**Universidade Federal de Campina Grande – UFCG**

**Centro de Engenharia Elétrica e Informática – CEEI**

**Departamento de Sistemas e Computação – DSC**

Disciplina: Laboratório de Programação 2

# **Usando Testes de Unidade**

O objetivo deste documento é introduzir o conceito de testes de unidade, mostrando como usá-los para ajudar na criação das suas classes.

Sobre testes

Como você identifica, hoje, se **sua classe está correta?** Se sua resposta foi "escrevo um main", está na hora de aprender sobre testes de unidade e sobre uma ferramenta que apoia o desenvolvimento de tais testes.

Toda classe que você escrever precisa vir acompanhada de uma outra classe, que é uma classe de teste (chamamos de test case). Chamamos esse tipo de teste de teste de unidade, onde a menor porção "testável" de uma aplicação, chamada unidade, é verificada em relação ao comportamento que se espera dela. Em se tratando de orientação a objetos, é natural que esta unidade seja uma classe.

Assim, iremos escrever testes de unidade para testar a classe e achar erros. Uma outra abordagem é escrever os testes primeiro e usar esses testes como parte da especificação das classes a serem implementadas.

Em java, existe uma API chamada [JUnit](http://junit.org) que vai facilitar o desenvolvimento de classes de teste, ao mesmo tempo que permite a execução do. Você vai ter que aprender a usar uma API JUnit e este lab vai lhe ajudar a entender um pouco melhor como usar essa API.

Preparando o ambiente

Vamos usar um exemplo simples no contexto do sp2fy (um sistema para organizar e armazenar músicas de um perfil de usuário)**.**

O primeiro passo é definir onde no projeto vão ficar os nossos testes. Tipicamente, as classes de teste (chamado de código de teste) ficam em um diretório diferente dos usados para armazenar as Classes com o código da aplicação (também chamado de código de produção). Se, por exemplo, as classes da sua aplicação estão no pacote sp2fy, os seus testes podem estar em um diretório testes, seguindo a mesma estrutura do código de produção (testes.sp2fy).

**Crie um projeto** chamado labTestes e dentro dele **crie o pacote sp2fy** onde serão implementadas as classes da sua aplicação, como por exemplo, a classe **Musica.java**. Em seguida, crie um outro diretório fonte (src) para armazenar seu código de teste. Para isso clique com o botão direito no projeto e selecione **New > Source Folder**. Nomeie seu novo diretório fonte para “testes”. O ideal é que a mesma hierarquia de pacotes que existe no package **sp2fy** exista também no seu diretório de testes.

Agora você vai copiar a classe de testes fornecida **MusicaTest.java** para o pacote dos testes testes.sp2fy. Observando a janela do "Package Explorer", podemos ver que o pacote da aplicação está vazio e o pacote dos testes contém a classe **MusicaTest.java**.

Configurando o JUnit

Vá "build path" do projeto criado (clicando com o botão direito do mouse sobre o pacote, escolha a opção **"Build path>Add Libraries**" e escolha a opção JUnit clicando no item **JUnit**. Trabalhe sempre com a versão estável mais recente do JUnit, que no caso é a 4. Finalize esta configuração.

Você já pode rodar o teste que você escreveu clicando com o botão direito do mouse sobre a classe e selecionando "**Run as > JUnit Test**". Como o seu código ainda está cheio de erros, o teste vai falhar e isso é representado por uma barra vermelha no término da execução do teste. O próximo passo é implementar a classe referenciada no teste, incluindo construtores, métodos e atributos.

Usando os testes para implementar as classes sendo testadas

Para alcançar a tão almejada “barra verde” vamos começar restringindo os problemas. Então, comente o método de teste testEquals() e vamos começar a construir a classe Musica para passar nos testes do método testMusica().

