

JUnit

1. Introdução a teste unitários e técnicas de teste

Testes de Unidade

- Forma de **Execução Controlada** para repetições;
- Pode revelar defeitos ou inconsistências na aplicação;
- Demonstra e avalia a qualidade do código;
- Mostra os problemas no melhor momento para “descobrir falhas”.
 - Antes que seja validado pelo cliente.

Testes quanto a Granularidade

De acordo com tamanho e tipo de testes, podemos classificar em:

- **Unitário:**
 - Testes realizados em Componentes em individuais, tal como uma Classe, um método, uma regra.
- **Integração:**
 - Testes aplicados sobre Módulos, podendo ser sobre integração com o banco de dados ou mesmo entre a aplicação e um serviço.
- **Sistema:**
 - Teste realizado sobre o Conjunto de Módulos que formam o sistema.

Testes quanto às Técnicas

De acordo com as técnicas de testes, podemos classificar em:

- **Caixa-branca:**
 - Envolve teste diretamente no código, no nosso caso com a ferramenta JUnit, testando explicitamente o código implementado.
- **Caixa-preta:**
 - O teste é a nível de sistema, sem considerar a implementação.
- **Caixa-cinza:**
 - É uma forma híbrida. O teste ocorre a nível de sistema, mas o código pode ser depurado com o objetivo de validar valores em tempo de teste;

Testes quanto às Técnicas

De acordo com as técnicas de testes, podemos classificar em:

- **Caixa-branca:** → **Foco do Teste Unitário**
 - Envolve teste diretamente no código, no nosso caso com a ferramenta JUnit, testando explicitamente o código implementado.
- **Caixa-preta:**
 - O teste é a nível de sistema, sem considerar a implementação.
- **Caixa-cinza:**
 - É uma forma híbrida. O teste ocorre a nível de sistema, mas o código pode ser depurado com o objetivo de validar valores em tempo de teste;

Quando não é um teste unitário

Não podemos considerar um teste como unitário quando*:

- O Teste deve usar o banco de dados;
- A funcionalidade envolve comunicação através da rede;
- A funcionalidade acessa o sistema de arquivos;
- O Teste não puder ser executado concorrentemente;
- O testes exigir configuração de arquivos ou do ambiente para funcionar;

* Segundo Michael Feathers no “A Set of Unit Testing Rules”

Sobre o JUnit

JUnit é um Framework Open-Source desenvolvido por Kent Beck (XP) e Erich Gamma (GoF) para a criação de testes automatizados em Java.



Podemos encontrar muita informação e downloads em <http://junit.org>.

Sobre o JUnit

Suas características e motivações:

- Facilita a criação de testes em Java;
- Possui uso simples e funcional;
- Disponibiliza diversos métodos que auxiliam os testes;
- E, segue a especificação **xUnit**.

Princípios do xUnit

Há alguns princípios que definem a especificação xUnit:

- Cada teste deve rodar isoladamente de outros testes;
- Erros devem ser detectados e reportados a cada teste;
- Devem deixar claro quais testes devem ser rodados;
- Devem ser fáceis de adicionar novos testes;
- Devem nos ajudar a reduzir o custo de desenvolvimento promovendo o reuso.

Por que usar JUnit?

Quais as vantagens de utilizar JUnit?

- Já é amplamente utilizado em projetos Java, está maduro;
- É base para a vários dos outros frameworks (Arquillian, Selenium, etc);
- Roda em linha de comando, com Ant, com Maven, em IDEs, etc;
 - Tem boa integração com IDEs (Eclipse, NetBeans, etc).
- Possui métodos auxiliares como de pré e pós execução;
- Possui múltiplas e **extensíveis** implementações de ***Runner***.
 - Veremos no decorrer no curso sobre isso.

