1 - Caso de Teste Unitário no JUnit

Um Caso de Teste de Unidade (Unit Test Case) é uma parte do código, usada para garantir que outra parte do código (método) funciona como esperado. Para alcançar os resultados desejados de forma rápida, uma estrutura de teste é necessária. O JUnit é um framework de teste de unidade perfeita para a linguagem de programação Java.

A estrutura de um Caso de Teste de Unidade formal é caracterizada por uma entrada e uma saída conhecida e esperada, pois cabe ao programador saber antes de programar o teste qual é o resultado correto de uma operação (resposta do método para determinados parâmetros). A entrada conhecida deve testar uma condição prévia e a saída esperada deve testar uma pós-condição.

Devem haver pelo menos dois Casos de Teste de Unidade para cada requisito (método) - um teste positivo e um teste negativo. Se um requisito tem sub-requisitos, cada sub-requisito deve ter pelo menos dois casos de teste como positivo e negativo.

Para verificarmos estes conceitos teremos antes de aprender a criar as classes de teste do tipo JUnit e entender a sua codificação. Então recomenda-se seguir os passos da Seção 2 - Exemplo Aula2 - que estão na sequência deste documento - para testar os conceitos a serem apresentados, por enquanto programe somente os Passos 1 e 2.

1.1 Anotações da classe Test Case

Anotações: são como metadados adicionados ao código para prover informações as classes e membros da classe.

Para testar as anotações faça o Passo 3 da seção Exemplo Aula2.

No JUnit usa-se as seguintes anotações nos métodos, da classe de teste, para prover as seguintes informações:

- @Test: todo método marcado com essa anotação será chamado ao executar a classe de teste;
- @BeforeClass: todo método marcado com essa anotação será chamado antes de qualquer teste. Eles são chamados antes mesmo do construtor, por este motivo eles só podem estar em métodos estáticos;
- @AfterClass: todo método marcado com essa anotação será chamado após todos os testes. Esses métodos precisam ser estáticos;
- @Before: este método será chamado antes de cada método de teste. Este método pode ter qualquer nome, mas é comum usar o nome setUp.

  Como exemplo, neste método pode ser alocado algum recurso que será consumido no método de @Test;
- @After: este método será chamado após cada método de teste. Este método pode ter qualquer nome, mas é comum usar o nome tearDown.

  Como exemplo, neste método pode estar o código para liberar recursos alocados no método @Before;
- @Ignore: todo método marcado com esta anotação será ignorado pelo teste. Ele é útil quando o método a ser testado ainda não está pronto.

  Esta anotação pode ser usada também na classe para ignorar todos os métodos de teste.

Observações:

- Todos os métodos marcados pelas anotações precisam ser públicos;
- Os método marcados pelas anotações @BeforeClass e @AfterClass precisam ser estáticos, já os métodos marcados pela outras anotações **não** podem ser estáticos;
- Na classe de teste podem ter métodos que não possuem as anotações.

A Figura 7 mostra o resultado do teste da classe NossoTest da Figura 6. Observe a sequência como os métodos foram invocados pelo JUnit.

1.2 Classe Assert

Esta classe provê um conjunto de métodos estáticos para testar asserções.

> *Em computação, asserção (em inglês: assertion) é um predicado que é inserido no programa para verificar uma condição que o desenvolvedor supõe que seja verdadeira em determinado ponto* (https://pt.wikipedia.org/wiki/Asserção).

Essa classe se encontra no pacote org.junit e os seus métodos são estáticos, desta forma, eles podem ser invocados diretamente a partir da classe:

```
Assert.assertEquals( “abc”, “abc”);
```

Porém recomenda-se fazer uma importação estática:

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

e chamar o método diretamente sem a classe:

```
assertEquals( “abc”, “abc”);
```

pois fica mais fácil a leitura do código pelo programador.

