Testes Unitários: JUnit

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Campus Rio Grande
Divisão de Computação

Agenda

Tipos de Teste

- **Teste de Integração**
  - Garante que um ou mais componentes combinados (ou unidades) funcionam.
- **Teste de Regressão**
  - Toda vez que algo for mudado, deve ser testada toda a aplicação novamente.
- **Teste Funcional**
  - Testar as funcionalidades, requerimentos, regras de negócio presentes na documentação. Validar as funcionalidades descritas na documentação (pode acontecer de a documentação estar inválida)

Tipos de Teste

- **Teste de Interface com o usuário**
  - Verifica se a navegabilidade e os objetivos da tela funcionam como especificados e se atendem da melhor forma ao usuário.
- **Teste de Carga**
  - Verifica o funcionamento da aplicação com a utilização de uma quantidade grande de usuários simultâneos.
- **Teste de Stress**
  - Testar a aplicação sem situações inesperadas. Testar caminhos, às vezes, antes não previstos no desenvolvimento/documentação.

Tipos de Teste

- **Teste de Segurança**
  - Testar a segurança da aplicação das mais diversas formas. Utilizar os diversos papéis, perfis, permissões, para navegar no sistema.
- **Teste de Configuração**
  - Testar se a aplicação funciona corretamente em diferentes ambientes de hardware ou de software.
- **Teste Unitário**
- E etc...

Teste Unitário

Teste Unitário

- O teste unitário é implementado com base no menor elemento testável (unidades) do software.
- Implica em testar a estrutura interna (como fluxo lógico e de dados), a função da unidade e os comportamentos observáveis.

O que é unidade?

- Componente, classe, método, etc.

Formas de Fazer Testes

Teste Manual

Executando testes de forma manual (Sem qualquer ferramenta de suporte)

- Onera muito tempo
- Tedioso
- Alto investimento em recursos humanos
- Menos reutilizável
- Não é programável

Formas de Fazer Testes

Testes Automatizados

Executando testes usando uma ferramenta.

- Maior velocidade na execução
- Menor investimento em recursos humanos
- Programável

Teste Unitário O que testar?

Sempre nós ficamos em dúvida sobre o que devemos testar em nossas classes, existem alguns macetes que podem nos ajudar a descobrir quais e quantos testes deverão ser escritos...

Teste Unitário O que testar?

- A principal regra para saber o que testar é: "Tenha criatividade para imaginar as possibilidades de testes".
- Comece pelas mais simples e deixe os testes "complexos" para o final.
- Use apenas dados suficientes (não teste 10 condições se três forem suficientes)
- Não teste métodos triviais, tipo *get* e *set*.
- No caso de um método set, só faça o teste caso haja validação de dados.
- Achou um bug? Não conserte sem antes escrever um teste que o pegue (se você não o fizer, ele volta)!

Como Definir um teste com JUnit

```java
import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class MyTests {

  @Test
  public void multiplicationOfZeroIntegersShouldReturnZero() {

    // MyClass is tested
    MyClass tester = new MyClass();

    // assert statements
    assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
    assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
    assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
  }

}
```

Instalação

A instalação pode ser pelo Maven ou por download.

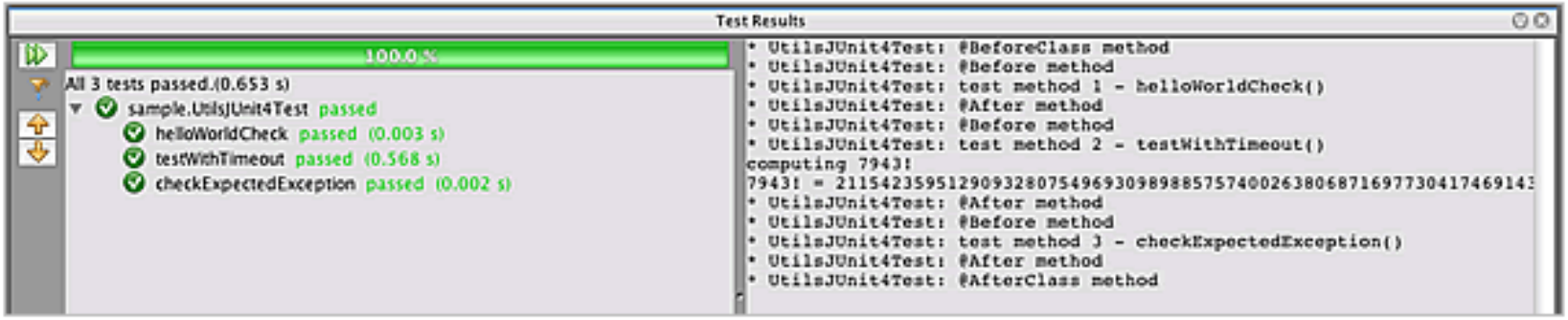
```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

Figure: Maven