Conceitos

Alguns conceitos importantes para a criação de testes com *JUnit*:

- **Caso de teste:**
 - Teste isolado, podemos considerar cada método como um Caso de Teste
- **Suíte de testes:**
 - Envolve um Conjunto de Classes de teste, podendo representar por exemplo, uma funcionalidade ou um conjunto delas.
- **Teste Runner:**
 - Executor do Caso de Teste. É quem sabe como testar e como enviar a saída.
- **Resultado dos testes:**
 - Sumarização dos os resultados, com os sucessos, falhas ou erros.

JUnit

2. Desenvolvimento orientado a testes (TDD)

TDD: O que é?

É o Desenvolvimento de Software Orientado a Testes.

Ou mais do que isso, é uma filosofia, uma cultura de desenvolvimento.

A ideia principal é: “Escreva o código de **teste antes** da implementação!”

TDD: Roteiro de Execução

Passos para implementar funcionalidades utilizando o TDD:

1. Escrever os Testes;
2. Executar os testes e verificar as falhas;
3. Escrever o código;
4. Executar os testes para identificar sucesso;
5. Refatorar o código para corrigir defeitos e evoluir continuamente.

Se houver uma nova implementação, volte a etapa 1.

TDD: Vantagens

- Garante que todo o código está testado;
 - Ou pelo menos boa parte dele, sendo que código com teste tem mais garantias de funcionar.
- Ajuda na modelagem do sistema;
 - Inclusive pode indicar falhas de arquitetura.
- Execução rápida para verificação de integridade.
 - Feita automaticamente pelo servidor de integração ou na máquina do desenvolvedor.

TDD: Mitos e distorções

- Não temos tempo para testes;
 - Na verdade perde-se mais tempo corrigindo bugs posteriormente.
- Tudo irá demorar mais por causa dos testes;
 - Na verdade a automatização auxilia no desenvolvimento futuro, evitando tanto gasto de tempo manual.
- Testes manuais possam ser mais eficazes que ter cobertura automatizada.
 - Testes automatizados rodam em segundos! Mas percorrem o mesmo caminho sempre. Então os manuais ainda podem ser necessários.

TDD: Dicas

Algumas dicas para escrever um teste com TDD:

- Para saber o que testar, criar uma lista de tarefas antes de começar;
- Escreva uma classe e nesta um método para uma das tarefas;
- Na primeira execução o teste deve falhar, para então ser escrita a implementação que fará o primeiro sucesso;
- Mesmo que o teste passe, **refatore** o código identificando melhorias, falhas ou cenários de teste ainda não escritos.
 - Repita todo o procedimento começando pelo teste e fazendo falhar;
 - Faça isso para cada tarefa;