A classe Assert possui os seguintes métodos que são úteis para escrever testes. Esses métodos são usados para verificar falhas, ou seja, eles não lançam a exceção AssertionError quando o resultado do teste está correto:

- fail(String message) e fail(): estes métodos são usados criar uma falha, ou seja, ao colocar ele no corpo de um método de @Test ele irá acusar a falha. Veja como exemplo a Figura 8 e Figura 9. Para fazer esse teste programe o Passo 4 da seção Exemplo Aula2;
- assertEquals: este método é usado para testar se o resultado da operação confere com o valor esperado. Por ele estar sobrecarregado ele possui várias formas de ser chamado. Veja como exemplo a Figura 10. Para fazer esse teste programe o Passo 5 da seção Exemplo Aula2;
- assertTrue e assertFalse: estes métodos são usados para conferir se uma condição é true e false, respectivamente. Para fazer esse teste programe o Passo 6 da seção Exemplo Aula2;
- assertNull e assertNotNull: estes métodos são usados para conferir se um objeto é nulo. Para fazer esse teste programe o Passo 7 da seção Exemplo Aula2;
- assertSame e assertNotSame: estes métodos são usados para conferir se dois objetos referenciam o mesmo objeto. Para fazer esse teste programe o Passo 8 da seção Exemplo Aula2;
- assertArrayEquals: este método é usado para conferir se dois arrays possuem o mesmo conteúdo. Para fazer esse teste programe o Passo 9 da seção Exemplo Aula2.

O JUnit prove ainda as seguintes opções para tratar condições de teste. Para fazer esse teste programe o Passo 10 da seção Exemplo Aula2:

- @Test(timeout=1000): o teste irá falhar caso ele dure mais de 1 segundo;
- @Test(expected=Exception): o teste irá falhar caso ele não lance a exceção esperada ou um subtipo dela.

1.3 Testes Parametrizados

Os testes parametrizados permitem ao programador executar o mesmo teste com diferentes valores, ou seja, uma bateria de testes. Programe o Passo 11 da seção Exemplo Aula2.

Para programar um teste parametrizado a classe de teste precisa ter os seguintes requisitos:

i. A classe precisa ser anotada com @RunWith(Parameterized.class);
ii. A classe precisa ter um método estático anotado por @Parameters, que retorna uma coleção de objetos (como Array) com os parâmetros de teste (entrada) e resultado (esperado). Esses parâmetros serão usados para construir um objeto da classe;
iii. A classe precisa ter um construtor que recebe cada elemento do array retornado pelo método estático anotado;
iv. A classe precisa ter os **atributos** para manter os valores a serem usados no teste;
v. Na chamada do método de @Test tem-se de fazer o uso dos **atributos** da classe parametrizada.

2 - Exemplo Aula2

Nesta parte do texto são mostrados os passos para criar um projeto no Eclipse com alguns exemplos de como fazer testes de unidade.

Passo 1: Criar um projeto de nome Aula2 no Eclipse.

i. Crie um novo projeto no Eclipse acessando File -> New -> Project... e, na sequência, escolha Java Project. Nomeie o projeto de Aula2. Adicione a biblioteca JUnit 4 no projeto antes de finalizar a criação do projeto, assim como mostra a Figura 1;
ii. Os testes do projeto não devem ficar dentro do folder src, então crie um folder chamado test clicando com o botão direito do mouse sobre o nome do projeto Aula2 e acesse a opção New -> Source Folder. Ao final desse passo a estrutura do projeto estará assim como na Figura 2.

