FTCE

Faculdade de Tecnologia e Ciências Exatas

USJT - 2016 - ARQDESIS - Arquitetura e Desenvolvimento de Sistemas

Professores: Bonato, Élcio e Rodrigo

Aula: 3
Assunto: Testes Unitários em Java com JUnit (Revisão e Aplicação)

Conceitos básicos abordados pelo professor:

Os testes unitários, inegavelmente, aumentam a qualidade do software desenvolvido pois reduzem enormemente a quantidade de bugs (BECK, 2000, p.118). E resultados melhores são obtidos quando as baterias de testes são repetidas a cada alteração significativa do código, os chamados testes de regressão. Porém, retestar tudo, sempre, é uma tarefa cara e tediosa. Por este motivo, utilizam-se ferramentas de automação de testes. Uma das ferramentas de uso consagrado é o JUnit (BECK; GAMMA, 2004).

Teste unitário - definição

Um teste unitário é um código escrito por um programador com o objetivo de testar uma funcionalidade específica do código a ser testado. Seu alvo é a menor unidade de código: um método em uma classe. São testes denominados **testes de caixa preta**, pois verificam apenas a saída gerada por um certo número de parâmetros de entrada, não se preocupando com o que ocorre dentro do método.

O quê testar?

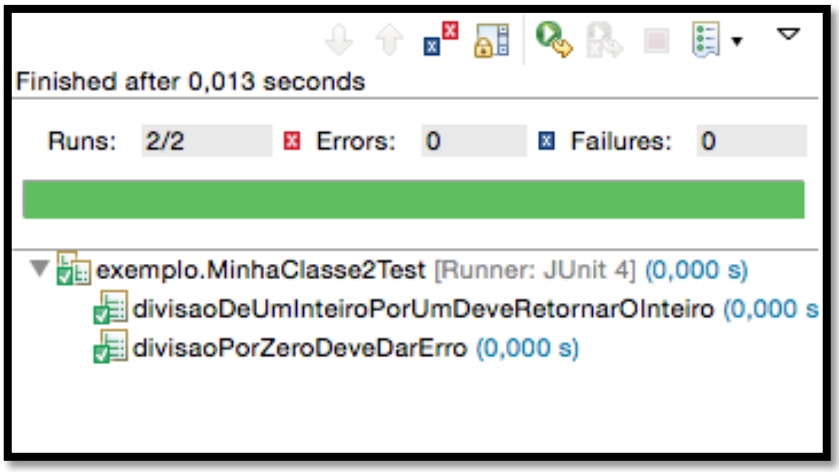
Este assunto sempre gera controvérsias, mas um código trivial, como métodos get e set, não precisa ser testado. Estes testes geram trabalho e pouco resultado. Por outro lado, código mais complexo, como métodos que contenham regras de negócio, estes sim devem ser alvos de diversos testes. Um conjunto sólido de testes protege o sistema de erros de regressão (basicamente, estragar sem querer código que não foi mexido) quando acontecem alterações no sistema.

O JUnit

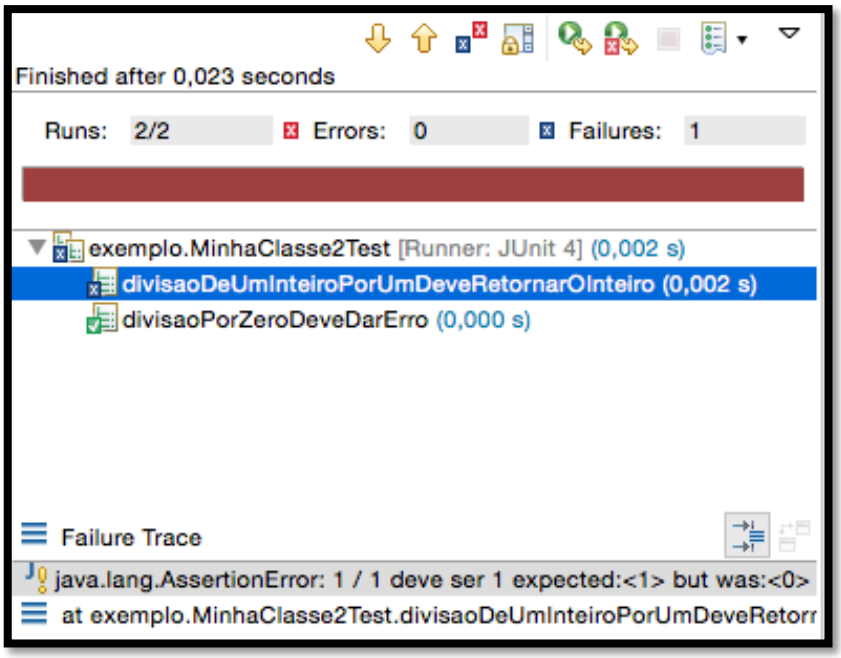
O JUnit é um framework open source de testes criado por Kent Beck (Extreme Programming) e Erich Gamma (Design Patterns), disponível em www.junit.org.
A versão atual do framework é a 4.x, que faz o uso extensivo de anotações (Java Annotations) para identificar os métodos de teste. O download pode ser feito de https://github.com/junit-team/junit.