JUnit

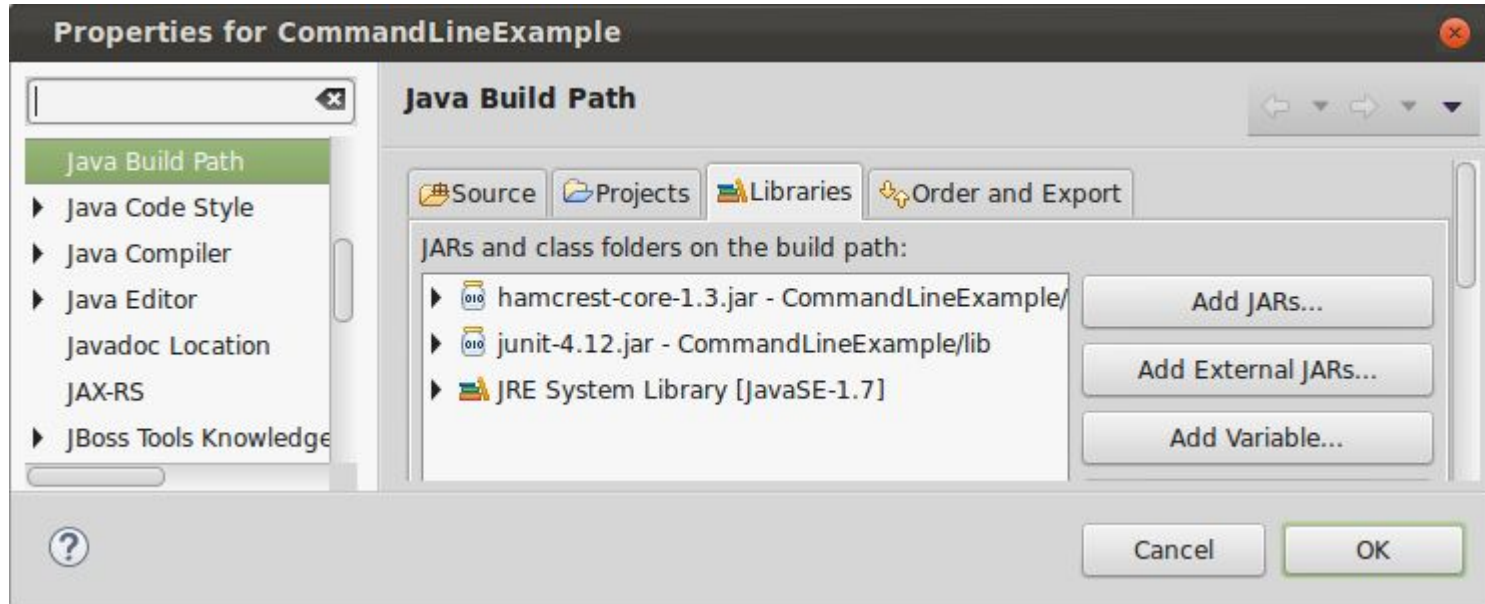
3. Instalação e Configuração do JUnit

Configurando com Biblioteca

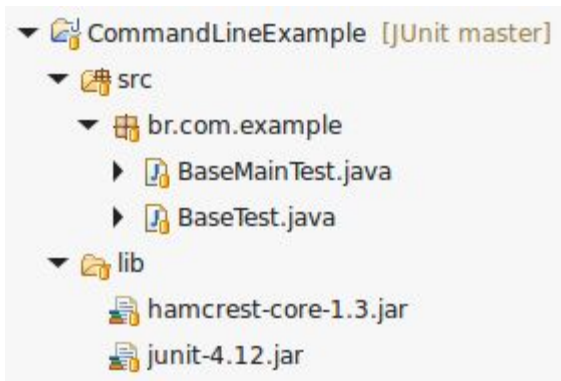
Para criar um projeto base com JUnit basta seguir os passos:

- Criar um projeto Java no Eclipse com *New > Java Project*;
- Adicionar bibliotecas do JUnit 4 ao *ClassPath* em *Properties > Java Build Path > Libraries > Add Library*
 - junit-4.12.jar
 - hamcrest-core-1.3.jar

Configurando com Biblioteca



Estrutura do Projeto Básico



- Vamos ao GitHub, baixar o projeto **CommandLineExample**;
 - /JUnit/Projetos/CommandLineExample;
- Importar com o eclipse diretamente do Git;
 - Ou clonar o repositório e importar como “Existing Eclipse Projects”;