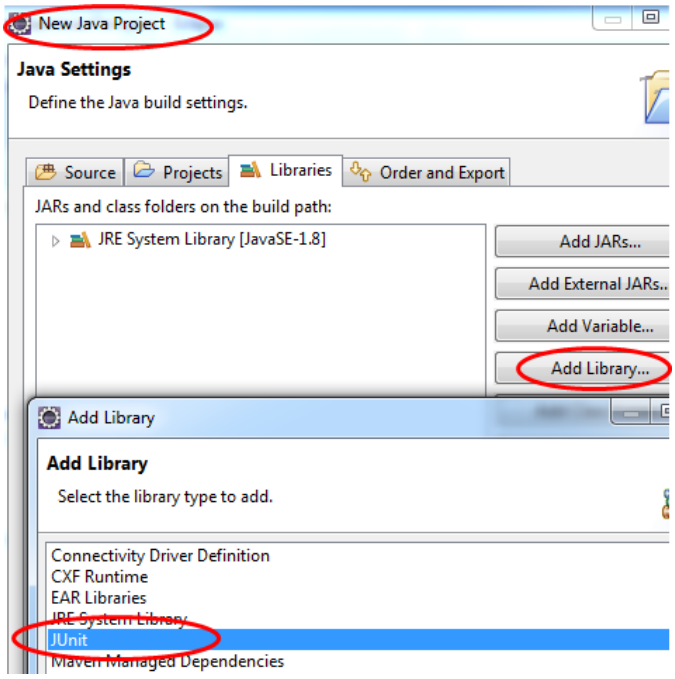

Figura 1 – Adicionar a biblioteca JUnit no projeto.

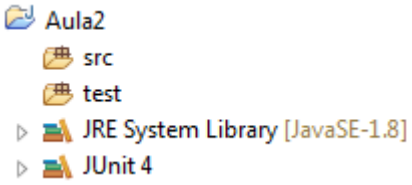

Figura 2 – Estrutura do projeto com a biblioteca JUnit e o folder test.

Passo 2: Criar uma classe de teste do JUnit.

i. Crie um pacote de nome aula no folder test. Lembre-se que basta clicar com o botão direito do mouse sobre o folder test e escolher New -> Package;

ii. Crie uma classe de nome NossoTest no pacote aula do folder test. Lembre-se que basta clicar com o botão direito do mouse sobre o pacote aula e escolher New -> JUnit Test Case. A Figura 3 mostra a estrutura atual do projeto e a Figura 4 mostra o código criado pelo Eclipse;

iii. Como o nosso objetivo é apenas entender os conceitos da classe de teste do JUnit então não iremos usar, por enquanto, o folder src (código fonte do projeto a ser entregue para o cliente). Para executar a classe NossoTest é necessário acessar o menu Run -> Run As -> JUnit Test. A **Erro! Fonte de referência não encontrada.** mostra a tela de resultados, a barra vermelha indica que algum teste falhou.

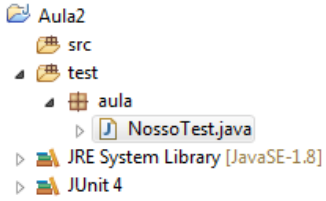

Figura 3 – Estrutura do projeto com a classe NossoTest.

```
package aula;

import static org.junit.Assert.*;

public class NossoTest {

    @Test
    public void test() {
        fail("Not yet implemented");
    }

}
```

Figura 4 – Código da classe NossoTest.

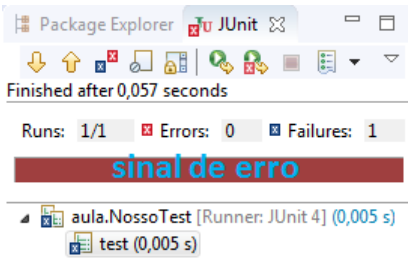

Figura 5 – Tela do Eclipse mostrando o resultado da classe NossoTest.

Passo 3: Programar as anotações da classe de teste.

i. Substitua o código da classe NossoTest pelo da Figura 6 e, na sequência, execute o caso de teste (Run -> Run As -> JUnit Test). A Figura 7 mostra o resultado, veja a sequência que os métodos foram invocados.

```
package aula;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;

public class NossoTest {

        @BeforeClass
        public static void inicio() {
```

