|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sistema** | | PortoprintWeb | | | |
| **Controle de Versão** | | | | | |
| **Versão** | **Controle** | | **Data** | **Razões para alteração** | **Responsável** |
| 01 | 00 | | 21/06/2012 | Criação | Pedro Oliveira |
|  |  | |  |  |  |
|  |  | |  |  |  |

PortoprintWeb – Diretrizes para desenvolvimento

Conteúdo

[1 Objetivo 3](#_Toc328079709)

[2 Introdução 3](#_Toc328079710)

[2.1 Vocabularios utilizados nos testes de software 3](#_Toc328079711)

[2.1.1 Unit Test (Teste de Unidade). 4](#_Toc328079712)

[2.1.2 Functional Test (Teste Funcional). 4](#_Toc328079713)

[2.1.3 Component Test (Teste de Componente). 4](#_Toc328079714)

[2.1.4 Regression Suite (Suite de Regressão). 4](#_Toc328079715)

[2.1.5 Dummy Object e Dummy Values 4](#_Toc328079716)

[2.1.6 Fake Object 4](#_Toc328079717)

[2.1.7 Stub Object 4](#_Toc328079718)

[2.1.8 Mock Object 4](#_Toc328079719)

[2.2 JUnit Framework 4](#_Toc328079720)

[2.3 Testes Unitários e Injeção de Dependências. 5](#_Toc328079721)

[2.4 As três principais Leis do TDD. 5](#_Toc328079722)

[2.5 Acrônimo F.I.R.S.T 5](#_Toc328079723)

[2.6 Spring Test Framework 6](#_Toc328079724)

[2.7 Mockito Framework 6](#_Toc328079725)

[2.8 Alguns Benefícios do TDD 6](#_Toc328079726)

[2.8.1 Redução do tempo entre encontrar e resolver um problema: 6](#_Toc328079727)

[2.8.2 Auxílio no design do software: 6](#_Toc328079728)

[2.8.3 Auxílio na análise do problema: 6](#_Toc328079729)

[2.8.4 Desenvolvimento colaborativo: 6](#_Toc328079730)

[3 Visão Geral Estrutura de Testes para o PortoPrintWeb 7](#_Toc328079731)

[3.1 Padrão para a criação da estrutura de qualquer teste. 7](#_Toc328079732)

[3.2 Como criar e estruturar seus testes (Guia Rápido). 8](#_Toc328079733)

[3.2.1 Explicando de forma rápida a linguagem fluente do Mockito. 16](#_Toc328079734)

[3.2.2 Criando um teste funcional para testar a camada de persistencia do PPW. 17](#_Toc328079735)

[3.2.3 Documentações de Referências. 18](#_Toc328079736)

[4 Considerações Finais 18](#_Toc328079737)

# Objetivo

Fornecer visão geral da arquitetura e metodos de testes no PortoPrintWeb.

# Introdução

Desenvolvimento guiado por testes (Test Driven Development) é uma prática popularizada pela metodologia Extremme Programing (XP) e difundida no mercado desde os anos 80. Hoje em dia é uma prática em evolução adotada pelas principais empresas que possuem equipes de desenvolvimento de software e que precisam de um processo de entrega de alta qualidade.

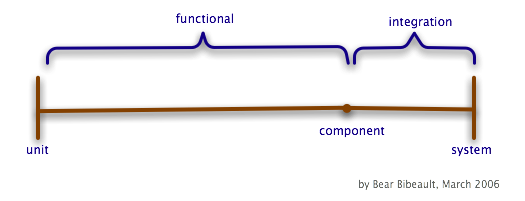
O desenvolvimento guiado por testes tem como premissa criar uma suite de testes automatizada para cada elemento de software contido no sistema. Através dessa suíte cada elemento pode ser testado separadamente reduzindo assim o contexto dos problemas que poderiam ser gerados por um bug desse elemento.

Nessa prática nenhum elemento de software é criado sem antes ter sido criado o teste, essa ideia de testar isoladamente esses elementos surgiu após a análise dos próprios desenvolvedores do tempo gasto entre escrever um componente de software e debugar o mesmo a procura de falhas que teriam ocorrido após a integração desse componente ao sistema.

Para se ter uma suíte de testes automatizada é nescessário um Framerwork de Testes Unitários.

## Vocabularios utilizados nos testes de software

Para uma melhor compreensão de todo o resto dessa documentação, segue alguns termos utilizados pelo mercado, que são muitas vezes confundidos:



### Unit Test (Teste de Unidade).

Uma unidade, é a menor quantidade de código testável, muitas vezes um único método ou função. Testes de unidade nunca devem utilizar nenhum tipo de infra-estrutura externa ao framework de teste, sendo assim milhares de testes de unidade podem ser executados em questões de segundos.

### Functional Test (Teste Funcional).

É maior que um teste de unidade, e menor do que um teste de um componente completo, normalmente testa algumas unidades trabalhando juntas afim de testar uma funcionalidade do componente por completo. Um teste funcional normalmente demora mais para rodar que o teste de unidade e pode ser que precise de algum tipo de infra-estrutura.

### Component Test (Teste de Componente).

Esse é o teste do código de produção do componente, difícilmente é integrado a suíte de testes de unidade por ser um teste mais demorado e que requer infra-estrutura.

### Regression Suite (Suite de Regressão).

A Suite de regressão é o conjunto de testes que podem ser rodados em conjunto. Uma suíte de regressão pode ser tanto de testes unitários quanto de testes funcionais.

### Dummy Object e Dummy Values

São objetos ou valores que são passados apenas para suprir os contratos a serem testados mas que acabam não sendo utilizados.

### Fake Object

São objetos que possuem códigos e comportamentos semelhantes ao de produção, mas que não podem ser utilizados em produção.

### Stub Object

São objetos que fornecem respostas pré-programadas para as chamadas dos testes, ocasionalmente podem conter código que registrem essas chamadas.

### Mock Object

São objetos pré-programados que mimetizam um comportamento esperado pelo teste. Normalmente possuem expectativas e assertações a respeito de como uma certa funcionalidade deve se comportar.

## JUnit Framework

O JUnit é um framework de teste open source criado para a plataforma java, o JUnit pode ser utilizado tanto para a criação de testes unitários quanto para rodar/simular uma infra-estrutura para testes funcionais.

