Para programar os testes primeiramente crie um projeto de nome Aula3 no IDE Eclipse, não esqueça de incluir a biblioteca JUnit. Na sequência, crie um source folder de nome test e adicione os pacotes de nome aula nos folders src e test, assim como mostra a Figura 1.

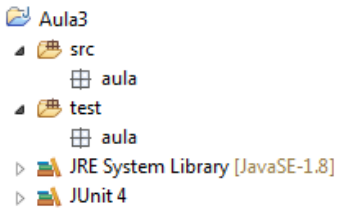

Figura 1 – Estrutura do projeto Aula3.

1 - Teste de Métodos Sem Retorno

Um método com retorno void é difícil de ser checado, já que ele não retorna um valor para ser comparado. O que pode ser feito é verificar se a ação foi efetuada pelo método. Veja como exemplo o método add da classe Vetor (Figura 2), como não é possível checar o seu retorno, então usou-se a instrução assertTrue(Vetor.existe(12)) (Figura 3) para conferir se o valor 12 foi colocado no array. A Figura 4 mostra o resultado do teste, veja que o test4 falhou porque o array lista da classe Vetor aceita somente 3 elementos.

```
package aula;
public class Vetor {
   private static Object[] lista = new Object[3];

   /* procura a 1a posição vazia no array e
      coloca o obj nela */
   public static void add(Object obj){
      for( int i = 0; i < lista.length; i++ ){
         if( lista[i] == null ){
            lista[i] = obj;
            break;
         }
      }
   }

   /* returna true se o obj existe na lista */
   public static boolean existe(Object obj){
      for( int i = 0; i < lista.length; i++ ){
         if( obj.equals( lista[i] ) ){
            return true;
         }
      }
      return false;
   }
}
```

Figura 2 – Código da classe Vetor.

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

public class VetorTest {
   @Test
   public void test4() {
      Vetor.add(12);
      assertTrue( Vetor.existe(12));
   }

   @Test
   public void test3() {
      Vetor.add("10");
      assertTrue( Vetor.existe("10"));
   }

   @Test
   public void test1() {
      Vetor.add(true);
      assertTrue( Vetor.existe(true));
   }

   @Test
   public void test2() {
      Vetor.add(2.5);
      assertTrue( Vetor.existe(2.5));
   }
}
```

Figura 3 – Código da classe VetorTest.

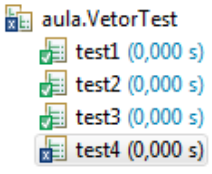

Figura 4 – Resultado do teste da Figura 3.

2 - Definindo a Ordem de Execução dos Testes

Os métodos @Test são invocados numa ordem aleatória definida pela API. Veja que no caso da Figura 3 os métodos não foram invocados na ordem que eles foram programados. Para definir uma ordem tem-se de anotar a classe com @FixMethodOrder:

- @FixMethodOrder(MethodSorters.JVM): os métodos de teste são invocados na ordem retornada pela JVM, esta ordem pode variar de uma execução para outra;
- @FixMethodOrder(MethodSorters.NAME_ASCENDING): os métodos de teste são invocados na ordem alfabética;
- @FixMethodOrder(MethodSorters.DEFAULT): os métodos de teste são invocados numa ordem aleatória.

A Figura 5 mostra a classe VetorTest usando a anotação @FixMethodOrder e a Figura 6 e Figura 7 mostram os resultados dos testes.

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

import org.junit.FixMethodOrder;
import org.junit.Test;
import org.junit.runners.MethodSorters;

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class VetorTest {
   @Test
   public void test4() {
      Vetor.add(12);
      assertTrue( Vetor.existe(12));
   }

   @Test
   public void test3() {
      Vetor.add("10");
      assertTrue( Vetor.existe("10"));
   }

   @Test
   public void test1() {
      Vetor.add(true);
      assertTrue( Vetor.existe(true));
   }

   @Test
   public void test2() {
      Vetor.add(2.5);
      assertTrue( Vetor.existe(2.5));
   }
}
```

Figura 5 – Código da classe VetorTest com a anotação @FixMethodOrder.

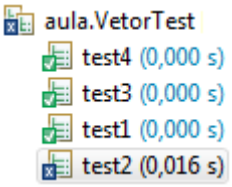

Figura 6 – Resultado do teste da Figura 5 usando MethodSorters.JVM.

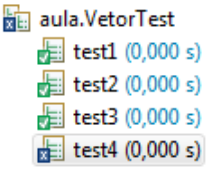

Figura 7 – Resultado do teste da Figura 5 usando MethodSorters.NAME_ASCENDING ou MethodSorters. DEFAULT.