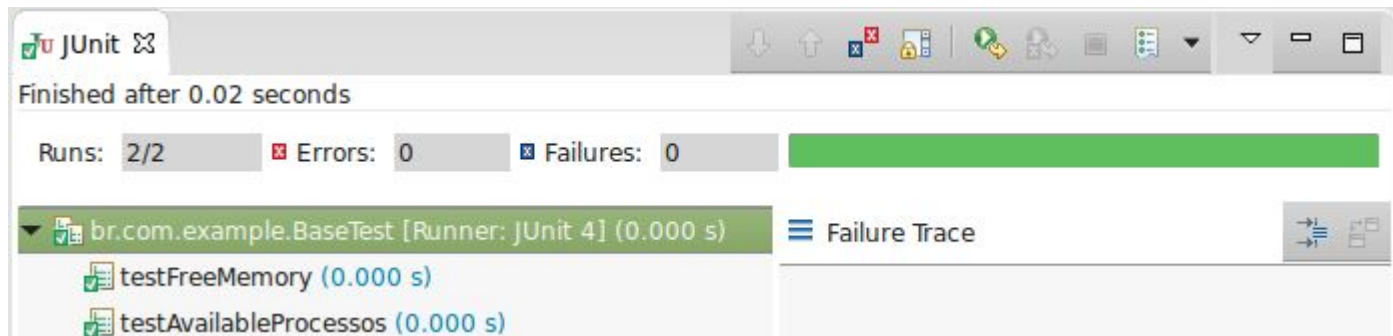
Configurando com Maven

- Converter projeto para Maven com *Configure > Convert to Maven Project*;
 - Ou criar diretamente um projeto Maven (caso de novo projeto);
- Adicionar dependência ao pom.xml

```
<dependency>  
  <groupId>junit</groupId>  
  <artifactId>junit</artifactId>  
  <version>4.12</version>  
  <scope>test</scope>  
</dependency>
```

Executando os Testes

- Para executar a Classe de Teste *br.com.example.BaseTest*;
 - Selecionar classe e acessar o *Run As... > JUnit Test*;
- A aba de testes é exibida com o resultado da execução:



JUnit

4. Criando casos de teste unitário

Vamos praticar

No projeto **CommandLineExample** que já está configurado para JUnit 4:

- Criar nova classe java *MeuPrimeiroTest*;
- Criar método de teste com um *Assert.assertTrue()*;
- Anotar método com *@Test*;
- Executar!

Asserts

- Auxiliam na validação do processamento dos testes
- Parâmetro *message*
 - Melhora a leitura dos resultados
 - Ex.: ***assertTrue***("Saldo insuficiente", *conta.isSaldoPositivo()*);
- Use *import static* para facilitar o uso dos **Asserts**
 -

```
import static org.junit.Assert.*;
```

Asserts

Tipos de Asserts:

- **assertArrayEquals:** compara cada elemento do array;
- **assertEquals:** compara se um elemento é igual a outro;
- **assertFalse:** verifica se a expressão é falsa;
- **assertNotEquals:** compara se o elemento é diferente de outro;
- **assertNotNull:** verifica se o elemento foi instanciado;

Asserts

Tipos de Asserts:

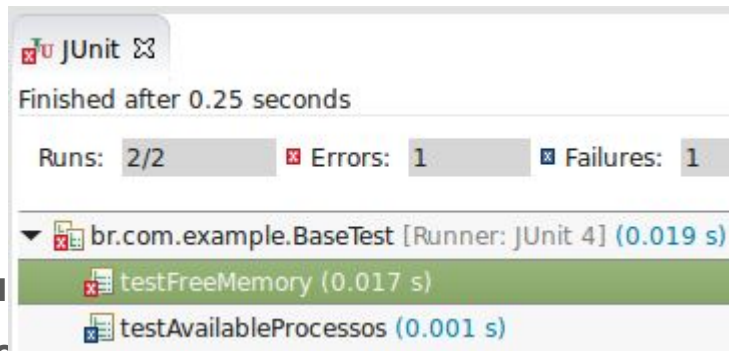
- **assertNotSame:** verifica se o elemento não é a mesma instância;
- **assertNull:** verifica se o elemento não foi instanciado;
- **assertSame:** verifica se o elemento é a mesma instância;
- **assertThat:** permite o uso de comparações personalizadas;
- **assertTrue:** verifica se a expressão é verdadeira.

Fail...

- Use o fail() para causar falha no teste por alguma condição tratada no próprio teste.

Fail é diferente de Exception.

- Fail indica que uma **asserção falhou**
- Exception indica que ocorreu um **erro** na implementação ou no teste!



Annotations

Várias anotações são fornecidas pelo JUnit, é importante saber utilizá-las. As anotações a seguir podem ser utilizadas **nos métodos de teste**:

- **@Test**
 - Indica que é um método a ser testado.
 - Pode ter um tempo limite, ex.: `@Test(timeout=3000)`.
- **@Ignore**
 - Indica que (temporariamente) o método deve ser ignorado.

