**Universidade Federal de Campina Grande – UFCG**

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

**Departamento de Sistemas e Computação – DSC**

Disciplina: Laboratório de Programação 2

# **Laboratório 05**

O objetivo deste lab é introduzir o conceito de testes, os diferentes níveis de testes, e focar na criação de testes de unidade e iniciar a criação de tais testes usando JUnit.

Testes e Qualidade de Software

Uma das características principais de um software é sua **qualidade**. Desenvolver código de qualidade não significa, necessariamente, desenvolver um produto de qualidade. Qualidade é uma característica que deve estar presente em todas as etapas de criação de um software. Essas etapas são definidas pelos processos de desenvolvimento de software. Esses processos especificam as diferentes formas nas quais podemos criar um software. Existem diversas variações desses processos, mas a base de todos é que um software deve ter as etapas de **Concepção**, **Projeto/Design**, **Desenvolvimento**, **Verificação e Validação** e então **Manutenção**.

A etapa responsável por **avaliar a qualidade** do produto é a Verificação e Validação, também conhecida como V&V. Apesar de parecidas, essas duas palavras significam coisas bem distintas:

* **Validação:** Observar se o software atende aos requisitos e **necessidades do cliente**;
* **Verificação:** Observar se o software está de acordo com a **especificação**;

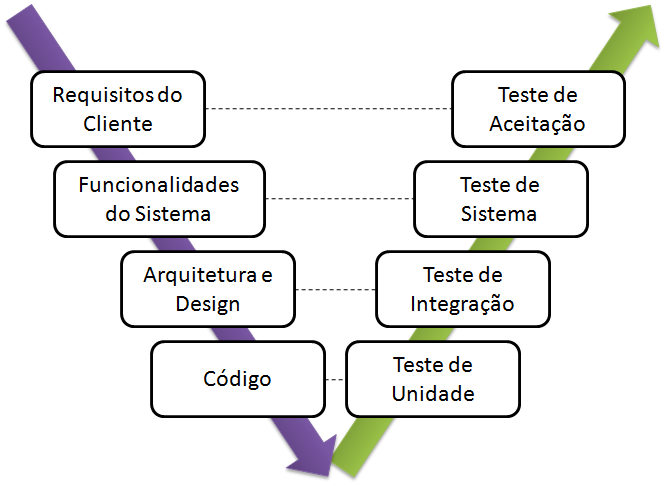
Uma forma simples de lembrar a diferença é pensar nas seguintes perguntas:

|  |
| --- |
| * **Validação :** Estamos construindo o **software certo**? * **Verificação:** Estamos **construindo certo** o software? |

A verificação está relacionada com o comportamento do código. Ou seja, se o **código está correto** para o que o time de desenvolvimento decidiu e decidiu nas etapas de Concepção, Projeto/Design e Desenvolvimento. No entanto, o objetivo final é **satisfazer o cliente** para o qual estamos produzindo o software, seja ele uma pessoa, empresa, a sociedade, uma instituição, etc.

Podemos ter um software funcionando corretamente do ponto de vista da especificação, porém ao apresenta-lo ao cliente, o software pode não estar satisfatório, seja porque entendemos errado um requisito do cliente, ou não conseguimos completar as expectativas do cliente. Nesse caso, conseguimos verificar o software, mas ele não foi validado. Em outro casos, o software pode atingir às expectativas do cliente, porém ele pode estar bem seboso e longe da especificação (cheio de gambiarras). Nesse caso, temos um software validado, mas com uma verificação pobre. Portanto, vamos **sempre garantir a verificação e validação dos nossos softwares** para manter a qualidade tanto do que produzimos quanto daquilo que entregamos aos nossos clientes.

Uma das atividades de V&V é o **teste de software** (existem outras como inspeções, análise estática, métodos formais, etc.) Dentre outras razões, os testes são populares pois nos permite **observar o comportamento** do software, e nos ajudam a entender o software por meio de sua **execução**. Ainda assim, temos várias formas de testar. O modelo mais básico de teste é o **modelo-V** apresentado abaixo:



O modelo-V (note o formato em V) passa pelos diversos níveis de **abstração do software**. Desde o requisito alto nível até o baixo nível do código (para o cliente código é baixaria ;) ). Cada um desses níveis podem apresentar defeitos, portanto há estratégias de testes específicas para cada **nível de teste**.

Em P2 vamos exercitar o teste de unidade e o teste de aceitação para que vocês possam entender algumas das formas de garantir qualidade do software.

|  |
| --- |
| Podemos dizer que **testes de unidade é validação?**  E o **teste de aceitação** está relacionada à **verificação**? |

Como testadores e desenvolvedores, usamos **ferramentas** para criar, executar e gerenciar automaticamente os nossos testes. Isso deixa o processo de testes mais rápido e eficaz. Uma dessas ferramentas é o JUnit, voltada para o teste de unidade. Note que o JUnit não é adequado para os outros níveis de testes (exemplo, o de sistemas). Daí, podemos usar outras ferramentas como o **Selenium**, **EasyAccept**, **Cucumber**, **TestLink**, dentre outros.