3 - Testando Métodos Protegidos (protected)

Constitui boa prática manter os testes em uma estrutura de pacotes semelhantes a das classes a serem testadas. Contudo, métodos protegidos (protected) só podem ser acessados dentro do mesmo pacote ou pela herança. Então seria necessário manter a classe de testes no mesmo pacote que se encontra a classe a ser testada.

Porém, o JUnit executa métodos protegidos desde que a estrutura do folder test seja a mesma do folder src. Veja que no exemplo da Figura 2 e Figura 3 a classe Vetor se encontra em /src/aula/ e a classe VetorTest se encontra em test/aula/, ou seja, possui a mesma estrutura de pacotes.

Para testar um método protected altere a visibilidade do método existe da classe Vetor, assim como na Figura 8, e veja que o teste funciona normalmente. O método add também poderia ter a sua visibilidade alterada para protected.

Agora renomeie o pacote test/aula/ para, por exemplo, test/aula**x**/ e veja que o método existe não está visível na classe VetorTest, assim como mostra a Figura 9.

```
package aula;
public class Vetor {
   private static Object[] lista = new Object[3];
   public static void add(Object obj){
      for( int i = 0; i < lista.length; i++ ){
         if( lista[i] == null ){
            lista[i] = obj;
            break;
         }
      }
   }

   protected static boolean existe(Object obj){
      for( int i = 0; i < lista.length; i++ ){
         if( obj.equals( lista[i] ) ){
            return true;
         }
      }
      return false;
   }
}
```

Figura 8 – Código da classe Vetor com método existe protected.

```
1  package aulax;
2  import static org.junit.Assert.*;
3  import org.junit.Test;
4
5  import aula.Vetor;
6
7  public class VetorTest {
8     @Test
9     public void test4() {
10       Vetor.add(12);
11    The method existe(Object) from the type Vetor is not visible
12    }
13
14    @Test
15    public void test3() {
16       Vetor.add("10");
17       assertTrue( Vetor.existe("10"));
18    }
```

Figura 9 – Classe VetorTest no pacote aula**x**.

4 - Testando Métodos Privados (private)

Uma forma de testar um método privado é criar uma classe interna de teste, mas isso é muito intrusivo, pois a classe interna precisa estar dentro da classe a ser testada, que por vez, pode ter sido programada por outro desenvolvedor. Uma forma melhor é usar reflexão Java, pois a reflexão permite acessar membros privadas de uma classe.

> *Segundo a Wikipédia, reflection (reflexão) no contexto da ciência da computação, é a capacidade de um programa de observar ou até mesmo modificar a sua estrutura ou comportamento e que ocorre tipicamente em tempo de execução.*

A Figura 10 mostra como exemplo a classe Operacao com todos os métodos privados e a Figura 11 mostra o seu teste usando reflexão. A classe Operacao deverá estar no pacote /src/aula/ e a classe OperacaoTest no pacote /test/aula/, ou seja, a classe OperacaoTest não é interna a classe Operacao.

No código da Figura 11 fez se o uso de int.class, essa instrução pode causar estranheza, uma vez que, int é tipo de dado primitivo. Acontece que, a API java.lang.Class.class.isPrimitive() define nove classes pré-definidas para representar os oito tipos primitivos, mais o void. Estes são criados pela Máquina Virtual Java, e têm os mesmos nomes dos tipos primitivos que eles representam, ou seja, boolean, byte, char, short, int, long, float e double. Por este motivo, é possível chamar int.class, mesmo int sendo um tipo de dado primitivo.

```
package aula;
public class Operacao {
   private int soma(int a, int b){
      return a + b;
   }

   private int soma(int a, int b, int c){
      return a + b + c;
   }

   private boolean isPar(int a){
      return a%2 == 0;
   }

   private static int comprimento(String a){
      return a.length();
   }
}
```