## Testes Unitários e Injeção de Dependências.

Hoje em dia a maioria das aplicações utilizam um container de injeção de dependências onde as dependências das classes são injetadas automáticamente pelo container. Essas dependências são injetadas através dos métodos set ou via construtor da classe.

Da mesma forma que o container realiza essa injeção nossos testes também precisarão fazer quando nescessário. Para isso nenhuma classe concreta deve depender de classes concretas, todas as implementações precisam depender estritamente de interfaces, assim no lugar dessas interfaces podemos injetar Fakes, Stubs e Mock Objects para realizar os testes das implementações.

## As três(Cinco?) principais Leis do TDD.

Essas são leis/premissas básicas do TDD que podem ser vistas em diversas literaturas a respeito do assunto.

**Primeira Lei:** Você não deve escrever nenhum código de produção sem que antes se tenha um teste falhando.

**Segunda Lei:** Você não deve escrever no teste de unidade nada a mais que o suficiente para que o teste compile, rode e falhe.

**Terceira Lei:** Você não deve escrever no código de produção nada a mais que o suficiente para que o teste falhando, passe.

**Quarta Lei:** Assim que você tenha um teste passando, realize o refactoring do código de produção.

**Quinta Lei:** Assim que o código de produção estiver acitável, realize o refactoring do teste unitário.

## Acrônimo F.I.R.S.T

Esse é uma acrônimo interessante de como os testes unitários devem ser desenvolvidos:

**FAST:** Os testes devem rodar rapidamente, testes lentos estimulam o desenvolvedor a não roda-los continuamente e consequentemente caem em desuso.

**INDEPENDENTE:** Os testes de unidade não devem depender de infra-estutura nenhuma ou mesmo de outros testes, uma porção de código de produção precisa ser auto-testável.

**REPETÍVEL:** Os teste de unidade devem ser capazes de serem repetidos em qualquer ambiente, seja na máquina do desenvolvedor, seja na máquina de homologação com o código de homologação, seja no laptop do cliente. A única coisa que um teste deve depender é do próprio Framework de teste.

**SELF-VALIDATING:** Um teste de unidade deve se auto validar, todo teste de unidade deve possuir assertivas afim de validar a implementação do código de produção. Testes de unidade que não possuem assertivas não servem para nada.

**TIMELY:** Os testes de unidades devem ser escritos no momento correto, antes do código de produção. É completamente errada a ideia de se poder testar um sistema nos últimos ciclos do desenvolvimento.

O problema comumente gerado do desrespeito a essa regra é quando o código produtivo se torna tão complexo ou grande que simplismente fica impossível de ser testado.

## Spring Test Framework

O Spring test framework é uma extensão do framework JUnit que permite a injeção de dependências no contexto dos testes unitários, essa é umaa forma de suprir as dependências do código a ser testado. Nesse caso poderiam ser injetados na classe a ser testada: dummies, stubs, fake ou objetos concretos de produção.

## Mockito Framework

Também é um framework de extensão do JUnit que facilita a criação de Mock Objects afim de suprir as depêndencias das classes a serem testadas. O Mockito ajuda a reduzir a complexidade da escrita dos testes unitários.

## Alguns Benefícios do TDD

### Redução do tempo entre encontrar e resolver um problema:

Como um teste unitário demora segundos, podemos ter um feedback instantâneo a respeito de qualquer funcionalidade adicionada ao sistema, sem ter que testar/debugar todo sistema integrado, reduzindo assim o contexto e a complexidade dos objetos envolvidos no teste.

### Auxílio no design do software:

Desenvolver guiado a testes força o desenvolvedor a utilizar boas práticas de desenvolvimento de software tais como: Design por contrato, Princípio da responsabilidade única, Principio da segregação de interfaces.

### Auxílio na análise do problema:

TDD auxília no entendimento dos problemas do negócio, forçando o desenvolvedor a pensar nas consequências da sua implementação antes mesmo de sua concepção, solidificando o conhecimento do desenvolvedor a respeito da funcionalidade que será adicionada ao sistema.

### Desenvolvimento colaborativo:

Assim como no desenvolvimento guiado por contratos, o testes unitários facilitam no desenvolvimento colaborativo e na passagem de conhecimento entre os desenvolvedores, enquanto um desenvolvedor cria o código de produção, um outro desenvolvedor pode estar criando o teste.

# Visão Geral Estrutura de Testes para o PortoPrintWeb

Para a criação da suite de teste do PortoPrintWeb utilizamos os frameworks: JUnit, Mockit e Spring, afim de se ter uma estrutura capaz de manter teste unitários de funcionais automatizados para a nova arquitetura do sistema.

Foram criadas classes de suporte para os testes do sistema, sendo assim qualquer teste no sistema deve extender uma das classes abaixo:

**AbstractUnitTestSupport:** Essa é a classe abstrata que qualquer classe de teste unitário que só irá depender de mocks precisa extender. (Grande maioria dos testes).

**AbstractFunctionalTestSupport:** Essa é a classe que todos os testes que não utilizam mocks e que irão utilizar os recursos de injeção de depêndencias do Spring precisam extender. O teste que extender essa classe terá que implementar o método, prepareDataBase() que é reponsável por preparar a base de dados para o teste.

**AbstractComponentTestSupport:** Essa é a classe que os testes que possuem mocks e recursos do Spring precisam extender.