Annotations

Estas anotações devem ser utilizadas **em métodos auxiliares** das classes de teste:

- **@Before:** Método chamado antes de cada teste;
- **@After:** Método chamado após de cada teste;
- **@BeforeClass:** Método chamado antes da classe;
- **@AfterClass:** Método chamado após a classe.

Annotations

Estas anotações podem ser utilizadas diretamente **nas classes de teste**:

- **@RunWith:**
 - Indica qual implementação do executor queremos usar;
 - Por padrão é atribuído o BlockJUnit4ClassRunner.class (> 4.4).
- **@SuiteClasses:**
 - Agregador de casos de teste.

Annotations

Estas anotações podem ser utilizadas diretamente **nas classes de teste**:

@FixMethodOrder:

- Altera a ordem padrão de execução dos métodos.
- MethodSorters.**DEFAULT**: maneira padrão determinístico do JUnit.
- MethodSorters.**JVM**: deixa a JVM determinar.
- MethodSorters.**NAME_ASCENDING**: pelo nome do método de teste.

Exercício 1

Na classe de testes `FuncionarioTest` do projeto `PetJUnit`, adicionar testes para o método `com.targettrust.model.Funcionario.getSalario(float valorHora)` informando diferentes valores de valor hora, como 50.

Para o *assert*, usar um `assertEquals` como este:

```
Assert.assertEquals(salarioTotalCalculado, salarioCorreto , 1);
```

Matchers

- Provido pela biblioteca Hamcrest que é uma dependência do JUnit;
- Criados para facilitar o uso de testes deixando a sintaxe mais legível;
- Composta por diversos métodos que agregam semântica;

Vamos observar o exemplo usando Matchers:

```
Assert.assertThat(alfabeto, Matchers.contains("a", "d", "f"));
```

- Os métodos `assertThat` e `contains` tentam trazer maior legibilidade.
 - Sem eles seriam necessários `assertEquals()` para cada letra.

Matchers

Métodos que podem ser utilizados com Matchers:

- **allOf():** Verifica se todas as condições são válidas;
- **notNullValue():** Testa se a variável não é nula;
- **nullValue():** Testa se a variável é nula;
- **is():** Testa se é algo? Tem equivalência com o equalsTo();
- **instanceOf():** Testa o tipo da classe;
- **sameInstance():** Considera a instância;
- **any():** Qualquer instância do classe informada;
- **anyOf():** Qualquer um destes;
- **anything():** Qualquer coisa;
- **equalTo():** Comparação;
- **not():** Inverte a sentença.

Exemplo

No projeto PetJUnit, o exemplo `com.targettrust.model.AgendaTest` utiliza os Matchers em sua construção ao invés dos tradicionais `assertTrue` e `equals`;

O uso de Matcher deixa o código mais semântico e permite aninhamento dando muito poder aos asserts. Permitindo estruturas complexas como no método *`com.targettrust.model.AgendaTest.testIncluiAgendaAninhados()`*:

```
Assert.assertThat(agenda.getConsultas().get(0).getAnimal().getEspecie(),
    CoreMatchers.anyOf(
        CoreMatchers.is(CoreMatchers.nullValue()),
        CoreMatchers.is(consultaInclusao.getAnimal().getEspecie()),
        CoreMatchers.allOf(
            CoreMatchers.any(Especie.class),
            CoreMatchers.notNullValue(),
            CoreMatchers.is(consultaInclusao.getAnimal().getEspecie()))));
```

Exercício 2

Na classe de teste `AgendaTest` do projeto `PetJUnit`, adicionar testes para o método `com.targettrust.model.Agenda.consultarPorApelido(String)` onde basta informar o apelido do usuário para retornar a consulta relacionada.