Desenvolver a habilidade de testar bem, ou ser um bom testador envolve conhecer, além das ferramentas, os principais aspectos de um software. Ou seja, quais os **erros mais comuns** que os desenvolvedores (noobs e veteranos) cometem? Quais as formas mais adequadas de **integrar pedaços do sistema**? Que **tipos de problemas** o sistema pode apresentar na entrega?

Um excelente testador geralmente é um excelente desenvolvedor, mas o contrário nem sempre é verdadeiro! Portanto, procurem entender como testar o seu programa para encontrar todos os seus defeitos… é melhor ele “quebrar” na sua frente, do que na frente do seu cliente. Qualidade, geralmente, é a característica mais valiosa de um software, portanto, garanta a qualidade do seu software.

|  |
| --- |
| A área de testes é bem ampla com sua terminologia e princípios específicos. Por exemplo, você sabia que existem dezenas de tipos de testes? Você já ouviu falar em Testes de: Interface, Stress, Fumaça, Sanidade, baseado em Modelos, Mutação, Regressão, Segurança, Componentes, dirigido a Modelos, Usabilidade, Carga, Confiabilidade, Caixa-branca, Alfa, Beta, Caixa-preta, etc.  Cada um com suas vantagens e desvantagens. Procure material adicional, pergunte aos seus professores no curso, entenda BEM como melhorar a qualidade do seu software, e consequentemente, das suas atividades como profissional de computação. |

Sobre testes de unidade

Os testes de unidade tem se tornado “bons amigos” dos alunos de lp2. Até então, vimos como usar esse tipo de teste como parte da especificação das classes implementadas. Nesse caso, usamos testes escritos por terceiros como guia para a implementação das nossas classes de modo que ao obter sucesso na execução dos testes, as funcionalidades testadas estariam implementadas corretamente nas respectivas classes.

Agora queremos escrever os testes de unidade para testar cada classe implementada e achar seus erros. Este lab vai lhe ajudar a alcançar tal objetivo.

É importante salientar que 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. Em java, existe uma API chamada [JUnit](http://junit.org) que vai facilitar o desenvolvimento de classes de teste. Você deve consultar o seguinte [material de apoio](https://goo.gl/pFfVUk) para lembrar os principais conceitos envolvidos em testes de unidade e questões de preparação do ambiente e configuração do JUnit.

Antes de criar sua própria classe de testes, vamos entender um pouco sobre o que testar.

O que testar?

Essa é uma questão sutil e você só vai ter segurança sobre essa questão depois que tiver experiências escrevendo testes de unidade e também experiências vendo bons testes de unidade serem escritos. Você pode em casa, por exemplo, ver um exemplo da criação de testes de unidade no seu material de P2.

Em resumo, o teste é a atividade que fornece informação concreta se seu software funciona como deveria. A principal regra para saber o que testar é **ter criatividade para imaginar as possibilidades de testes**.

* Comece pelas classes mais simples, que tenham menos dependências de outras classes (não tem outras classes que você escreveu com atributos, por exemplo ou que sejam passadas como parâmetros de entrada para métodos) e deixe os testes complexos para o final;
* Use apenas dados suficientes (não teste 10 situações se três forem suficientes). **Evite** testar repetidamente a mesma lógica com dados parecidos. Use valores bem distintos (por exemplo, explore **os limites da variável**: maximo/mínimo inteiro possível, ou Strings com valores não alfa-numéricos).
* Não teste métodos triviais, tipo get e set, exceto se eles não forem triviais (isso é possível em algumas situações). 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)!

Essas frases podem estar meio sem sentido agora. Mas depois que começarmos voce pode voltar e reler o que está aqui.

Escrevendo o primeiro teste

Clicando com o botão direito sobre o pacote de testes, escolha "**New>JUnit Test Case**". Você vai ter que oferecer informação para que o esqueleto da sua classe de teste seja criado com pouco esforço. A sua classe sob teste (class under test) é [academia.programas.Exercicio](https://drive.google.com/file/d/0B8n9er4GBTlxMG45TmtiMThSeHM/view?usp=sharing) (essa classe faz parte da solução para uma academia OO, como na [seguinte especificação](https://goo.gl/RZnF2D)). A boa prática de programação sugere que o nome de sua classe de teste seja o mesmo nome da classe sendo testada seguido do nome Test: ExercicioTest. Clique em "Next" para continuar a configuração de seu esqueleto de teste. Você vai agora definir que métodos da classe Exercicio você quer testar. Você quer testar todos, exceto getNome, setNome e getCph, porque são muito triviais e foram gerados pelo eclipse. Clique em "Finish".