Além das classes acima existem outras classes de Factory e Builders que foram criadas afim de facilitar a criação dos dados para os testes, essas classes foram colocadas dentro da estrutura de teste do projeto no pacote: com.porto.portoprint.automovel.test.factory

## Padrão para a criação da estrutura de qualquer teste.

Todos os testes da aplicação devem estar estruturados da seguinte forma:

**Padrão de nome da classe de teste: ("NomeClasse" + "Test" )**

Exemplo:

classe: MinhaClass

classe de teste: MinhaClassTest

**Padrão de nome do método a ser testado: ( @Test public void "test" + "NomeMetodo" + "Cenário")**

Exemplo:

método: public Retorno meuMetodo();

método de teste:

@Test

public void testMeuMetodoDadoCertoCenario();

**Imports nescessários:**

import java.util.\*;

import org.junit.After;

import org.junit.Before;

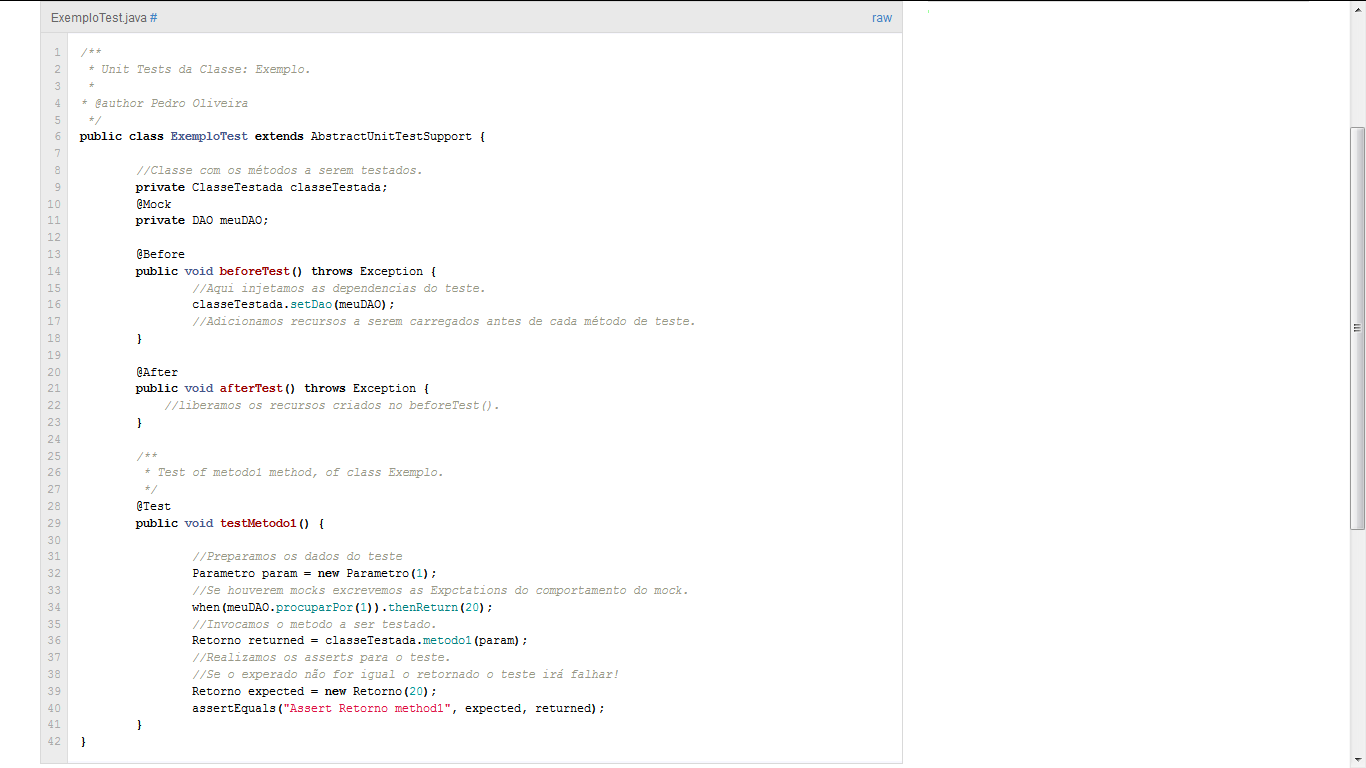
import org.junit.Test;

import static org.mockito.Mockito.\*; //Se o teste precisar de Mock.

import static org.junit.Assert.\*;

import org.mockito.Mock;