Para o assert desta vez utilizaremos matchers, tal como:

```
Assert.assertThat(
    consultaRetornada,
    CoreMatchers.anyOf(CoreMatchers.notNullValue()));
```

Tratamento de exceções

- Utilização:
 - Modifique para `@Test(expected=Exception.class)`.
 - Sendo `Exception.class` a classe da exceção esperada.
- Para casos onde é necessário verificar a mensagem da exceção:
 - Use try catch com **fail()**.
- Pode-se usar ainda a **@Rule** `ExpectedException`.

Exemplo

No projeto PetJUnit, o exemplo `com.targettrust.model.FornecedorTest` utiliza duas formas de tratamento de exceção:

1. Na primeira forma, no método `testCnpjNuloTestandoMensagem()` faz-se o controle utilizando *try catch* com auxílio do assert `fail()`.
2. A outra forma de fazer é utilizando o `expected`, como no método `testCnpjNulo()` onde é passado um argumento inconsistente para o método falhar.

Exercício 3

Na classe de teste `AgendaTest` do projeto `PetJUnit`, adicionar testes para o método `com.targettrust.model.Agenda.incluiAgenda(Agenda)` mas deixando o primeiro parâmetro `NULO` correspondente ao animal. Desta forma:

```
agenda.incluiAgenda(new Consulta(null, new Date(), false, 70F));
```

Assim uma exceção será disparada, trate para que o teste passe, conforme os exemplos vistos anteriormente.

Este exercício não é preciso de *assert*, uma vez que uma exceção será disparada, se ocorrer conforme esperado, e o processamento irá parar.

Rules

- Semelhante a um interceptor;
- Modificar o comportamento do teste.
 - “Injeta” código antes e depois deste.
 - Ou se o teste passa ou falha (TestWatcher).
- Pode substituir a criação de Runners mais simples.
- Rules estão disponíveis a partir do JUnit 4.7.

Rules

Algumas das Rules mais conhecidas:

- **TemporaryFolder:** Cria pastas e arquivos para serem utilizados no teste;
- **ExpectedException:** Acumular exceções aninhadas;
- **ExternalResource:** Permite definir métodos de inicialização e finalização de recursos como servers;
- **ErrorCollector:** Base para agrupamento de erros;
- **TestWatcher:** Base para monitorar estado dos testes e adicionar ações nestes pontos.

Executores (Runners)

- Implementação responsável por executar as classes de teste.
 - Quem sabe o que e como fazer.
- IDEs possuem seus próprios Runners.
 - Com interface gráfica com cores representativas.
- Alguns exemplos:
 - Suite, SpringJUnit4ClassRunner, MockitoJUnitRunner, HierarchicalContextRunner, **Parameterized...**

Parameterized Runner

Passos:

- Anotar a classe com `@RunWith(Parameterized.class)`.
- Criar um método que retorna uma coleção de valores para o teste.
 - Anotar método com `@Parameters`.
- Criar as variáveis necessárias para cada item da coleção anterior.
 - Anotar com `@Parameter`.

Exemplo

No projeto TarifaTaxi, o exemplo *br.com.targettrust.taxi.util. CalculaTarifaParameterizedTest* utiliza o *Runner Parameterized*, de forma que múltiplos valores serão usados para cada método.

O método `getCasosTeste()` irá fornecer os valores de entrada para cada teste.

```
@Parameters(name="Teste nro {index}: Calculando {1} km: {2}")
public static Collection<?> getCasosTeste() {
    return Arrays.asList(new Object[][] {
        { Cidade.PORTO_ALEGRE, 7.8, "22.15", "30.26"},
        { Cidade.MANAUS, 50, "124.00", "149.00"},
        { Cidade.PORTO_ALEGRE, 63, "146.90", "212.42"},
        { Cidade.PORTO_ALEGRE, 0, "4.52", "4.52"},
        { Cidade.SAO_PAULO, 63, "177.75", "199.80"} });
}
```

Exemplo 2

- Ainda no *CalculaTarifaParameterizedTest*, foi utilizado um `@Rule` que intercepta cada teste, altera o Locale e registra no console se o mesmo falhou ou teve sucesso.
- ```
@Rule
public LocaleRule localeRule = new LocaleRule();
```

 a por e-mail ou gerar um log mais elaborado em caso de falha, por exemplo.