Veja que o método testMusica() faz referências ao construtor da classe Musica e a outros métodos, no caso, métodos acessadores, que devem ser implementados na classe.

|  |
| --- |
| @Test  public void testMusica() {  Musica chandelier = new Musica("Chandelier", 3, "Pop");  ...  assertEquals("Chandelier", chandelier.getTitulo());  assertEquals(3, chandelier.getDuracao());  assertEquals("Pop", chandelier.getGenero());  ... |

A classe Musica.java deve ter um construtor com 3 parâmetros do tipo dos argumentos passado na criação do objeto chandelier. Da mesma forma, deve ter métodos acessadores para seus atributos que serão título, duração, e gênero. Você deve implementar o construtor da classe, definir os atributos da classe e os seus respectivos métodos acessadores e, assim, os erros de compilação irão ser resolvidos. Em seguida, você deve executar novamente a classe de testes MusicaTest.java usando o JUnit e, possivelmente, os testes serão bem sucedidos agora.

Agora você pode descomentar o método testEquals() e analisar que método está sendo testado e qual o comportamento desse método. No caso, são testes para o método equals da classe Musica. Observe as assertivas sobre o objeto chandelier e perceba que dois objetos serão iguais de acordo com os valores dos atributos título e duração da música. Implemente o equals para obedecer a essas condições e execute os testes novamente. Barra verde?

|  |
| --- |
| @Test  public void testEquals() {  Musica chandelier = new Musica("Chandelier", 3, "Pop");  Musica elasticHeart = new Musica("Elastic Heart", 3, "Pop");  Musica cellophane = new Musica("Cellophane", 4, "Pop");  assertEquals(chandelier, new Musica("Chandelier", 3, "Pop"));  assertEquals(chandelier, new Musica("Chandelier", 3, "Pop-rock"));  assertNotEquals(elasticHeart, chandelier);  assertNotEquals(elasticHeart, cellophane);  } |

Entendendo um pouco mais a classe de teste

Cada método de teste na classe de teste começa com uma anotação **@Test**. Essa anotação diz a JVM que cada método da classe de teste deve testar um aspecto "pequeno" da classe sob teste. Por exemplo, deve haver um método de teste para testar cada método da classe Musica.java separadamente. Cada método de teste deve ser pequeno e específico.

Os métodos de teste JUnit se utilizam de asserções ("**assertions**"), que são declarações que checam se uma condição é verdadeira ou falsa. Se a condição é falsa, o teste falha. Quando todas as asserções feitas em um método de teste são verdadeiras, vai aparecer uma barra verde ao final da execução do caso de teste. “Passar” e “Falhar” são veredictos de um caso de teste.

JUnit oferece muitos métodos de assertion. Veja a API da classe Assert [aqui](http://junit.org/javadoc/latest/org/junit/Assert.html).

O objeto a ser usado no teste

Em todo caso de teste que exercita uma classe, um ou mais objetos da classe sob teste precisam ser criados. É com base nesse(s) objetos que as asserções são avaliadas. Esses objetos podem ser criados em cada método de teste como ilustrado abaixo:

|  |
| --- |
| public class MusicaTest {  @Test  public void testMusica() {  Musica chandelier = new Musica("Chandelier", 3, "Pop");  Musica elasticHeart = new Musica("Elastic Heart", 3, "Pop");  Musica cellophane = new Musica("Cellophane", 4, "Pop");  …  }  ... |

É ainda mais comum ter um método que cria esses objetos como a seguir:

|  |
| --- |
| public class MusicaTest {  private Musica chandelier;  private Musica elasticHeart;  private Musica cellophane;    @Before  public void criaMusica(){  chandelier = new Musica("Chandelier", 3, "Pop");  elasticHeart = new Musica("Elastic Heart", 3, "Pop");  cellophane = new Musica("Cellophane", 4, "Pop");  }  ...  }  } |

Perceba que a classe de testes tem vários métodos que começam com a anotação @Test. E agora você observou um outro método com a **anotação @Before**. Essas anotações são mensagens passadas para a JVM que vai saber que métodos chamar. Antes de executar qualquer método que inicia com @Test é preciso executar os métodos que iniciam com @Before. O que vem no método associado a @Before é a preparação do ambiente de teste. Mesmo que um método de teste faça um set e modifique um atributo do objeto usado nos testes, no próximo teste um novo objeto estará disponivel, pois o método associado a @Before vai ser executado sempre antes da execução de qualquer método de teste. Você pode ter outros métodos que não tem anotações, e eles serão chamados normalmente como qualquer outro método.

