



# WTEIA

WORKSHOP DE TECNOLOGIA, EMPREENDEDORISMO  
E INOVAÇÃO DO AGRESTE PERNAMBUCANO





# Testes unitários com JUnit

Francisco do Nascimento (IFPE)  
[francisco.junior@jaboatao.ifpe.edu.br](mailto:francisco.junior@jaboatao.ifpe.edu.br)

# Agenda

---

- Introdução a testes de software
- Níveis de testes
- JUnit
- Prática
- TDD
- Ferramentas complementares: Eclemma, Mutation

# Cenário atual

---

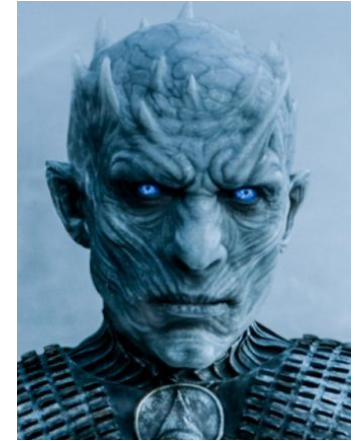
- Software desempenha diversos papéis no nosso dia-a-dia (social, econômico, acadêmico)
- Processos reais sendo mapeados para sistemas automatizados
- Aumenta a pressão sobre a qualidade do software
  - Softwares de baixa qualidade devem ser rejeitados
  - Falhas podem causar catástrofes

***Você entraria num avião se soubesse que o trem de pouso não passou em 100% dos testes?***

# 3 lições sobre testes de software

---

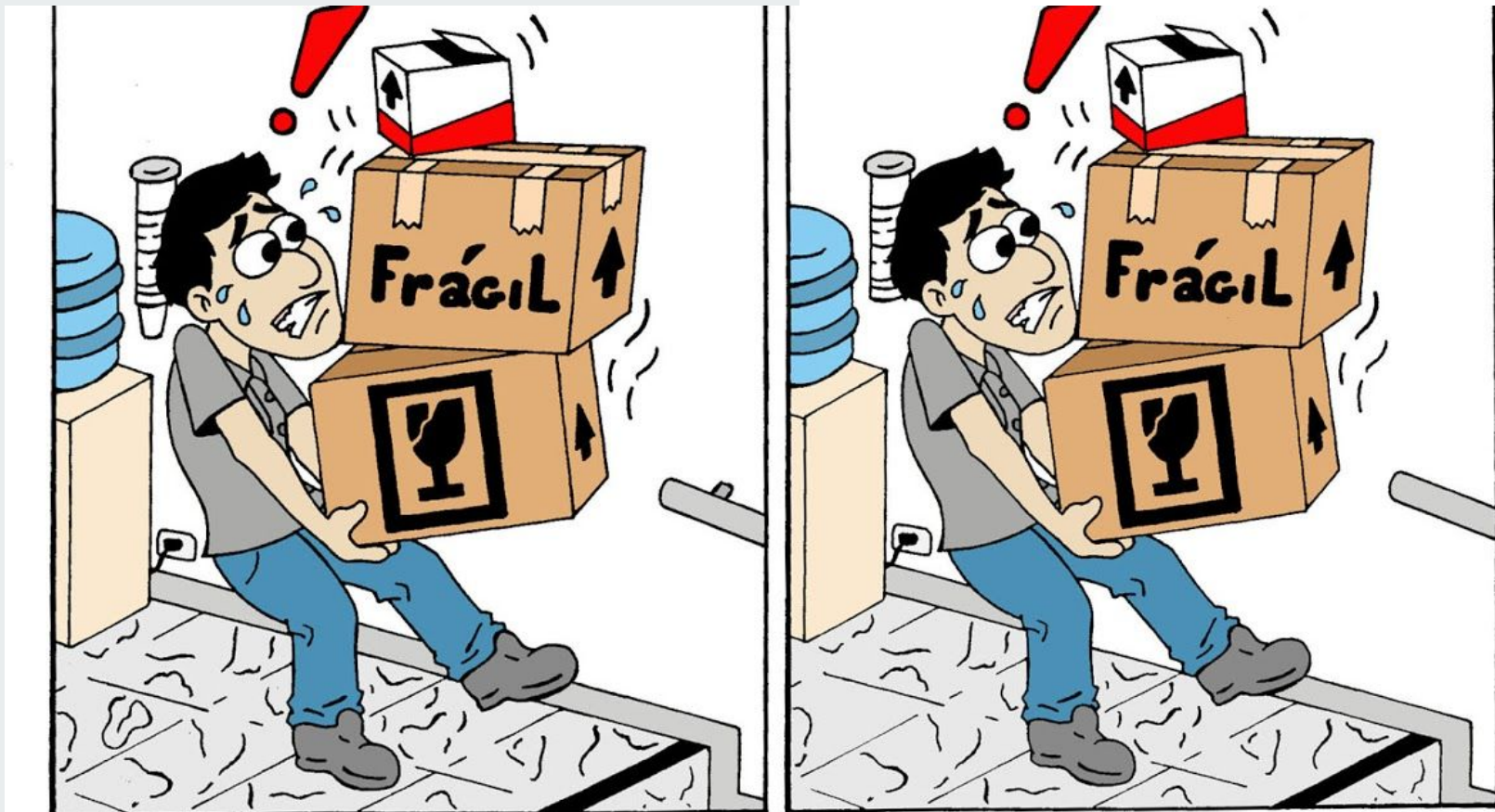
# 1ª Lição: Testar é uma ação destrutiva