# JUnit

## 5. Planejando os testes



## Planejando os testes

- Inicialmente é necessário identificar os cenários de teste;
- Da mesma forma que a aplicação, testes devem ter:
  - Uma organização lógica, tal como as classes de negócio.
  - Uma arquitetura padronizada, o framework deve considerar também *helpers* para classes de teste ou classes utilitárias.
- Organizar as classes em grupos maiores, que podem ser rodados de uma só vez, através de *Suítes*.

# Suítes

- Agrupa conjuntos de classes que podem formar funcionalidades.
- É um *runner*, é necessário utilizar a anotação `@RunWith(Suite.class)`.
- Recebe como parâmetro as classes que fazem parte da suíte: Ex.:
  - `@SuiteClasses(AlunoRNTTest.class, DisciplinaRNTTest.class)`.

## Dica:

- É interessante que haja uma suíte principal na aplicação para invocar todos os testes.  
Pode-se chamar de `AllTests.java`.

# Exemplo

No projeto PetJUnit, no package *com.targettrust.model* temos uma suite que abrange todas as classes do pacote, ou seja, AgendaTest, FornecedorTest e FuncionarioTest. A estrutura é simples, conforme esta:

```
@RunWith(Suite.class)
@SuiteClasses({ AgendaTest.class, FornecedorTest.class, FuncionarioTest.class })
public class AllTests {
 A{
```

# JUnit

## 6. Arquitetura das classes

# Arquitetura das Classes

- O padrão de pastas de um projeto de testes segue a **estrutura Maven**, com uma pasta para o código da aplicação (src) e outra para o código de teste (test).
- Como as classes de teste estão relacionadas diretamente as classes de código testadas, utilizam o mesmo nome, com o sufixo Test.
  - Ex.: ProdutoRN.java resulta em ProdutoRNTest.java no mesmo pacote, mas separado pela pasta raiz test ao invés de src.

# JUnit

## 7. Boas práticas, dicas e padrões

# Boas práticas e dicas de uso

- Utilizar o nome dos métodos de teste iniciando por test.
  - Ex.: `public void testCalculaTarifa()`.
  - Utilizar nomes intuitivos ajudam a documentar o código.
- No nome das classes de teste utilizar o sufixo Test. Ex.: `TarifaTaxiTest`.
- Para as suítes, utilizar o sufixo Suit. Ex.: `FuncionalidadeSuit`.
- Utilizar uma pasta diferente para os fontes da aplicação e os testes:
  - `src/main/java` e `src/main/test`

# Boas práticas e dicas de uso

- Testes unitários devem ser pequenos e rápidos.
- Os Dados devem ser realistas.
- Deve-se pensar em testes que não vão funcionar.
  - Ou, teste tudo o que deve quebrar.
- Os testes devem ser isolados.
- Somente um assert deve ser usado por método.
- **Comece a pensar em código testável!**



# Boas práticas e dicas de uso

- Em classes com objetos comuns entre os métodos:
  - Utilize o método `setUp()` para inicializar objetos de classe compartilhados.
- Se precisar liberar recursos ao final, como fechar a conexão :
  - Utilize o método `tearDown()`.
- Se tiver necessidade de testar métodos `protected`:
  - Coloque o teste e a implementação a ser testada no mesmo pacote.
- Necessidade de testar métodos `private`
  - Isso normalmente significa falha na organização dos métodos. Refatore.

# Boas práticas e dicas de uso

- Bugs encontrados devem virar testes para que erros não voltem a ocorrer;
- Sempre considerar os testes durante o desenvolvimento;
- Automatize a execução dos testes no servidor de Integração Contínua;
  - Sempre que quebrar todos os envolvidos devem ser notificados;
- Lembrar de rodar os testes no ambiente local.