Exemplos de código JUnit

Veja os exemplos 1 e 2. No exemplo 1 é descrita uma classe com um método que multiplica dois inteiros recebidos por parâmetro.
O exemplo 2 mostra a classe de teste deste método. Há um método de teste, indicado pela *annotation* @Test. E, dentro dele, 3 chamadas ao método *AssertEquals*, que tem um parâmetro String com a identificação do caso de teste, um parâmetro com a chamada do método a ser testado e um parâmetro com o resultado esperado. O *AssertEquals* compara o retorno do método multiplica com o resultado que o próprio programador indicou como esperado e retorna **true** se for igual e **false** se for diferente.
Ao se executar o teste, o JUnit mostra uma barra de progressão dos testes. Ela começa na cor verde e continua verde enquanto nenhum erro for encontrado. Se algum caso de teste der erro, a barra muda para vermelho continua assim até o fim dos testes.
No final, o JUnit mostra os retornos de todos os casos de teste, destacando os que apresentaram erro e mostrando o resultado esperado e o obtido.



Teste sem erros.



Teste com erros. Note no rodapé o detalhe do erro.

O exemplo 3 mostra uma classe com um único método que faz a divisão de dois inteiros. O caso de teste, no exemplo caso, possui 2 métodos: um que testa a divisão por 1, novamente usando o AssertEquals. E outro que testa a divisão por zero. Note que, neste caso, a *annotation* diz qual a exceção esperada. Como esta classe tem dois métodos de teste, foi usado o método setUp para instanciar a classe a ser testada.
O exemplo 5 mostra uma classe com um método de ordenação BubbleSort (já visto em ALGED). A classe de testes (exemplo 6) testa a ordenação de 3 número inteiros, testando todas as combinações de entrada possíveis. Note que, desta vez, o método usado é o AssertArrayEquals, que compara dois vetores. Se você usar o AssertEquals ele irá comparar a referência do vetor, e não seu conteúdo.

Conjuntos de Testes (*Test Suites*)

Os diversos casos de teste podem ser reunidos em um único conjunto de testes, de modo que todos seja executados, automaticamente, em sequência. Você pode ver isso no exemplo 7.

Anotações disponíveis no JUnit

Annotation	Descrição
@Test public void method	Identifica um método como um método de teste.
@Test (expected = Exception.class)	Falha se o método não lançar a exceção esperada
@Test(timeout=100)	Falha se o método levar mais de 100 milissegundos para retornar
@Before public void method()	É executado antes de cada teste. Usado para preparar o ambiente de testes (ler dados de entrada, instanciar classes, etc).
@After public void method()	É executado ao final de cada teste. Usado para limpar o ambiente de testes, como deletar dados temporários ou restaurar defaults.
@BeforeClass public static void method()	Executado uma única vez, antes de todos os testes. Usado para atividades demoradas, como fazer uma conexão com o banco de dados. Note que este método deve ser static.
@AfterClass public static void method()	Executado uma única vez, depois de todos os testes. Analogamente ao anterior, deve ser usado para atividades demoradas, como fechar a conexão com o banco de dados. Note que este método deve ser static.
@Ignore	Ignora um método de teste. Usado quando você não quer executar um determinado método de teste mas não quer apagá-lo ou comentá-lo.

Asserts disponíveis no JUnit

Annotation	Descrição
fail(String)	Faz com que um método de teste falhe. Todo método de teste falha antes de ser implementado. O parâmetro é opcional.
assertTrue([mensagem], boolean condição)	Verifica se a condição lógica é verdadeira.
assertFalse([mensagem], boolean condição)	Verifica se a condição lógica é falsa.
assertEquals([String mensagem], esperado, obtido)	Verifica se dois valores são iguais. Não compara conteúdo de vetores.
assertEquals([String mensagem], esperado, obtido, tolerance)	Verifica de dois double são iguais dentro da tolerância indicada, que é o número de casas decimais que devem ser iguais.
assertArrayEquals([String mensagem], esperado, obtido)	Compara o conteúdo de dois vetores, elemento por elemento.
assertNull([mensagem], objeto)	Verifica se o objeto é nulo.
assertNotNull([mensagem], objeto)	Verifica se o objeto não é nulo.
assertSame([String], esperado, obtido)	Verifica se ambas as variáveis se referem ao mesmo objeto.
assertNotSame([String], esperado, obtido)	Verifica se as variáveis se referem a diferentes objeto.