No exemplo acima estamos dizendo que antes de executar cada método de teste (@Test) devemos criar 3 objetos do tipo Musica que chamamos de chandelier, elasticHeart e cellophane.

O teste do construtor

Que testes podemos realizar no construtor? Esses testes irão ajudar a verificar se o construtor da classe Musica está correto.

Um possível teste é verificar se o objeto chandelier realmente tem o título “Chandelier”, a duração 3 e o gênero “Pop”. Como isso pode ser feito? Através de uma asserção.

|  |
| --- |
| [**assertEquals**](http://junit.org/javadoc/latest/org/junit/Assert.html#assertEquals(java.lang.String,%20java.lang.Object,%20java.lang.Object))([String](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html) message, [Object](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html) expected, [Object](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html) actual) |

Veja um exemplo no método chamado testMusica.

|  |
| --- |
| @Test  public void testMusica() {  Musica chandelier = new Musica("Chandelier", 3, "Pop");  assertEquals("Chandelier", chandelier.getTitulo());  assertEquals(3, chandelier.getDuracao());  assertEquals("Pop", chandelier.getGenero());  ...  } |

Os testes do equals

A forma de testar o equals vai sempre requisitar que vários objetos sejam criados até que todas as possibilidades sejam testadas. Vejamos um exemplo de um possível método que testa o equals:

|  |
| --- |
| @Test  public void testEquals() {  Musica chandelier = new Musica("Chandelier", 3, "Pop");  Musica elasticHeart = new Musica("Elastic Heart", 3, "Pop");  Musica cellophane = new Musica("Cellophane", 4, "Pop");  assertEquals(chandelier, new Musica("Chandelier", 3, "Pop"));  assertEquals(chandelier, new Musica("Chandelier", 3, "Pop-rock"));  assertNotEquals(elasticHeart, chandelier);  assertNotEquals(elasticHeart, cellophane);  } |

Normalmente o teste do equals deve verificar se um objeto é igual a ele mesmo, se é igual a outro objeto com os mesmos valores dos atributos que determinam igualdade, ou se é diferente caso contrário.

Testes com exceções

Suponho que alguém queira criar um objeto Musica da seguinte forma:

Musica chandelier = new Musica(null, 3, “Pop”);

Devemos aceitar **nomes que sejam nulos**? Nesse caso não faz sentido! Então é preciso testar essa condição.

O que queremos que aconteça quando uma musica com nome "null" é criada? Aqui chegamos em uma novidade. Queremos lançar uma **exceção**. Uma exceção é lançada toda vez que um parâmetro de entrada é inadequado para o contexto do programa, como é o caso do null. **Como podemos testar se nosso código está tratando bem os erros?**

1. Forçar a situação de erro (por exemplo, passando o valor inválido);
2. Esperar que o programa saiba se recuperar do erro (por meio da Exception).

Portanto, uma exceção é testada da seguinte forma: você coloca dentro de um **try{ … }** o código que deve lançar exceção. Ainda dentro do try você faz uma chamada a asserção [fail](http://junit.org/javadoc/latest/org/junit/Assert.html#fail(java.lang.String)) (que sempre vai fazer o teste falhar se for executado). Veja o código a seguir:

|  |
| --- |
| @Test  public void testMusicaInvalida() {  try {  Musica tituloInvalido = new Musica(null, 3, "Pop");  fail(); **// se chegar aqui da erro, pois deveria lancar exception.**  } catch (Exception e) {  String msgEsperada = "Titulo da musica nao pode ser nulo ou vazio.";  assertEquals(msgEsperada, e.getMessage());  }  …  } |

Dentro do **catch** você pode testar se a mensagem de erro associada à exceção lançada foi a esperada.

Observe ainda que os testes para condições normais de criação de objetos Musica devem ser modificados para refletir o uso de exceções. Isso significa que o teste deve verificar se, de fato, não vai ser lançada exceção para tais condições.

|  |
| --- |
| @Test  public void testMusica() {  try {  Musica chandelier = new Musica("Chandelier", 3, "Pop");  assertEquals("Chandelier", chandelier.getTitulo());  assertEquals(3, chandelier.getDuracao());  assertEquals("Pop", chandelier.getGenero());  } catch (Exception e) {  fail();// nao deveria ter lancado nenhuma Exception nesse teste.  }  } |

Técnicas avançadas e boas práticas de teste de unidade

Da mesma forma que um bom código, um bom teste deve ser modularizado, reusável e bem legível. Além disso, utilizamos o conceito de cobertura, para indicar que um teste exercita (ou seja, ele executa) o máximo do código possível. Como criamos bons testes?