## 2ª Lição:

### Teste bem-sucedido encontra erros



3ª Lição:  
Quanto mais cedo,  
menor o custo!





# Então, o que é testar?

- ( ) Demonstrar que os erros não estão presentes.
- ( ) Mostrar que o programa executa as funções no tempo esperado.
- ( ) Estabelecer a certeza de que o programa faz o que se propôs a fazer.
- ( ) Executar um programa na intenção de encontrar erros.

# Resumindo...

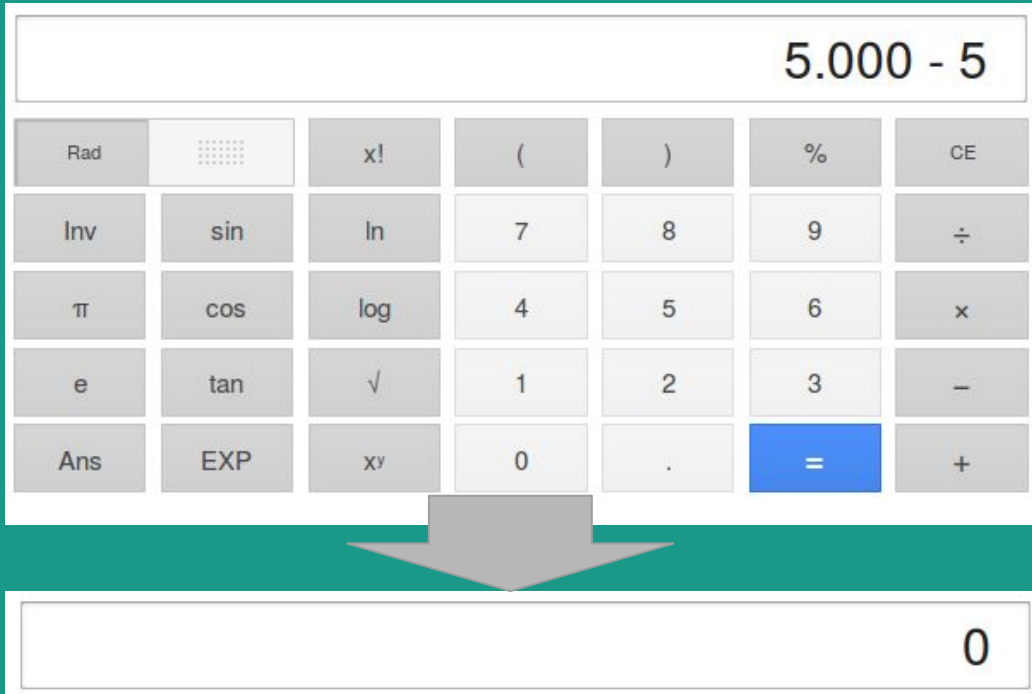
---

Visão de Testes  
como um processo

O principal objetivo de testar um software é encontrar, de forma **sistemática**, o maior número possível de **defeitos** embutidos durante a construção do software

O que pode ser  
considerado um  
defeito?