Nomenclatura dos Testes

Nenhum padrão é obrigatório. O padrão sugerido é que as classes tenham o mesmo nome da classe a ser testada com o sufixo Test e os métodos indiquem o que está sendo testado. Veja os exemplos 2, 4 e 6.

Ordem de Execução dos Testes

O JUnit executa os testes, em uma classe de testes, na ordem determinada por ele. Se em alguma situação for necessário executar os testes em uma ordem definida pelo desenvolvedor use a anotação @FixMethodOrder(MethodSorters.NAME_ASCENDING) antes da declaração da classe de testes.

Uso do JUnit Integrado ao Eclipse

O JUnit vem integrado ao pacote do Eclipse. Para criar uma classe de teste, selecione a classe que quer testar e clique em **Arquivo > Novo > Outro > Java > JUnit > JUnit Test Case** Selecione os métodos que quer criar (exceto os de teste) e clique em **Next**. Então selecione os métodos que quer testar e clique em **Finish**. O Eclipse gera a classe e os métodos.
Para gerar um conjunto de testes, clique em **Arquivo > Novo > Outro > Java > JUnit > JUnit > JUnit Test Suite.** O Eclipse irá pedir para você selecionar as classes que quer que façam parte do conjunto de testes.

Uso do JUnit integrado ao JGrasp

Você terá que baixar do GitHub (https://github.com/junit-team/junit/wiki/Download-and-Install) o junit.jar e o hamcrest-core.jar. Coloque os dois arquivos jar no mesmo diretório do seu computador.
No JGrasp, vá em **Tools > JUnit > Configure** e indique o diretório onde você colocou os arquivos. Será criado um ícone novo na barra de ferramentas do JGrasp para geração de classes de teste. Clique nele e a classe de teste será gerada. Note que, agora, surgiram ícones para compilar, rodar e depurar testes. Clique em compilar e depois em rodar para executar os casos de teste. Se houver problemas, coloque o hamcrest no classpath do JGrasp.
O JGrasp não gera conjuntos de testes automaticamente. Você terá que fazer isso manualmente.
Obs: Para integrar com o JUnit o JGrasp deve ser o 1.8.6 ou superior, com pelo menos Java 1.5.

Exemplo1: Classe a ser testada

```
package exemplo;

public class MinhaClasse1 {

   public int multiplica(int x, int y) {
      return x * y;
   }
}
```

Exemplo2: Classe de Teste do Exemplo 1

```
package exemplo;

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

public class MinhaClasse1Test {

   @Test
   public void multiplicacaoDeUmInteiroPorZeroDeveRetornarZero() {

      //Classe a ser testada
      MinhaClasse1 teste = new MinhaClasse1();

      //Testes
      assertEquals("10 X 0 deve ser 0", teste.multiplica(10, 0), 0);
      assertEquals("0 X 10 deve ser 0", teste.multiplica(0, 10), 0);
      assertEquals("0 X 0 deve ser 0", teste.multiplica(0, 0), 0);
   }

}
```

Exemplo3: Classe a ser testada

```
package exemplo;

public class MinhaClasse2 {

   public int divide(int x, int y){
      return x/y;
   }
}
```

Exemplo4: Classe de Teste do Exemplo 3

```
package exemplo;

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

public class MinhaClasse2Test {
   MinhaClasse2 teste;

   @Before
   public void setUp(){
      //Classe a ser testada
      teste = new MinhaClasse2();
   }

   @Test
   public void divisaoDeUmInteiroPorUmDeveRetornarOInteiro() {

      //Testes
      assertEquals("10 / 1 deve ser 10", teste.divide(10, 1), 10);
      assertEquals("0 / 1 deve ser 0", teste.divide(0, 1), 0);
      assertEquals("1 / 1 deve ser 1", teste.divide(1, 1), 1);
   }

   @Test(expected = ArithmeticException.class)
   public void divisaoPorZeroDeveDarErro() {

      //Teste de Exceção
      assertEquals("10 / 0 deve dar ArithmeticException", teste.divide(10,
0));
   }

}
```

Exemplo5: Classe a ser testada

```
package exemplo;

public class MinhaClasse3 {
   /**
    * @param v vetor de inteiros a ser ordenado
    * Ordena usando uma ordenacao por bolha
    * @return vetor v ordenado em ordem crecente
    */
   public int[] crescente(int[] v) {
      for (int i = v.length - 1; i > 0; i--) {
         for (int j = 0; j <= i - 1; j++) {
```

```
          if (v[j] > v[j + 1]) {
            int aux = v[j + 1];
            v[j + 1] = v[j];
            v[j] = aux;
          }
        }
      }
      return v;
   }
}
```