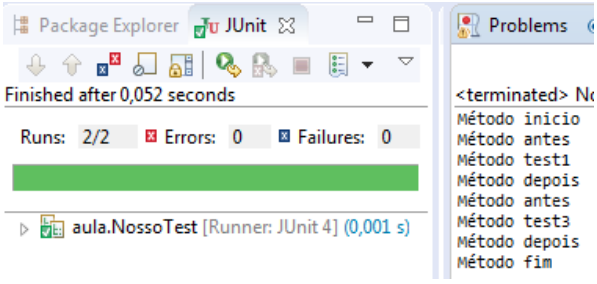

Figura 7 – Resultado do teste da Figura 6 no JUnit e no console do Eclipse.

```
                System.out.println("Método inicio");
        }

        @AfterClass
        public static void fim() {
                System.out.println("Método fim");
        }

        @Before
        public void antes(){
                System.out.println("Método antes");
        }

        @After
        public void depois(){
                System.out.println("Método depois");
        }

        @Test
        public void test1() {
                System.out.println("Método test1");

        }

        @Ignore
        @Test
        public void test2() {
                System.out.println("Método test2");
        }

        @Test
        public void test3() {
                System.out.println("Método test3");
        }
}
```

Figura 6 – Código para testar o uso das anotações na classe de teste.

Passo 4: Chamar os métodos fail(String message) e fail() da classe org.junit.Assert.

i. Substitua o código da classe NossoTest pelo da Figura 8 e, na sequência, execute o caso de teste. A Figura 9 mostra o resultado, veja que em ambos os casos o teste falhou, mas usando o método fail() não foi apresentada uma mensagem informativa.

```
package aula;

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

public class NossoTest {

    @Test
    public void test1() {
        System.out.println("Método test1: antes");
        fail("Faz o test1 falhar");
        System.out.println("Método test1: após");
    }

    @Test
    public void test2() {
        System.out.println("Método test2: antes");
        fail();
        System.out.println("Método test2: após");
```

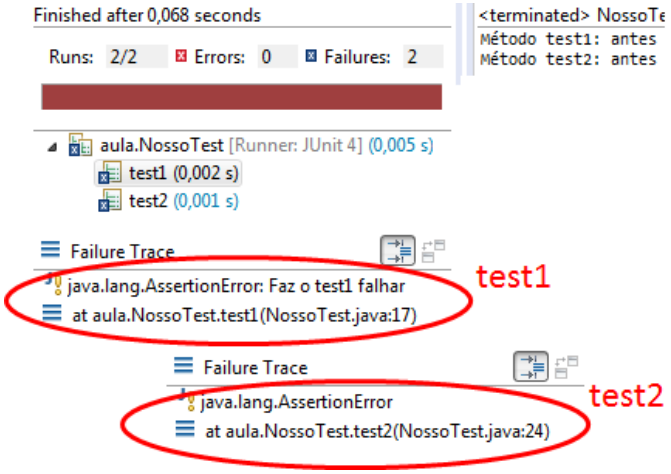

Figura 9 – Resultado do teste da Figura 8 no JUnit e no console do Eclipse.

```
    }
}
```

Figura 8 – Código para testar os métodos fail da classe Assertion.

Passo 5: Chamar os métodos assertEquals da classe org.junit.Assert.

i. Substitua o código da classe NossoTest pelo da Figura 10 e, na sequência, execute o caso de teste. A Figura 11 mostra o resultado, veja que o método assertEquals está sobrecarregado.

```
package aula;
import static org.junit.Assert.*;
import org.junit.Test;
public class NossoTest {
    @Test
    public void test1() {
        /* Parâmetros: mensagem, valor esperado, operação, delta
         * Não apresenta falha, pois será testado o intervalo
         * 3.5 < (2.1 + 2.1) < 4.5 */
        assertEquals("Msg test1", 4, 2.1+2.1, 0.5);
    }

    @Test
    public void test2() {
        /* Apresenta falha, pois será testado o intervalo
         * 4 < (2.1 + 2.1) < 4 */
        assertEquals("Msg test2", 4, 2.1+2.1, 0);
    }

    @Test
    public void test3() {
        /* Parâmetros: mensagem, valor esperado, operação
         * Apresenta falha, pois será testado
         * 4 == (3 + 3) */
        assertEquals("Msg test3", 4, 3+3);
    }

    @Test
    public void test4() {
        /* Parâmetros: valor esperado, operação, delta
         * Não apresenta falha, pois será testado
         * 3.5 < (2 + 2) < 4.5 */
        assertEquals(4, 2+2, 0.5);
    }

    @Test
    public void test5() {
        /* Parâmetros: valor esperado, operação
         * Apresenta falha, pois será testado
         * 4 == (3 + 3) */
        assertEquals(4, 3+3);
    }
}
```