# Exemplo: Calculadora

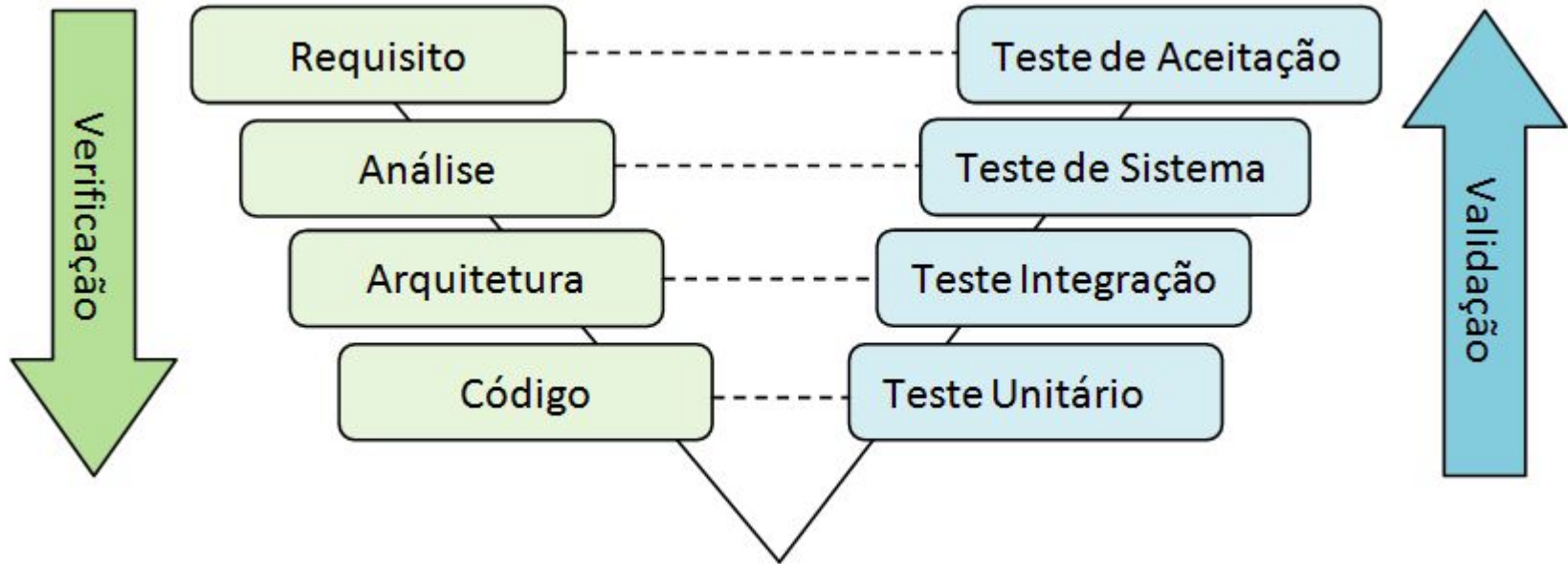


- O resultado não deveria ser 4995?
- É um defeito? Por que?  
**BUG vs FEATURE**
- Precisamos verificar como foi especificado o software

# Testes unitários

---

# Conceito V de Testes





# Testes de unidade (ou unitários)

---

- **Objetivo:** Explorar a menor unidade do projeto (classes, métodos, componentes, páginas web, servlets, etc.)
- **Foco:** Exame minucioso do código e da interface disponibilizada para a unidade
- **Quando:** Após o desenvolvimento (ou antes -- TDD)
- **Onde:** Usando a própria IDE
- **Quem:** Geralmente realizado pelo próprio desenvolvedor
- **Ferramentas:** JUnit (Java), DUnitX (Delphi), NUnit (.Net)

# JUnit



- Framework para facilitar o desenvolvimento e a execução de testes de unidade em programas Java
- Facilidade em fazer as asserções
  - Asserção: Uma afirmação - se aquilo não for verdade, o teste deve indicar uma falha.
- JUnit 4: Surgiram as annotations
  - @Before, @After, @Test, @SuiteClasses
- JUnit 5: Novas annotations
  - @ParametrizedTest, @RepeatedTest, @TestFactory
- Apresenta diagnóstico: sucesso/falha e detalhes

Links: <https://junit.org/junit5/>, <http://junit.org/junit4/>

# Exemplo 1: Verificar aprovação

**Objetivo:** Verificar se um aluno está aprovado ou não.

**Especificação:** O aluno estará aprovado se atender simultaneamente aos dois critérios:

- (1) Média aritmética entre duas notas maior ou igual a 7.0
- (2) Não faltar mais que 25% das aulas.

**Entrada:**

- Nota 1 e Nota 2 (valores de 0 a 10)
- Frequência: 0 a 1 (valor percentual)

**Saída:** Verdadeiro ou Falso

# Exemplo 1: Especificação de cenários e casos de testes

**Objetivo:** Verificar se um aluno está aprovado.

**Especificação:** O aluno estará aprovado se atender simultaneamente aos dois critérios:

(1) Média aritmética entre duas notas maior ou igual a 7.0

(2) Não faltar mais que 25% das aulas.