Exemplo6: Classe de Teste do Exemplo 5

```
package exemplo;

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

public class MinhaClasse3Test {

   @Test
   public void tresInteirosEmQualquerOrdemSaoOrdenadosEmOrdemCrescente() {
      //Classe a ser testada
      MinhaClasse3 teste = new MinhaClasse3();

      //Testes
      // testar as 6 permutacoes possiveis
      int[] res = { 1, 2, 3 };
      int[] v1 = { 1, 2, 3 };
      int[] v2 = { 1, 3, 2 };
      int[] v3 = { 2, 1, 3 };
      int[] v4 = { 2, 3, 1 };
      int[] v5 = { 3, 1, 2 };
      int[] v6 = { 3, 2, 1 };
      // testar as 6 permutacoes possiveis
      assertArrayEquals("Entrada 1, 2, 3", teste.crescente(v1), res);
      assertArrayEquals("Entrada 1, 3, 2", teste.crescente(v2), res);
      assertArrayEquals("Entrada 2, 1, 3", teste.crescente(v3), res);
      assertArrayEquals("Entrada 2, 3, 1", teste.crescente(v4), res);
      assertArrayEquals("Entrada 3, 1, 2", teste.crescente(v5), res);
      assertArrayEquals("Entrada 3, 2, 1", teste.crescente(v6), res);
   }

}
```

Exemplo7: Classe a ser Testada

```
package exemplo;

public class Aluno {
   private int ra;
   private String nome;

   public Aluno(int ra, String nome) {
      this.ra = ra;
      this.nome = nome;
   }
   public int getRa() {
      return ra;
```

```
  }
  public void setRA(int rA) {
    ra = rA;
  }
  public String getNome() {
    return nome;
  }
  public void setNome(String nome) {
    this.nome = nome;
  }
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Aluno other = (Aluno) obj;
    if (ra != other.ra)
      return false;
    return true;
  }
}
```

Exemplo8: Classe de Teste do Exemplo 7

```
package exemplo;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class AlunoTest {
  Aluno aluno;

  @Before
  public void setUp() throws Exception {
    aluno = new Aluno(12345, "Joao");
  }

  @Test
  public void testEquals(){
    assertEquals("Aluno 12345, Joao", aluno, new Aluno(12345, "Joao"));
  }

}
```

Exemplo9: Conjunto de Testes

```
package exemplo;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
```

```
@RunWith(Suite.class)
@SuiteClasses({ MinhaClasse1Test.class, MinhaClasse2Test.class,
     MinhaClasse3Test.class, AlunoTest.class })
public class AllTests {

}
```

Exercícios

Encontre no material do SOL este material de aula e o pacote de código aula03_codigo_teste.zip. Importe para o Eclipse.

Crie a tabela cliente do banco de dados usando o script cliente1.sql e execute a suite de tests AllTests. Entenda as classes de teste.

Drope a tabela cliente do banco e crie outra usando o script cliente2.sql - agora o campo chave primária id é autoincrement.

Refatore o código da classe ClienteDAO e da classe ClienteDAO test para lidar com esta alteração na tabela cliente.

Dica: alterar o método incluir do DAO para que retorne o id gerado pelo banco. Para isso adicione o fragmento de código abaixo após o execute().

A função LAST_INSERT_ID() do MySQL retorna o valor do último id inserido pela sua sessão.

```
String sqlSelect = "select LAST_INSERT_ID()";
try(PreparedStatement pst1 = conn.prepareStatement(sqlSelect);
     ResultSet rs = pst1.executeQuery();){
   if(rs.next()){
     idGerado = rs.getInt("id");
   }
} catch (SQLException e) {
   e.printStackTrace();
}
```

Exercícios para Entrega

Cada dupla deve:

1. reorganizar as packages do projeto Eclipse gerado durante a refatoração da aula passada conforme o código exemplo.

2. criar a suite de testes para o código gerado durante a refatoração da aula passada. Cada classe deve ter sua respectiva classe de teste.

3. a partir de agora, não será aceito mais código sem testes unitários no JUnit

Bibliografia

BECK, K. Extreme Programming explained: embrace change. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0-201-61641-6.

BECK, K.; GAMMA, E. JUnit Open Source Testing Framework. abr 2015. On-line. Disponível em: <http://www.junit.org/>.

KOSKELA, K; Effective Unit Testing: A guide for Java developers; 1a Ed. Manning Publications, 2015. ISBN 1935182579.

MESZAROS, G; XUnit Test Patterns: refactoring test code; 1ª Ed. Person Education, 2007. ISBN 0-13-149505-4.