# JUnit

8. Analisando e refatorando o código  
a partir dos testes

# Refactoring

- O objetivo do Refactoring Contínuo é **melhorar o código** de forma a manter coesão a cada funcionalidade adicionada.
- Com o uso de testes, **umenta a segurança** de realizar grandes alterações que exigem *refactoring*, que podem ser muito mais complicadas de testar manualmente.
- Podem ser mapeadas situações de **falhas** corrigidas com testes de forma que **não voltem a ocorrer** durante um *refactoring* futuro (mesmo que seja um novo membro da equipe).

# Refactoring

Indicativos que o código precisa ser refatorado:

- Classes de teste muito grandes:
  - Podem indicar classes de negócio com muitas responsabilidades.
    - Classes de Negócio e Testes deveriam ser quebradas em classes menores.
- Método *setUp()* extenso ou com inicialização de muitas variáveis:
  - Verificar os métodos muito complexos e quebre-os em vários.
- Dependência entre os métodos:
  - Pode indicar problemas na modelagem do negócio ou que seu caso exige testes de integração e não apenas da unidade.

# JUnit

## 9. Mockando objetos

# Sobre os Mocks

- Abstraem a lógica real das dependências;
  - Ex.: WebServices, recursos não testáveis, outras camadas (bd)...
- Trabalham de forma a fornecer um *proxy* quando são utilizados e a implementação real não é invocada.
- Isolam a classe alvo do teste, concentrando os testes somente nela e simulando as classes relacionadas.

# Sobre o Mockito

- *Framework* baseado em JUnit para reproduzir testes com *mocks*.
- Possui um Runner próprio.
- O retorno dos métodos pode ser configurado.
- É possível monitorar a quantidade de chamadas aos métodos.



# Configurando com Maven

- Utilizaremos o **Mockito** para nossos exemplos, para usá-lo é necessário adicionar uma nova dependência.
- Adicionar dependência ao pom.xml

```
<dependency>
 <groupId>org.mockito</groupId>
 <artifactId>mockito-all</artifactId>
 <version>2.0.2-beta</version>
</dependency>
```

# Exemplo

No projeto PetJUnit, o exemplo com `com.targettrust.servico.AnimalRNTest` foi construído utilizando o Mockito. É necessário:

- Usar um Runner diferenciado: `@RunWith(MockitoJUnitRunner.class)`
- Declarar os objetos que serão mock com a anotação `@Mock`, tal como:
- Usar `AnimalDAO animalDAOMock;` para registrar comportamentos. Exemplo:

```
Mockito.when([MÉTODO A SER CHAMADO]).thenReturn([RESPOSTA]);
```

## Exercício 4

Na classe de teste *AnimalRNTest* do projeto PetJUnit, adicionar testes para o método *AnimalRN.listarAnimaisPorEspecie(Especie)* usando *Mock*. Deve-se criar uma lista com vários animais de diferentes espécies e passar para o *mock* com algo semelhante a:

```
Mockito.when(
 animalDAOMock.listarAnimaisCadastrados()
 .thenReturn(animaisParaMock);
```

Após isso invocar o método *listarAnimaisPorEspecie* informando uma das espécies e testando se o resultado confere.

# JUnit

## 10. Uso de ferramentas de análise

# PMD E CheckStyle

- PMD
  - Utilizado para encontrar más práticas que podem levar a problemas
  - Identifica código não otimizado para ser refatorado
  - Encontra pontos de alta complexidade que devem ser quebrados
- Checkstyle
  - Também faz um trabalho de busca de potenciais problemas
  - Procura por quebra de convenções, como falta do JavaDoc.
  - Possui sugestões de autocorreção para alguns casos.

# EMMA: Eclipse Plugin

- Monitoramento da cobertura
- Exibição visual dos pontos mais críticos

# Links Úteis e Referências

- JUnit - About  
<http://junit.org>
- Checkstyle  
<http://checkstyle.sourceforge.net>
- PMD  
<http://pmd.sourceforge.net/pmd-5.0.0>
- FindBugs  
<http://findbugs.sourceforge.net/>