**Exemplo de uma classe de teste:**

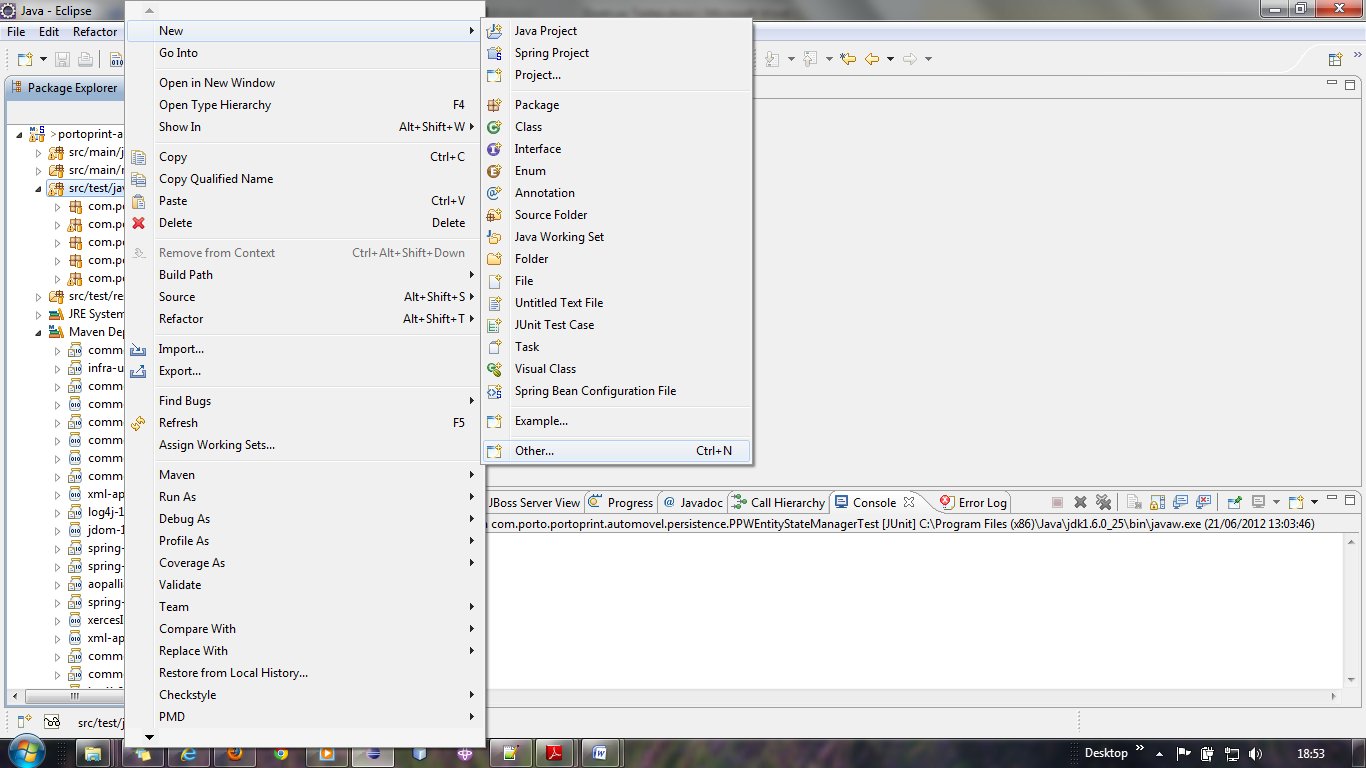


## Como criar e estruturar seus testes (Guia Rápido).

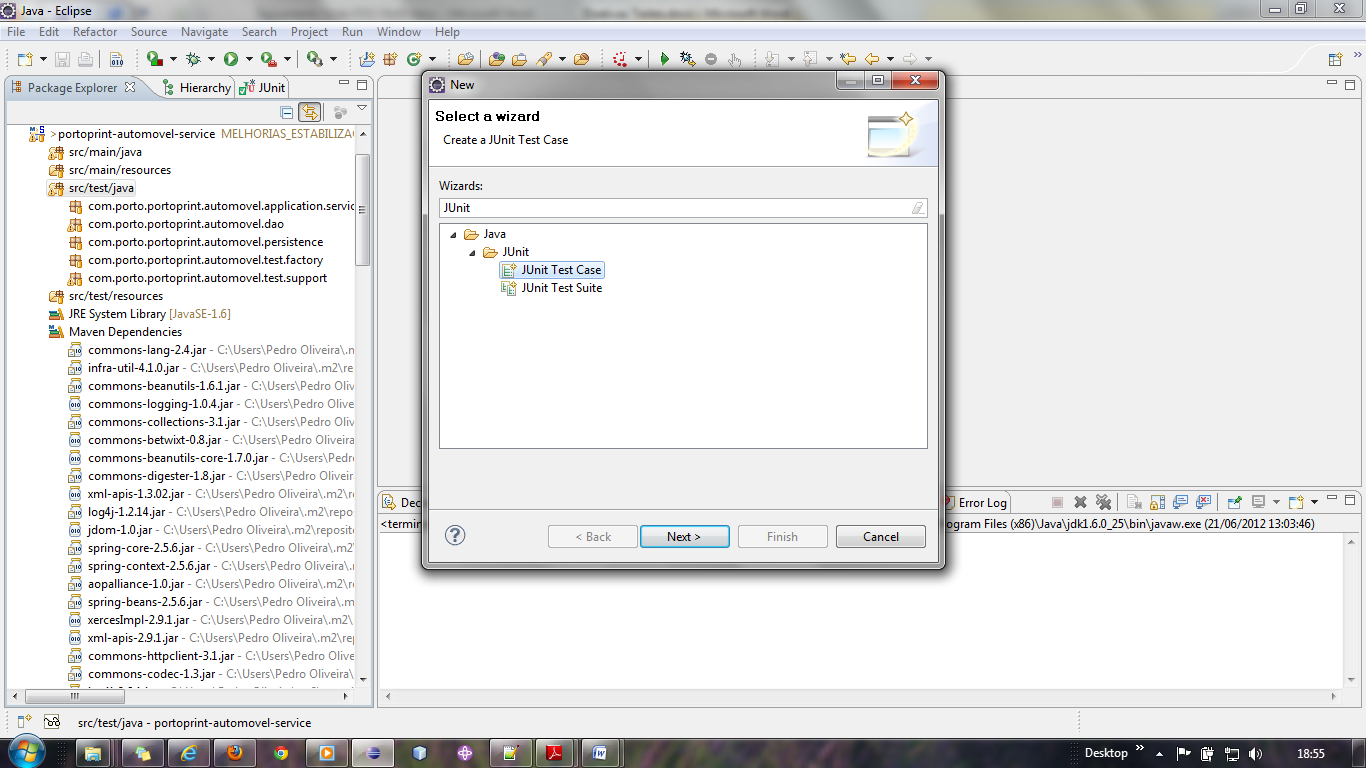
Esse é um guia rápido de como criar o seu teste unitário utilizando o Eclipse.

A estrutura geral do teste pode também ser copiada dos outros testes existentes no sistema.