**Entrada:**

- Nota 1 e Nota 2 (valores de 0 a 10)

- Frequência: 0 a 1 (valor percentual)

**Saída:** Verdadeiro ou Falso

CN	Descrição
CN01	Entradas válidas para média maior que 7.0 e frequência maior que 75%
CN02	Entradas válidas para média igual a 7.0 e frequência igual a 75%
CN03	Entradas válidas para média menor que 7.0 e frequência maior que 75%

CT	CN	Nota1	Nota2	Frequência	Saída esperada
CT01	CN01	7	8	0.88	Verdadeiro
CT02	CN01	10	10	1	Verdadeiro
CT03	CN01	9.9	4.2	0.75001	Verdadeiro
CT04	CN02	7	7	0.75	Verdadeiro
CT05	CN02	10	4	0.75	Verdadeiro
CT06	CN03	4	8	0.80	Falso

# Exemplo 1: Implementação dos testes unitários

```
public class Desempenho {  
  
    public boolean verificarAprovacao(float nota1,  
        float nota2, float frequencia){  
        float media = (nota1 + nota2)/3;  
        boolean resultado = false;  
  
        if (frequencia < 0.75){  
            resultado = false;  
        } else {  
            if (media >= 7.0){  
                resultado = true;  
            }  
        }  
  
        return resultado;  
    }  
}
```

1. Identificar o método que é objeto de teste:

**nome:** verificarAprovacao  
**entrada:** float, float, float  
**saída:** boolean

2. Criar uma classe de Teste

3. Implementar os casos de testes

Cenário: CN01

Caso de Teste: CT01

Nota1	Nota2	Frequência
7	8	0.88

Saída esperada: Verdadeiro



# Exemplo 1: Classe de testes unitários (JUnit)

```
public class DesempenhoTest { 1
    2 private Desempenho desempenho;

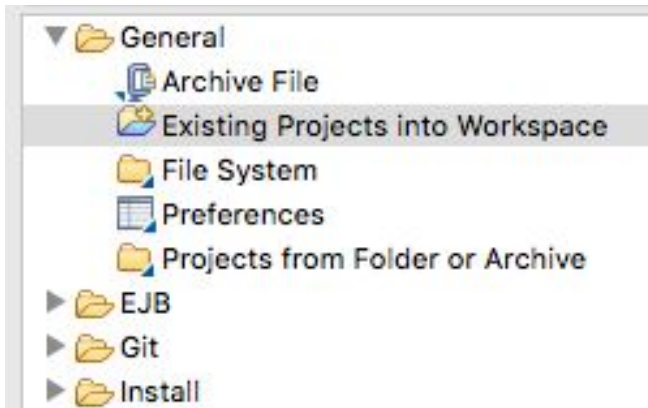
    @Before
    3 public void init(){
        desempenho = new Desempenho();
    }

    @Test
    4 public void testVerificarAprovacao1() {
        float nota1 = 8;
        float nota2 = 7;
        float freq = 0.88f;
        boolean esperado = true;
        boolean obtido = desempenho
    5         .verificarAprovacao(nota1, nota2, freq);
    6 Assert.assertEquals("TC001", esperado, obtido);
    }
```

1. Nova classe: **DesempenhoTest**
2. Objeto em teste: **desempenho**
3. Método de inicialização: **init() -- @Before**
4. Método de teste: **testVerificarAprovacao1 -- @Teste**
5. Chamada do método a ser testado
6. Asserção do teste: **assert**

# Prática 1: Configurar o ambiente (1/2)

1. Baixar o arquivo [src-wteia-ts.zip](https://bit.ly/wteia-ts) do projeto Java ([bit.ly/wteia-ts](https://bit.ly/wteia-ts))
2. Abrir o Eclipse
3. Importar o projeto:
  - a. File > Import > General > Existing project into Workspace
  - b. Select archive file :: selecionar o arquivo [src-wteia-ts.zip](https://bit.ly/wteia-ts)



# Prática 1: Configurar o ambiente (2/2)

4. Adicionar biblioteca do JUnit
  - a. Botão direito sobre o projeto
  - b. Build Path > Add Libraries > JUnit



# Prática 1: Criar casos de teste JUnit

1. Criar pacote 'testes'
2. Criar classe DesempenhoTest do tipo **JUnit Test Case** no pacote **testes**
3. Implementar os casos de testes