```
http://junit.org/
```

Figure: Site

Execução



Nesta imagem (clique na imagem para ampliá-la) você pode notar que o IDE executou o teste JUnit em Utils...

Anotações

- **@Test**: determina que um método deve ser executado como caso de teste
  - O método deverá ser *public* e *void*
- **@Ignore**: Anotação é usada para ignorar um teste (o teste não será executado)

Asserções

Statement	Description
fail(message)	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional.
assertTrue([message,] boolean condition)	Checks that the boolean condition is true.
assertFalse([message,] boolean condition)	Checks that the boolean condition is false.
assertEquals([message,] expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([message,] expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message,] object)	Checks that the object is null.
assertNotNull([message,] object)	Checks that the object is not null.
assertSame([message,] expected, actual)	Checks that both variables refer to the same object.
assertNotSame([message,] expected, actual)	Checks that both variables refer to different objects.

@BeforeClass, @AfterClass, @Before e @After

- **@BeforeClass**: marca o método de inicialização da classe de teste. O método é executado somente uma vez (antes de qualquer outro método da classe de teste)
- **@AfterClass**: anota um método que irá finalizar a classe de teste. O método é executado apenas 1 vez e depois que todos os outros métodos da classe de teste foram finalizados.

@BeforeClass, @AfterClass, @Before e @After

- **@Before**: anota um método como um método de inicialização de teste. É executado antes de cada método de teste.
  - Um método de inicialização de teste não é obrigatório para a execução de testes, mas se você precisar inicializar algumas variáveis antes de executar um teste, você poderá usar essa anotação.
- **@After**: a anotação marca um método de finalização de teste. O método anotado com @After é executado depois de cada caso de teste na classe de teste.

@BeforeClass e @AfterClass

Métodos de inicialização e encerramento da classe de teste.

```
@BeforeClass
public static void setUpClass() throws Exception {
}

@AfterClass
public static void tearDownClass() throws Exception {
}
```

São usadas para marcar métodos que devem ser executados antes e depois da classe de teste. (DEVEM ser estáticos)

Exemplo: Esperando uma Exceção

```java
package com.vogella.junit.first;

public class MyClass {
  public int multiply(int x, int y) {
    // the following is just an example
    if (x > 999) {
      throw new IllegalArgumentException("X should be less than 1000");
    }
    return x / y;
  }
}
```

Exemplo: Esperando uma Exceção

```java
package com.vogella.junit.first;

import static org.junit.Assert.assertEquals;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;

public class MyClassTest {

  @Test(expected = IllegalArgumentException.class)
  public void testExceptionIsThrown() {
    MyClass tester = new MyClass();
    tester.multiply(1000, 5);
  }

  @Test
  public void testMultiply() {
    MyClass tester = new MyClass();
    assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));
  }
}
```

Timeout

Para definir o timeout para interromper o teste.

```
@Test(timeout=1000)
public void testWithTimeout() {
    final int factorialOf = 1 + (int) (30000 * Math.random());
```

Há testes que podem demorar muito.

JUnit no NetBeans

Exemplo: Classe Matemática com os seguintes métodos:

- Multiplicação
- Divisão
- Subtração
- Raiz
- E etc...

Testes Unitários: JUnit

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Campus Rio Grande
Divisão de Computação