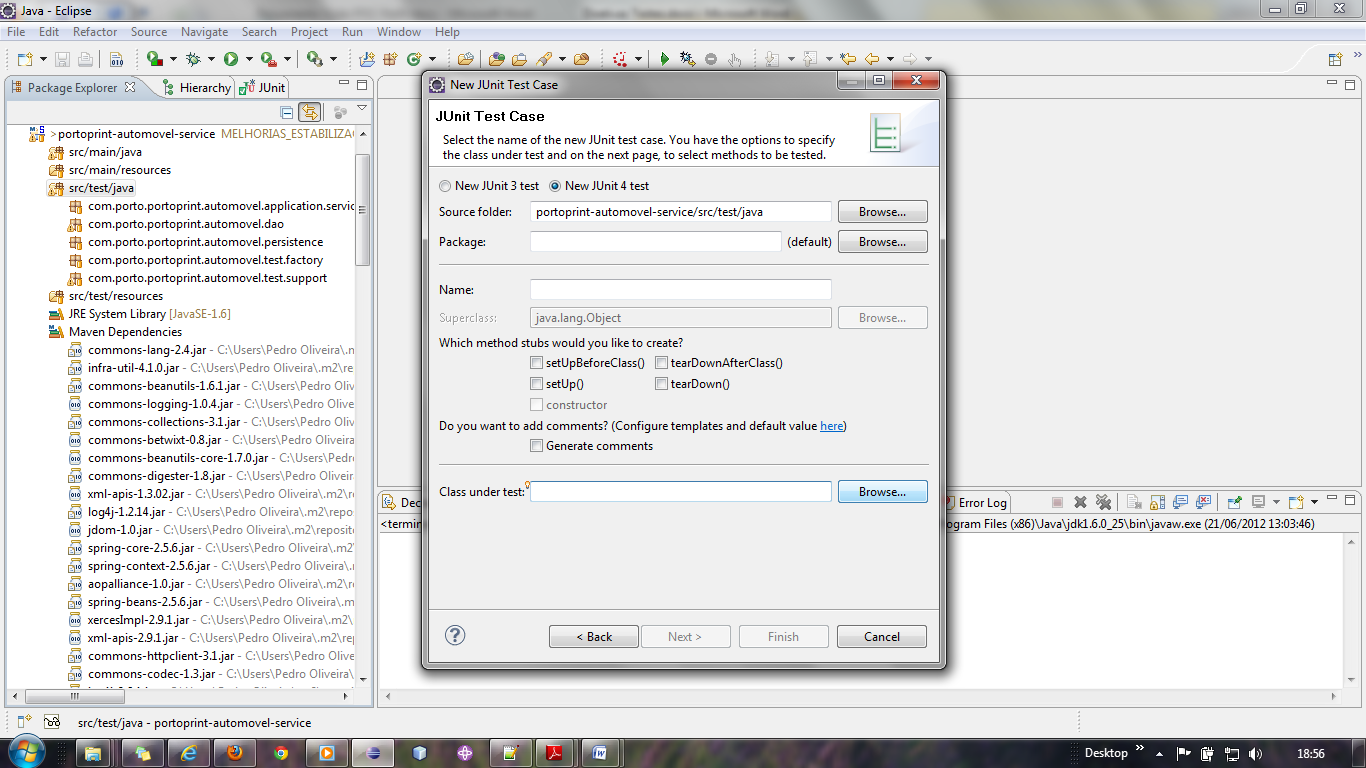
1. **Caminho:** Selecione o pacote de teste >> new >> Other

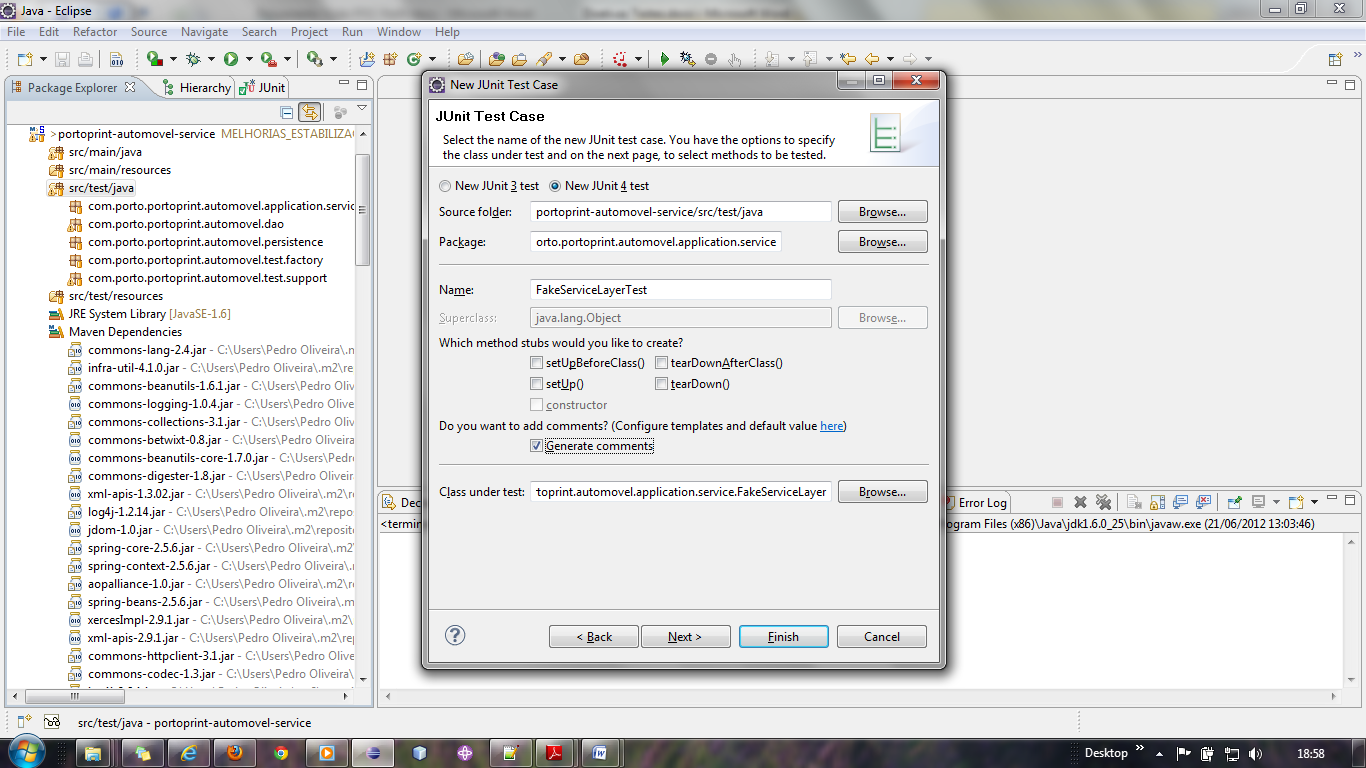


1. No wizard filtre por JUnit, conforme a figura:



1. Na opção: **classe under test**, seleciona a quesse que deverá ser testada.





1. Implemente seu teste:

Seu teste unitário deve conter a referência para a classe testada (sua futura implementação).

Seu teste também deve conter as referências para as interfaces que são dependências da sua classe testada.

No exemplo são os atributos:

private FakeServiceLayer serviceLayer (Classe Testada)

EntityStateManager entityStateManager (Interface Dependencia)

A sua classe de teste irá chamar os métodos da sua futura implementação passando os parâmetros esperados.

Nesse momento você deve pensar:

* Quais parâmetros que seu método a ser testado deve receber;
* Como ele deve processar os dados recebidos;
* Como ele deve responder.

Dentro desse contexto para validar o teste você pode verificar:

* Se os parâmetros passados ao seu método da implementação estão corretos;

Exemplo: meu método testado recebeu como parâmetro a String "xyz" ?