O teste de unidade é feito para testar as unidades de código, ou seja, as menores partes funcionais do seu código. No caso de programação OO, essas unidades são as classes. Há dezenas de tipos de testes (sistemas, aceitação, regressão, stress, segurança, integração, componentes, etc.), cada um com sua vantagem e custo. Um dos conceitos principais na área de teste é o **caso de teste**, que é o conjunto de passos/comandos que são executados para observar se seu sistema se comporta como esperado. Note que nós devemos saber qual é o comportamento esperado do sistema, então precisamos ter um **valor esperado** para comparar as saídas do código.

No JUnit, cada método com uma anotação @Test é visto como um caso de teste. Um bom caso de teste deve ser **atômico e independente**, ou seja, não deve depender de outros casos de testes. Porém eles podem depender de um **setup** do sistema (também podemos chamar isso de **pré-condição**). No JUnit, indicamos o método de setup por meio de uma anotação @Before.

Como boa pratica, cada classe do seu código de produção deve ter uma classe de teste correspondente. Essa classe de teste possui um método para cada caso de teste, ou seja, para cada situação que você deseja setar. A nomeclatura seria: *testGetTitulo()*; *testGetDuracao()*; *testGetGenero()*. Note que todos eles tem uma pré-condição, que é a criação de um objeto Música. Ou seja, criamos o método setup(){ …. } que cria uma Musica. Veja abaixo:

|  |
| --- |
| public class MusicaTest {  private Musica musica;  private String tituloMusica = "Formation";  private int duracaoEsperada = 5;  private String generoEsperado = "Pop";    @Before  public void setup() throws Exception{  this.musica = new Musica(tituloMusica,5,"Pop");  }    @Test  public void testGetTitulo(){  Assert.assertEquals(tituloMusica, musica.getTitulo());  }  @Test  public void testGetGenero(){  Assert.assertEquals(generoEsperado, musica.getGenero());  }    @Test  public void testGetDuracao(){  Assert.assertEquals(duracaoEsperada, musica.getDuracao());  }  } |

Também há técnicas específicas para o bom teste envolvendo Exception. A forma correta é forçar que o teste lance o Exception, porém verificar o lançamento da Exception por meio do expected parametrizado no @Test. Apresentamos o exemplo da música abaixo:

|  |
| --- |
| **@Test(expected = Exception.class)**  public void testTituloVazio() throws Exception{  //Aqui sera lancada uma Exception  musica = new Musica("",5,"Pop");  }    @Test(expected = Exception.class)  public void testTituloNull() throws Exception{ **//note o throws**  musica = new Musica(null,5,"Pop");  }    @Test(expected = Exception.class)  public void testGeneroVazio() throws Exception{  musica = new Musica("Fly me to the moon.",5,"");  }    @Test(expected = Exception.class)  public void testGeneroNull() throws Exception{  musica = new Musica("Fly me to the moon.",5,null);  }    @Test(expected = Exception.class)  public void testDuracaoInvalida() throws Exception{  musica = new Musica("Fly me to the moon.",-5,"Forro");  }  @Test(expected = Exception.class)  public void testDuracaoZero() throws Exception{  musica = new Musica("Fly me to the moon.",0,"Forro");  } |

Note que o teste dessa forma fica independente de assert de mensagens de Exception. Além disso, ao entender a Hierarquia de Excpetions e os diferentes tipos de Expcetion, podemos fazer casos de teste específicos para cada tipo de Exception no seu projeto.

Para ver a classe MusicaTest completa, clique [aqui](https://goo.gl/SAOZr4). Mas lembre-se que o teste, assim como o código, exige experiência do desenvolvedor. Então, fique atenta(o) e procure aplicar os conceitos e boas práticas de testes sempre que escrever código de produção. Enxergar as situações que irão quebrar o seu programa faz com que você se torne, cada vez mais, um excelente profissional da engenharia de software.