Figura 10 – Código para testar os métodos assertEquals da classe Assertion.

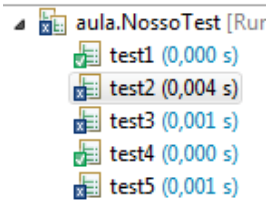

Figura 11 – Resultado do teste da Figura 10.

Passo 6: Chamar os métodos assertTrue e assertFalse da classe org.junit.Assert.

i. Substitua o código da classe NossoTest pelo da Figura 12 e, na sequência, execute o caso de teste. A Figura 13 mostra o resultado, veja que os métodos assertTrue e assertFalse estão sobrecarregados.

```
package aula;

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

public class NossoTest {
    @Test
    public void test1() {
        /* Parâmetros: mensagem, condição
         * Apresenta falha, pois será testado
         * (8 < 5) == true */
        assertTrue("Msg test1", 8 < 5);
    }

    @Test
    public void test2() {
        /* Parâmetros: condição
         * Não apresenta falha, pois será testado
         * (8 != 5) == true */
        assertTrue(8 != 5);
    }

    @Test
    public void test3() {
        /* Parâmetros: mensagem, condição
         * Não apresenta falha, pois será testado
         * (8 < 5) == false */
        assertFalse("Msg test3", 8 < 5);
    }

    @Test
    public void test4() {
        /* Parâmetros: condição
         * Apresenta falha, pois será testado
         * (8 != 5) == false */
        assertFalse(8 != 5);
    }
}
```

Figura 12 – Código para testar os métodos assertTrue e assertFalse da classe Assertion.

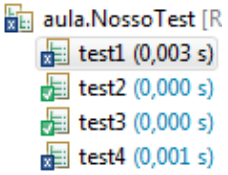

Figura 13 – Resultado do teste da Figura 12.

Passo 7: Chamar os métodos assertNull e assertNotNull da classe org.junit.Assert.

i. Substitua o código da classe NossoTest pelo da Figura 14 e, na sequência, execute o caso de teste. A Figura 15 mostra o resultado, veja que os métodos assertNull e assertNotNull estão sobrecarregados.

```
package aula;
import static org.junit.Assert.*;
import org.junit.Test;
public class NossoTest {
    @Test
    public void test1() {
        Object obj = null;
        /* Parâmetros: mensagem, condição */
        assertNull("Msg test1", obj);
    }

    @Test
    public void test2() {
        Object obj = new Object();
        /* Parâmetro: condição */
        assertNull(obj);
    }

    @Test
    public void test3() {
        Object obj = null;
        /* Parâmetros: mensagem, condição */
        assertNotNull("Msg test3", obj);
    }

    @Test
    public void test4() {
        Object obj = new Object();
        /* Parâmetro: condição */
        assertNotNull(obj);
    }
}
```

Figura 14 – Código para testar os métodos assertNull e assertNotNull da classe Assertion.

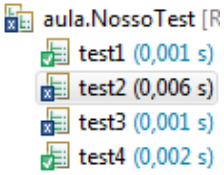

Figura 15 – Resultado do teste da Figura 14.

