

Tal taxa de cobertura nos dá uma boa garantia de que o sistema se comporta como o esperado.

4.2 SISTEMA MÓVEL

No sistema móvel foi utilizado um plugin chamado EMMA Coverage para gerar os relatórios de cobertura.

Pode-se ver um exemplo na FIGURA 4.2 de um relatório gerado com uso do plugin EMMA:

EMMA Coverage Report (generated Thu Oct 27 23:28:42 BRST 2011)				
[all classes]				
OVERALL COVERAGE SUMMARY				
name	class, %	method, %	block, %	line, %
all classes	62% (21/34)	70% (97/139)	72% (1278/1783)	73% (341.2/468)
OVERALL STATS SUMMARY				
total packages:	10			
total executable files:	20			
total classes:	34			
total methods:	139			
total executable lines:	468			
COVERAGE BREAKDOWN BY PACKAGE				
name	class, %	method, %	block, %	line, %
br.com.cardapionamao.view	43% (3/7)	32% (6/19)	40% (142/355)	38% (33/86)
br.com.cardapionamao	18% (2/11)	44% (11/25)	55% (168/308)	54% (40.5/75)
br.com.cardapionamao.model.json	100% (2/2)	67% (4/6)	60% (12/20)	50% (4/8)
br.com.cardapionamao.dao.helper	100% (2/2)	100% (10/10)	72% (82/114)	86% (37.8/44)
br.com.cardapionamao.model	100% (3/3)	80% (28/35)	82% (127/154)	82% (46.7/57)
br.com.cardapionamao.utils.json	100% (1/1)	67% (2/3)	86% (19/22)	83% (5/6)
br.com.cardapionamao.utils.http	100% (1/1)	75% (3/4)	87% (68/78)	76% (16/21)
br.com.cardapionamao.dao	100% (4/4)	89% (24/27)	89% (431/484)	94% (115.2/123)
br.com.cardapionamao.application	100% (1/1)	100% (4/4)	92% (119/129)	88% (23/26)
br.com.cardapionamao.adapter	100% (2/2)	83% (5/6)	92% (110/119)	91% (20/22)

Figura 4.2 – Relatório de cobertura do sistema móvel

Com base na FIGURA 4.2 pode-se ver que a cobertura obtida foi de 73%, inferior em relação ao sistema web, isso se dá porque apenas testes de integração foram criados para o sistema móvel e as ferramentas de testes dificultam o desenvolvimento seguindo o TDD.

Com base na FIGURA 4.1 e na FIGURA 4.2 tem-se como resultado do uso de TDD durante o desenvolvimento um aumento na confiabilidade do sistema desenvolvido.

5 CONCLUSÃO

De acordo com os resultados, a confiança em relação ao sistema aumenta com a aplicação dos princípios Lean e práticas ágeis.

Essa confiabilidade é obtida através da segurança e do *feedback* que o desenvolvimento orientado a testes proporciona.

Além disso, a cobertura de testes obtida pelo código gerado é muito boa.

Segundo (TASSEY, 2002), uma boa cobertura de testes nos garante uma redução significativa no número de defeitos. Assim pode-se concluir que quando *Test Driven Development* é usado tem-se como resultado um sistema com um número reduzido de defeitos.

Pode-se fazer uma comparação entre as taxas de cobertura obtidas pelo sistema mobile, que foi desenvolvido com testes de integração, e pelo sistema web, que foi desenvolvido com testes de unidade e testes de integração:

A taxa de cobertura obtida com testes de unidade é bem superior em comparação à taxa obtida apenas com testes de integração, além disso, a rastreabilidade dos erros nos testes de unidade é bem maior, já que é possível apontar com maior precisão a parte do código que deu origem ao defeito.

Outro ponto importante é que a integração entre sistemas web e sistemas móveis é perfeitamente possível e pode ser feita de forma fácil com o uso de novas tecnologias que estão surgindo.

REFERÊNCIAS

- WEBBER, J. **REST in Practice: Hypermedia and Systems Architecture**. [S.l.]: O'Reilly Media, 2010.
- BURNETTE, E. **Hello, Android: Introducing Google's Mobile Development Platform**. [S.l.]: Pragmatic Bookshelf, 2010.
- BECK, K. **Smalltalk Best Practice Patterns**. [S.l.]: Prentice Hall, 1996.
- BECK, K. **Extreme Programming Explained: Embrace Change**. [S.l.]: Addison-Wesley Professional, 1999.
- EVANS, G. **Agile RUP for non-object-oriented projects**. [S.l.]: [s.n.], 2003.
- DUSTIN, E. **Effective Software Testing: 50 Specific Ways to Improve Your Testing**. [S.l.]: Addison Wesley, 2002.
- FERNANDEZ, O. **The Rails 3 Way**. Boston: Addison-Wesley Professional, 2010.
- FLING, B. **Mobile Design and Development**. Sebastopol: O'Reilly Media, 2009.
- FREEMAN, S. **Growing Object-Oriented Software, Guided by Tests**. [S.l.]: Addison-Wesley Professional, 2009.
- GUERRA, E. M. **Um estudo sobre refatoração de código de teste**. São José dos Campos: [s.n.], 2005.
- HUNT, A. **Pragmatic Unit Testing in Java with JUnit**. [S.l.]: The Pragmatic Programmers, 2003.
- JACOBSON, I. **The Unified Software Development Process**. [S.l.]: Addison-Wesley Professional, 1999.
- NEUBURG, M. **Programming iOS 4**. [S.l.]: O'Reilly Media, Inc., 2011.
- MYERS, G. J. **Testing, The Art of Software**. 2ª Edição. ed. Hoboken: John Wiley & Sons, 2004.
- MATSUMOTO, Y. **Ruby in a Nutshell**. Sebastopol: O'Reilly, 2001.
- MCGREGOR, J. D. **A practical guide to testing object-oriented software**. Upper Saddle River: Addison Wesley, 2001.
- OHNO, T. **Toyota Production System: Beyond Large-Scale Production**. [S.l.]: Productivity Press, 1988.
- PILONE, D. **Head First iPhone Development**. [S.l.]: O'Reilly Media, 2010.

POPPENDIECK, M. **Lean Software Development: An Agile Toolkit**. [S.l.]: Addison-Wesley Professional, 2003.

PRESSMAN, R. S. **Software Engineering: A Practitioner's Approach**. [S.l.]: [s.n.], 2009.

SUCESU-RS. [sucsu.org. sucesu.org](http://www.rs.sucesu.org.br/noticias/teste_soft), 2011. Disponível em: <http://www.rs.sucesu.org.br/noticias/teste_soft>. Acesso em: 8 set. 2011.

SIERRA, K. **SCJP Sun Certified Programmer for Java 6**. [S.l.]: McGraw-Hill Osborne Media, 2008.

RUBY, S.; THOMAS, D.; HANSSON, D. H. **Agile Web Development with Rails**. Releigh: The Pragmatic Programmers LLC, 2010.

RISCHPATER, R. **Beginning Java ME Platform**. [S.l.]: [s.n.], 2008.

ROYCE, W. W. **Managing the Development of large Software Systems**. [S.l.]: [s.n.], 1970.

TASSEY, G. **The Economic Impacts of Inadequate Infrastructure for Software Testing**. National Institute of Standards and Technology. [S.l.]. 2002.