Figura 10 – Código da classe Operacao com todos os métodos privados.

```
package aula;
import static org.junit.Assert.*;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import org.junit.Before;
import org.junit.Test;

public class OperacaoTest {
   private Operacao op;

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

   @Test
   public void testSoma2() {
      try{
         /* o método getDeclaredMethod recebe como parâmetro o nome do método (soma) e a lista de parâmetros do
            método. No caso do método soma são dois parâmetros do tipo int.
            O método getDeclaredMethod retorna um objeto do tipo Method */
         Method metodoSoma2 = Operacao.class.getDeclaredMethod("soma", int.class, int.class);
         /* o valor true indica que o objeto refletido deverá ter o modificador de visibilidade suprimido */
         metodoSoma2.setAccessible(true);
         /* chama o método soma passando os parâmetros 12 e 20. É necessário fornecer um objeto do tipo Operacao */
         int r = (int) metodoSoma2.invoke(op, 8, 12);
         /* checa se o resultado é 20 */
         assertEquals( 20, r, 0 );
```

```
      } catch (NoSuchMethodException|SecurityException|IllegalAccessException|
               IllegalArgumentException|InvocationTargetException e) {
         e.printStackTrace();
      }
   }

   @Test
   public void testSoma3() {
      try{
         Method metodoSoma3 = Operacao.class.getDeclaredMethod("soma", int.class, int.class, int.class);
         metodoSoma3.setAccessible(true);
         int r = (int) metodoSoma3.invoke(op, 8, 12, 5);
         /* checa se o resultado é 25 */
         assertEquals( 25, r, 0 );

      } catch (NoSuchMethodException|SecurityException|IllegalAccessException|
               IllegalArgumentException|InvocationTargetException e) {
         e.printStackTrace();
      }
   }

   @Test
   public void testIsPar() {
      try{
         Method metodoIsPar = Operacao.class.getDeclaredMethod("isPar", int.class);
         metodoIsPar.setAccessible(true);
         boolean r = (boolean) metodoIsPar.invoke(op, 11);
         /* checa se o resultado é false */
         assertFalse( r );

      } catch (NoSuchMethodException|SecurityException|IllegalAccessException|
               IllegalArgumentException|InvocationTargetException e) {
         e.printStackTrace();
      }
   }

   @Test
   public void testComprimento() {
      try{
         Method metodoComprimento = Operacao.class.getDeclaredMethod("comprimento", String.class);
         metodoComprimento.setAccessible(true);
         /* quando é estático pode-se fornecer o argumento null */
         int r = (int) metodoComprimento.invoke(null, "Bom dia");
         /* checa se o resultado é 7 */
         assertEquals( 7, r, 0);

      } catch (NoSuchMethodException|SecurityException|IllegalAccessException|
               IllegalArgumentException|InvocationTargetException e) {
         e.printStackTrace();
      }
   }

}
```

Figura 11 – Código para testar os métodos privados da classe Operacao.

5 - Testando Classes Internas

Uma classe interna é uma classe como outra qualquer, a diferença é ela se encontra dentro de outra classe, ou seja, para acessar a classe interna é necessário referenciar a classe/objeto externo. Veja como exemplo a classe interna Ponto na Figura 12. Para criar um objeto dessa classe é necessário ter um objeto do tipo externo (Geometria):

```
Geometria g = new Geometria();
Geometria.Ponto p = g.new Ponto(2, 2);
```

O teste de um método de uma classe interna é semelhante ao teste do método de uma classe normal. Veja como exemplo o código da classe PontoTest da Figura 13.

```
package aula;
import java.util.ArrayList;
import java.util.List;
public class Geometria {
   private List<Ponto> lista = new ArrayList<>();

   public void add(Ponto p){
      lista.add(p);
   }

   public void imprimir(){
      for( Ponto p:lista){
         System.out.println( p );
      }
   }

   public class Ponto{
      private double x, y;

      public Ponto(double x, double y){
         this.x = x;
         this.y = y;
      }

      public double distancia(Ponto p){
         return Math.sqrt( Math.pow(p.x - this.x, 2)
                       + Math.pow(p.y - this.y, 2) );
      }

      @Override
      public String toString() {
         return x + "," + y;
      }
   }
}
```

Figura 12 – Código da classe Geometria com a classe interna Ponto.

```
package aula;
import static org.junit.Assert.*;
import org.junit.Test;
import aula.Geometria.Ponto;
public class PontoTest {
   @Test
   public void test() {
      /* cria um objeto da classe externa */
      Geometria g = new Geometria();
      /* cria um objeto da classe interna */
      Geometria.Ponto p = g.new Ponto(2, 2);
      /* cria um objeto da classe interna */
      Geometria.Ponto pZero = g.new Ponto(0, 0);
      /* testa o método distancia */
      assertEquals( 2.82, p.distancia(pZero), 0.01 );
   }
}
```

Figura 13 – Código da classe PontoTest.