Passo 8: Chamar os métodos assertSame e assertNotSame da classe org.junit.Assert.

i. Substitua o código da classe NossoTest pelo da Figura 16 e, na sequência, execute o caso de teste. A Figura 17 mostra o resultado, veja que os métodos assertSame e assertNotSame estão sobrecarregados.

```
package aula;
import static org.junit.Assert.*;
import org.junit.Test;
public class NossoTest {
    @Test
    public void test1() {
        String a = null, b = null;
        /* Parâmetros: mensagem, objeto, objeto */
        assertSame("Msg test1", a, b);
    }

    @Test
    public void test2() {
        String a = "oi", b = "oi";
        /* Parâmetros: objeto, objeto */
```

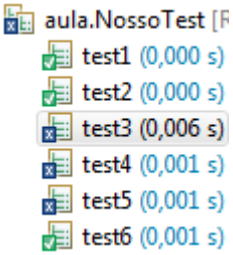

Figura 17 – Resultado do teste da Figura 16.

```
        assertSame(a, b);
    }

    @Test
    public void test3() {
        Object a = new Object(), b = new Object();
        assertSame(a, b);
    }

    @Test
    public void test4() {
        String a = null, b = null;
        /* Parâmetros: mensagem, objeto, objeto */
        assertNotSame("Msg test4", a, b);
    }

    @Test
    public void test5() {
        String a = "oi", b = "oi";
        /* Parâmetros: condição */
        assertNotSame(a, b);
    }

    @Test
    public void test6() {
        Object a = new Object(), b = new Object();
        assertNotSame(a, b);
    }
}
```

Figura 16 – Código para testar os métodos assertNull e assertNotNull da classe Assertion.

Passo 9: Chamar o método assertArrayEquals da classe org.junit.Assert.

i. Substitua o código da classe NossoTest pelo da Figura 18 e, na sequência, execute o caso de teste. A Figura 19 mostra o resultado, veja que os métodos assertSame e assertNotSame estão sobrecarregados.

```
package aula;
import static org.junit.Assert.*;
import org.junit.Test;
public class NossoTest {
    @Test
    public void test1() {
        String[] a = {"um","dois"}, b = {"três","quatro"};
        assertArrayEquals("Msg test1", a, b);
    }

    @Test
    public void test2() {
        String[] a = {"um","dois"}, b = {"um","dois"};
        assertArrayEquals( a, b);
    }
}
```

Figura 18 – Código para testar o método assertArrayEquals.

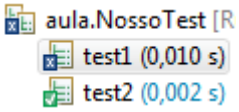

Figura 19 – Resultado do teste da Figura 18.

Passo 10: Testar as condições de timeout e exceção.

i. Primeiramente crie um pacote de nome aula no folder src. Lembre-se que basta clicar com o botão direito do mouse sobre o folder test e escolher New -> Package;

ii. Crie um classe de nome Operacao no pacote aula do folder src. Lembre-se que basta clicar com o botão direito do mouse sobre o pacote aula e escolher New -> Class. A Figura 20 mostra a estrutura atual do projeto;

iii. Programe o código da Figura 21 na classe Operacao;

iv. Substitua o código da classe NossoTest pelo da Figura 22 e, na sequência, execute o caso de teste. A Figura 23 mostra o resultado, veja como o resultado pode mudar com o uso dos parâmetros timeout e expected.

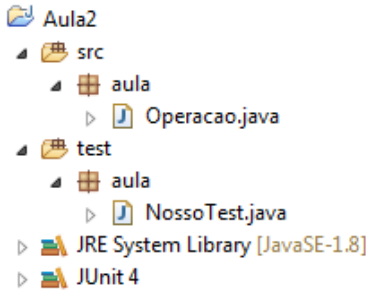

Figura 20 – Estrutura do projeto com a classe Operacao.

```
package aula;
import java.util.concurrent.TimeUnit;

public class Operacao {
    public int loopInfinito() {
        try{
            System.out.println("antes");
            /* sleep por 3 segundos */
            TimeUnit.SECONDS.sleep(3);
            System.out.println("após");
        }
        catch(Exception e){
            System.out.println("Exceção: "+ e.getMessage() );
        }
        return 1;
    }

    public double divisao(int a, int b){
        return a/b;
    }
}
```