* Se algum método da dependência da sua classe foi invocado;

Exemplo: o método markToInsert() da classe entityStateManager foi chamado?

* Se o parâmetro da dependência foi passado corretamente;

Exemplo: Após aplicar as regras de negócio da minha implementação o parâmetro chegou na chamada do método da dependencia de forma correta?

* Validar o estado retornado pelo seu método;

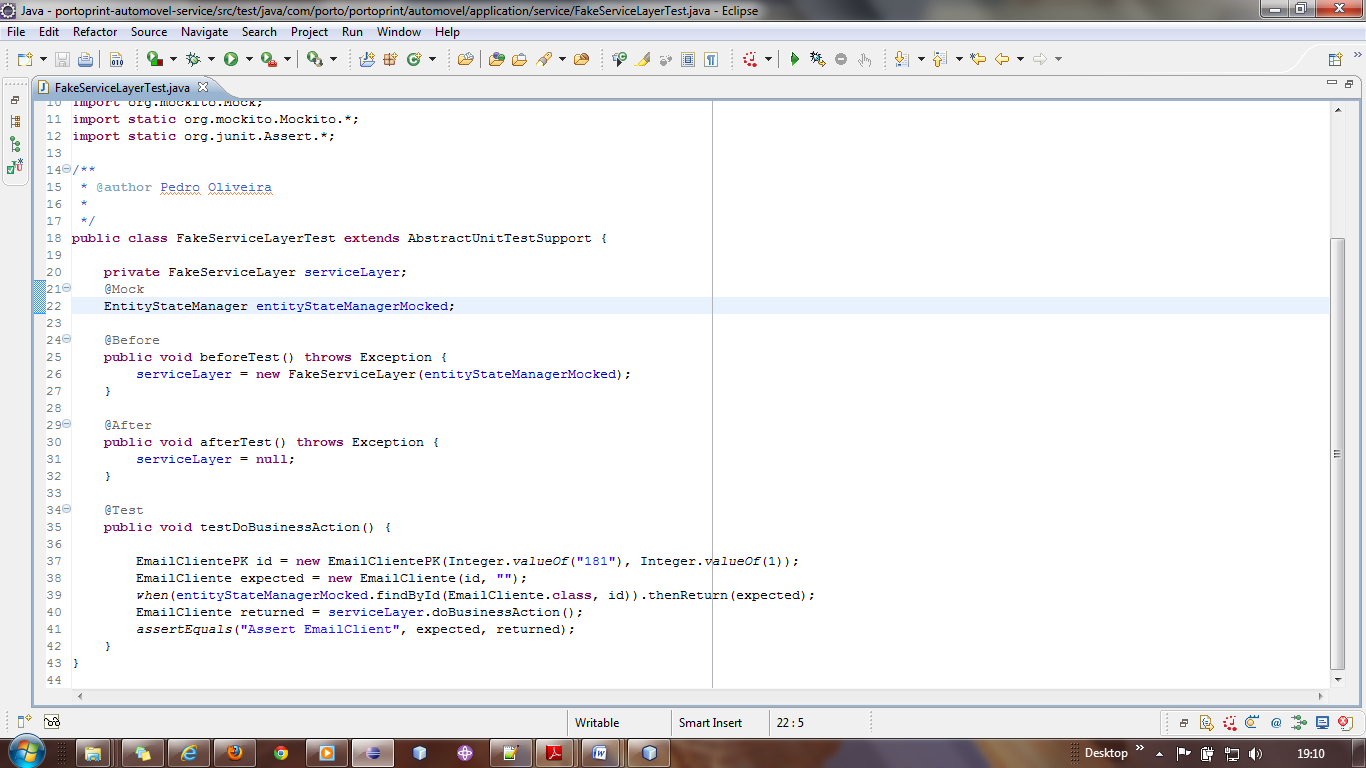
Exemplo: Meu método retornou o objeto certo?

* Verificar se o seu método lançou a excessão conforme esperado.

Exemplo: quando passado null, lançou IllegalArgumentException?

Qual mensagem da excessão?

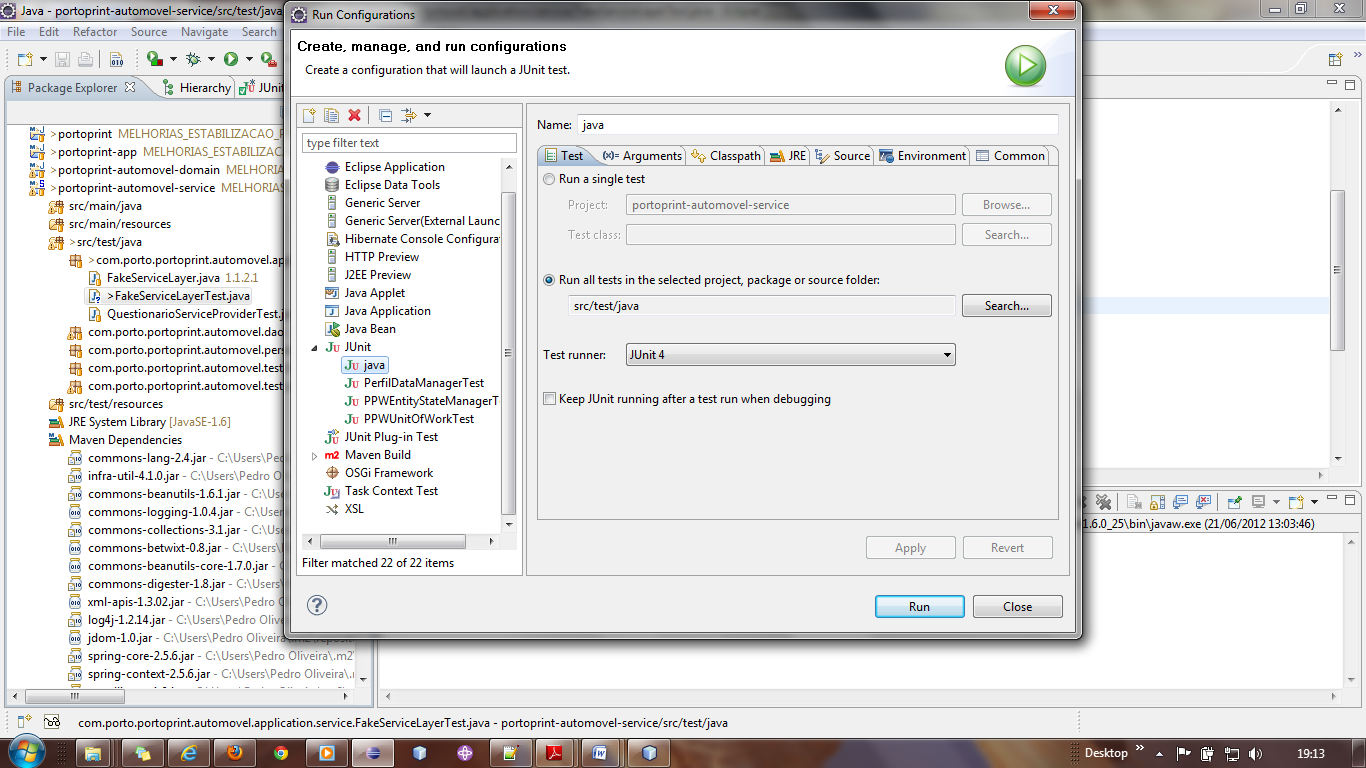
A estrutura do seu teste deve ficar conforme a figura abaixo:



1. Certifique-se que está utilizando o JUnit4 como configuração do Run.

Selecione sua classe de teste com botão direito > Run As.. > Run Configurations...

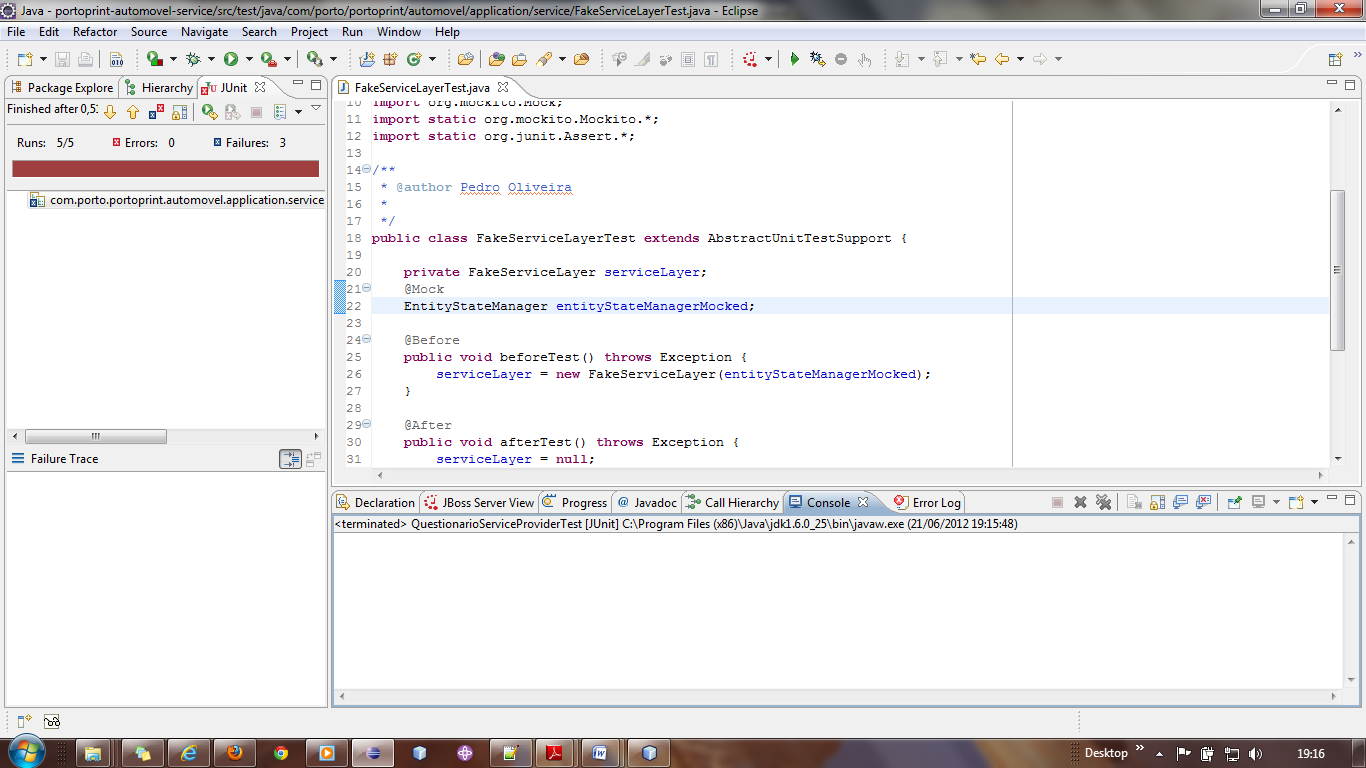
A configuração das classes java devem estar conforme a figura:



1. Rode o seu teste e verifique que o mesmo falhou (Barra Vermelha!).

**Caminho:** Selecione Run As.. > JUnit.

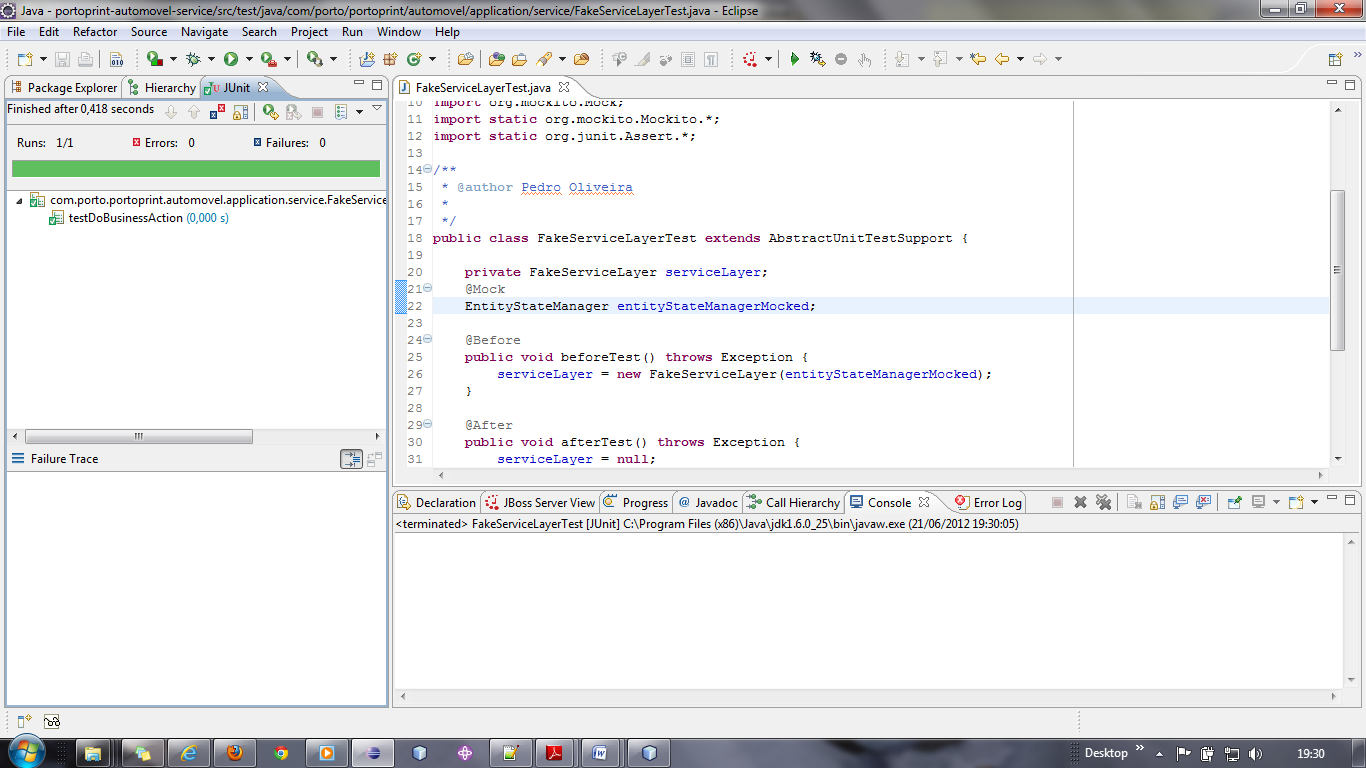
Seu teste inicialmente deve falhar, pois você ainda não implementou a sua classe testada.



1. Realize a implementação do método na sua classe testada(Nesse caso foi a classe FakeServiceLayer).

Sua implementação deve respeitar o que foi definido na sua classe de teste, fazendo com que o teste que está falhando passe com sucesso!

1. Rode novamente seu teste e confirme que o mesmo passou! (Barra JUnit Verde!)



Nesse momento você pode repassar o que foi testado.

Procure garantir que a sua implementação foi validada de forma consistente, não esqueça de verificar se a sua implementação atende exatamente o que é preciso.

Se você encontrar pontos que faltaram testar, reformule seu teste e repita o processo até garantir que tudo esteja sendo testado.

### Explicando de forma rápida a linguagem fluente do Mockito.

Utilizando o Mockito conseguimos criar comportamento para as interfaces "Mockadas" de forma simples e fluente. Abaixo descrevo alguns exemplos que você pode serguir para testar os comportamentos esperados pelas dependências das suas classes.

Para utilizar o Mockito, todas as dependências verificadas da sua classe precisam ser Interfaces.

Verificando Argumentos:

//Verifique na interface mock se o método someMethod foi invocado com os parâmetros: anyInt(), anyString(), **eq("third argument")**

verify(mock).someMethod(anyInt(), anyString(), **eq("third argument")**);

Verificando Número de chamadas das dependências:

//Verifique na interface mockedList se o método .add com o parâmetro "tree times" foi invocado pelo menos uma única vez.

verify(mockedList, atLeastOnce()).add("three times");

//Verifique na interface mockedList se o método .add com o parâmetro "five times" foi invocado pelo menos duas vezes.

verify(mockedList, atLeast(2)).add("five times");

//Verifique na interface mockedList se o método .add com o parâmetro "three times" foi invocado no máximo 5 vezes.

verify(mockedList, atMost(5)).add("three times");

Verificando a ordem das chamadas das dependências:

//Verifique se o método .add das interfaces firstMock e secondMock foram chamados na sequencia com os parâmetros "first" e "second".

InOrder inOrder = inOrder(firstMock, secondMock);

inOrder.verify(firstMock).add("first");

inOrder.verify(secondMock).add("second");

Adicionando um retorno padrão quando um método da dependencia for chamado:

//Quando o método someMethod da interface mock for chamado com os argumentos "some arg" retorne as Strings "one", "two", "three"

when(mock.someMethod("some arg")).thenReturn("one", "two", "three");

### Criando um teste funcional para testar a camada de persistencia do PPW.

Para o PortoPrintWeb montamos uma estrutura para simular as queries realizadas pelos DataManagers.

Para criar classes que precisam testar "Criterias/Queries" e como as classes de acesso a dados irão se comportar em conjunto com o Hibernate, sua classe de teste deve extender a classe: **AbstractFunctionalTestSupport**.

Quando essa classe de teste for executada um contexto de teste irá ser carregado pelo JUnit, inicializando o Spring de forma semelhante ao que ocorre em produção.

Para cada método de teste uma transação do hibernate será inicializada permitindo que as queries na base de teste do PPW sejam realizadas.

Ao extender a classe **AbstractFunctionalTestSupport** você deverá implementar o método prepareDataBase() chamando dentro dele o método addToDatabase(final Object entity) passando como parâmetro os ORMs preenchidos que serão utilizados para realizar o teste com o Hibernate.

Após a execução de cada método de teste, os comandos realizados na base serão dado rollback().

### Documentações de Referências.

**JUnit API:** Pode ser utilizada para consultar os asserts e anotações que podem ser utilizados nos testes unitários.

<http://junit.sourceforge.net/javadoc/index.html?org/junit/runners/JUnit4.html>

**Mockito:** A documentação rápida do Mockito pode ser utilizada para se obter exemplos de como os Mock Objects podem ser criados de forma a suprir as necessidades dos seus testes.

<http://docs.mockito.googlecode.com/hg/latest/org/mockito/Mockito.html>