6 - Rules

As regras @Rule permitem adicionar ou redefinir o comportamento dos métodos de teste. Veja alguns exemplos de regras:

- A regra Timeout aplica o mesmo tempo limite para todos os métodos de teste da classe. No exemplo da Figura 14, o limite de 40 milésimos será aplicado nos métodos test1 e test2. Veja como resultado a Figura 15. A regra Timeout é criada marcando o atributo do tipo TestRule com a anotação @Rule:

```
@Rule
public TestRule tr= Timeout.millis(40);
```

- A regra TestName torna o nome do método de teste disponível dentro do método de teste. A regra TestName é criada marcando o atributo do tipo TestName com a anotação @Rule:

```
@Rule
public TestName tn = new TestName();
```

O método test3 da Figura 14 faz uso da regra TestName.

```
package aula;
import static org.junit.Assert.*;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TestName;
import org.junit.rules.TestRule;
import org.junit.rules.Timeout;

public class RuleTimeoutTest {
   @Rule
   public TestRule tr = Timeout.millis(40);
   @Rule
   public TestName tn = new TestName();

   @Test
   public void test1() {
      while( true );
   }

   @Test
   public void test2() {
      while( !false );
   }

   @Test
   public void test3() {
      assertEquals( "test3", tn.getMethodName());
   }
}
```

Figura 14 – Classe para testar as regras Timeout e TestName.

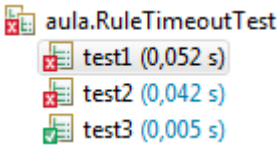

Figura 15 – Resultado do teste da Figura 14.

- A regra TemporaryFolder permite criar arquivos e pastas que são deletadas quando o teste é terminado. A regra TemporaryFolder é criada marcando o atributo do tipo TemporaryFolder com a anotação @Rule:

```
@Rule
public TemporaryFolder folder = new TemporaryFolder();
```

O método test1 da Figura 17 usa a classe TemporaryFolder para criar um arquivo que é enviado para a classe Arquivo da Figura 16. Já o método test2 usa a classe TemporaryFolder para criar uma pasta.

- A regra ExpectedException permite criar a expectativa de uma exceção no teste. A regra ExpectedException é criada marcando o atributo do tipo ExpectedException com a anotação @Rule:

  ```
  @Rule
  public ExpectedException thrown = ExpectedException.none();
  ```

  Os métodos test3 e test4 da Figura 17 acrescentam a expectativa da exceção no método.

```
package aula;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Arquivo {

   public static boolean escrever(File arquivo,
                String linha) throws IOException{
      FileWriter fw = new FileWriter(arquivo,true);
      BufferedWriter bw = new BufferedWriter(fw);
      bw.write(linha);
      bw.close();
      return true;
   }

   public static boolean imprimir(File arquivo)
                            throws IOException {
         /* abre o arquivo para leitura */
         FileReader reader = new FileReader(arquivo);
         BufferedReader b =
                         new BufferedReader(reader);
         String linha = null;
         do{
            linha = b.readLine();
            if( linha != null ){
               System.out.println( linha );
            }
         }while( linha != null );

         b.close();
         reader.close();
      return true;
   }
}
```

Figura 16 – Código da classe Arquivo.

```
package aula;
import static org.junit.Assert.*;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TemporaryFolder;

public class ArquivoTest {
   @Rule
   public TemporaryFolder folder =
                                  new TemporaryFolder();
   /* o método none retorna uma Rule que não espera
      exceções, semelhantemente ao comportamento do
      teste sem esta regra */
    @Rule
    public ExpectedException thrown =
                             ExpectedException.none();
   @Test
   public void test1() throws IOException {
     /* cria um arquivo temporário de nome teste.txt */
      File arquivo = folder.newFile("teste.txt");
      /* imprime o local onde o arquivo foi criado */
      System.out.println( folder.getRoot() );
      /* escrever no arquivo*/
      Arquivo.escrever(arquivo, "Bom dia");
      Arquivo.escrever(arquivo, "Boa tarde");
      /* ler o arquivo */
      Arquivo.imprimir(arquivo);
      /* verifica se o arquivo existe */
      assertTrue( arquivo.exists() );
   }

   @Test
   public void test2() throws IOException {
     /* cria uma pasta temporária de nome teste */
      File pasta = folder.newFolder("teste");
      System.out.println( folder.getRoot() );
      assertTrue( pasta.exists() );
   }   @Test
   public void test3() {
     /* acrescenta à lista de requisitos a exceção
      a ser esperada */
      thrown.expect(IOException.class);
      /* fornece uma mensagem para a exceção */
      thrown.expectMessage("Exceção no test3");
```

```
    }

    @Test
    public void test4(){
        thrown.expect(NullPointerException.class);
        thrown.expectMessage("Exceção no test4");
    }
}
```

Figura 17 – Código da classe ArquivoTest para testar as regras TemporaryFolder e ExpectedException.