Figura 21 – Código da classe Operacao.

```
package aula;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class NossoTest {
    private Operacao op;

    @Before
    public void setUp(){
        /* chamado antes de cada @Test */
        op = new Operacao();
    }

    @Test(timeout = 1000)
    public void test1() {
        /* Este teste falha porque ele dura mais de 1 seg. */
        assertEquals("Msg test1", 1, op.loopInfinito());
    }
```

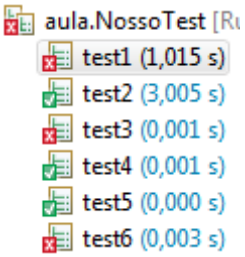

Figura 23 – Resultado do teste da Figura 22.

```
    @Test(timeout = 4000)
    public void test2() {
        /* Este teste não falha porque ele NÃO dura mais de 2 seg. */
        assertEquals("Msg test2", 1, op.loopInfinito());
    }

    @Test
    public void test3() {
        /* Este teste falha porque o método retorna uma
         * exceção e o teste espera um valor numérico*/
        assertEquals(1.25, op.divisao(5, 0),0);
    }

    @Test(expected=ArithmeticException.class)
    public void test4() {
        /* Este teste não falha porque o método
         * lança ArithmeticException*/
        assertEquals(1.25, op.divisao(5, 0),0);
    }

    @Test(expected=Exception.class)
    public void test5() {
        /* Este teste não falha porque Exception
         * é um super tipo para ArithmeticException */
        assertEquals(1.25, op.divisao(5, 0),0);
    }

    @Test(expected=Exception.class)
    public void test6() {
        /* Este teste falha porque NÃO lança Exception */
        assertEquals(1.25, op.divisao(5, 4),0);
    }
}
```

Figura 22 – Código para testar o timeout e exceções.

Passo 11: Criar uma classe parametrizada para testar diferentes valores de entrada.

i. Substitua o código da classe Operacao pelo da Figura 24;

ii. Substitua o código da classe NossoTest pelo da Figura 25 e, na sequência, execute o caso de teste. A Figura 26 mostra o resultado, veja que foram executados 5 testes.

```
package aula;

public class Operacao {

    public boolean isPar(int nro){
        return nro%2 == 0;
    }
}
```

Figura 24 – Código da classe Operacao.

```
package aula;

import static org.junit.Assert.*;
import java.util.Arrays;
import java.util.Collection;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

/* A classe precisa ser anotada com @RunWith(Parameterized.class) */
@RunWith(Parameterized.class)
```

```
public class NossoTest {
    /* A classe precisa ter os atributos que serão
     * usados no teste, que são a entrada e esperado. */
    private int entrada;
    private boolean esperado;
    private Operacao op;

    @Before
    public void initialize() {
        op = new Operacao();
    }

    /* Construtor que recebe cada elemento do array retornado
     * pelo método parametros.*/
    public NossoTest(int entrada, boolean esperado) {
        this.entrada = entrada;
        this.esperado = esperado;
    }

    /* Método estático anotado por @Parameters,
    * que retorna uma coleção de objetos (como Array)
    * com os parâmetros de teste (entrada) e resultado
    * (esperado). */
    @Parameterized.Parameters
    public static Collection parametros() {
        return Arrays.asList(new Object[][] {
                    { 1, false },
                    { 2, true },
                    { 5, false },
                    { 10, true },
                    { 0, false }  });
    }

    @Test
    public void test1() {
        System.out.println("Testando: " + entrada);
        assertEquals(esperado,  op.isPar(entrada));
    }
}
```

Figura 25 – Código da classe NossoTest para fazer um teste parametrizado.

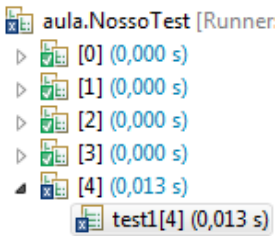

Figura 26 – Resultado do teste da Figura 25.