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**". O esqueleto da classe de teste criada automaticamente vai sempre falhar. Falhas são representadas por uma barra vermelha no término da execução do teste. O próximo passo é implementar os testes para testar cada método da classe.

Criando 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. É comum ter um método que cria esses objetos como a seguir:

|  |
| --- |
| **public class** ExercicioTest {  **private** Exercicio exc1;    @Before  **public void** criaExercicio() {  exc1 = **new** Exercicio("abdominal", 200);  }  ... |

Observe 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 um exercício que chamamos exc1.

**Adicione o código acima à sua classe de teste ExercicioTest.**

Testando o construtor

Que testes podemos realizar no construtor? Será que o construtor da classe Exercicio está correto?

Um possível teste é verificar se o objeto exc1 realmente tem o nome "abdominal" e cph é realmente 200. Como isso pode ser feito? Através de duas asserções.

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

Procure o método chamado testExercicio. Substitua esse método pelo seguinte método:

|  |
| --- |
| @Test  public void testExercicio() {  assertEquals("Nome do exercicio esta errado.",  "abdominal",exc1.getNome());  assertEquals("Cph do exercicio esta errada.", 200, exc1.getCph());  } |

Entenda essas asserções. Comente os demais métodos de teste (os que começam com @Test). Apenas o testExercicio deve estar descomentado e com o código novo acima. Deu barra verde?

Agora responda: você acha que existe alguma situação que deveria ser testada e que não está sendo testada? Por exemplo, se se alguém criar um objeto Exercicio da seguinte forma:

Exercicio exc = new Exercicio(null, 200);

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

O que queremos que aconteça quando um exercicio com nome "null" é criado? 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.

Entendendo como testar exceções

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:

|  |
| --- |
| Exercicio exc2;  try {  exc2 = new Exercicio(null, 200);  fail("Deveria ter lancado excecao de nome");  } catch ( Exception exp ) {  assertEquals("Mensagem de excecao capturada.","O nome do exercicio não pode ser null", exp.getMessage());  } |

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

Modifique a classe Exercicio para que esse teste dê a tão esperada barra verde. Em seguida, adicione novas asserções semelhantes às asserções acima para não permitir que cph seja 0 ou qualquer valor menor que zero. Modifique novamente a classe Exercicio para que esse teste dê a tão esperada barra verde (passar no teste).

Testando outros métodos

Para testar os métodos de uma classe você precisa saber exatamente como eles devem se comportar para diferentes "casos". Cph, por exemplo, é uma variável que nunca deve assumir um valor que seja zero ou menor que zero. Então é preciso testar esses valores e garantir que o comportamento desejado (lançar exceção) irá ocorrer. Usando como base o que vimos na seção anterior, escreva o método de teste testSetCph. Lembre de testar o método para um valor correto de cph, para um valor limite (nesse caso zero) e para um valor notadamente incorreto de cph, por exemplo, -5.

Crie também teste para o toString.

Testando 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 testEqualsObject() throws Exception {  assertEquals(exc1, exc1);  Exercicio exc2 = new Exercicio("abdominal", 200);  assertEquals(exc1, exc2);  Exercicio exc3 = new Exercicio("corrida", 700);  assertNotEquals(exc1, exc3);  Exercicio exc4 = new Exercicio("abdominal", 201);  assertEquals(exc1, exc4);  Exercicio exc5 = new Exercicio("natacao", 700);  assertNotEquals(exc3, exc5);  } |

Leia esse código e **entenda**. Você entende por que todos esses testes foram feitos? Por que um não foi suficiente?

Testando um método comportamental

O método testCalPerHoras é um método comportamental que deve ser testado de forma completa. Veja um exemplo de como testar esse método:

|  |
| --- |
| @Test  **public void** testCalPerHoras() throws Exception {  // testando comportamento em situacao correta  assertEquals("Valor de cal por horas incorreto.", 600,  exc1.calPerHoras(3));  // Existe um valor limite? O que acontece se passar 0 minutos?  // Isso deve ser especificado  assertEquals("Valor de cal por minutos incorreto.", 0,  exc1.calPerHoras(0));  // e se passar minutos negativos?  *try* {  exc1.calPerHoras(-30);  fail("Deveria lançar exceção de horas negativos");  } **catch** (Exception exp) {  assertEquals("Mensagem de excecao no metodo calPerHoras",  "O tempo passado nao pode ser negativo",  exp.getMessage());  }  } |

**Importante: Note que olhando para os seus testes você tem uma noção muito completa de como objetos da classe sob teste devem se comportar. Esta é mais uma vantagem de se ter testes de unidade. Além de ter mais segurança de que o código está correto, temos uma documentação clara de como os objetos da classe devem se comportar em situações variadas.**

Verifique se sua classe de testes está completa e rode os testes. No fim você deve ter **barra verde** :).