CT	CN	Nota1	Nota2	Frequência	Saída esperada
CT01	CN01	7	8	0.88	Verdadeiro
CT02	CN01	10	10	1	Verdadeiro
CT05	CN02	10	4	0.75	Verdadeiro
CT06	CN03	4	8	0.80	Falso

# Prática 1: Executar casos de testes

1. Para executar uma classe JUnit Test Case:



2. Aba de Resultados

Finished after 0,018 seconds

Runs: 1/1    ✖ Errors: 0    ✖ Failures: 1

testes.DesempenhoTest [Runner: JUnit 4] (0,000 s)

testVerificarAprovacao1 (0,000 s)

Failure Trace

java.lang.AssertionError: TC001 expected: <true> but was: <false>

at testes.DesempenhoTest.testVerificarAprovacao1(DesempenhoTest.java:25)



## Anotações: @Before → @Test → @After



@Before: Método a ser executado antes dos testes

@Test: Um método só será executado pelo JUnit se tiver esta anotação

@After: Método a ser executado após os testes

# Assertões

- Uma assertção é uma verificação realizada pelo JUnit para decidir se o teste foi com sucesso ou falhou.
- Métodos estáticos importados: `import static org.junit.Assert.*;`
  - Parâmetros: mensagem, expected, actual
    - mensagem: identificação do caso de teste
    - expected: valor que esperamos
    - actual: valor real obtido da execução
- Se o valor real for diferente do esperado, o teste vai falhar e será exibida a mensagem:  
**expected <valor esperado> but was <valor real>**

# Assertões: alguns tipos

---

- `assertEquals(String msg, Object expected, Object actual)`
- `assertTrue(String msg, boolean value)`, `assertFalse`
- `assertNull(String msg, Object obj)`
- `assertEquals(String msg, long expected, long actual)`
- `assertEquals(String msg, double expected, double actual, double delta)`
  - Double é inexato → delta = margem de erro
- `assertArrayEquals(String msg, Object[] expected, Object[] actual)`
- `assertArrayEquals(String msg, int[] expected, int[] actual)`

# Verificação de Exceções

```
public boolean verificarAprovacao(float nota1, float nota2,  
    float frequencia) throws ValorInvalidoException {  
    boolean resultado = false;  
    if (nota1 < 0 || nota1 > 10) {  
        throw new ValorInvalidoException(nota1);  
    }  
    if (nota2 < 0 || nota2 > 10) {  
        throw new ValorInvalidoException(nota2);  
    }  
    if (frequencia < 0 || frequencia > 1) {  
        throw new ValorInvalidoException(frequencia);  
    }  
    float media = (nota1 + nota2) / 2;  
    if (frequencia < 0.75){  
        resultado = false;  
    } else {  
        if (media >= 7.0){  
            resultado = true;  
        }  
    }  
    return resultado;  
}
```

# Adicionar um teste de exceções

```
@Test(expected=ValorInvalidoException.class)
public void testValoresInvalidos() throws ValorInvalidoException {
    float nota1 = -1;
    float nota2 = 8;
    float freq = 0.75f;

    desempenho.verificarAprovacao(nota1, nota2, freq);
}
```



## Prática 2: Contagem de Moedas



Considere a implementação da história do usuário abaixo disponibilizada no `src-wteia-ts.zip` (pacote `wteia.atividade2`) e elabore 6 casos de testes usando JUnit.

US032 - Calcular valor de moedas
Como usuário do sistema, eu gostaria de colocar moedas de reais referentes a qualquer valor para visualizar o total do montante submetido.
<ul style="list-style-type: none"><li>• Deverão ser consideradas apenas as moedas de reais de valor 1, 5, 10, 25 e 50 centavos, e de 1 real.</li><li>• O número de moedas submetidas de cada tipo não pode ultrapassar 250.</li><li>• O número total de moedas submetidas não pode extrapolar 1000.</li></ul>

---

# **Eclemma**

<http://www.eclemma.org/>

# EclEmma

- O EclEmma é um plugin para o Eclipse que mede e apresenta a cobertura de testes.
- A execução dos testes marca trechos do código, indicando a cobertura
- Não-invasivos: EclEmma não requer modificar seus projetos ou realizar qualquer outra configuração.
- Instalação direta pelo Eclipse: através do Eclipse Marketplace (menu Help)
- Após reiniciar o Eclipse, surgirá um novo botão
- Funciona exatamente como os modos Run e Debug

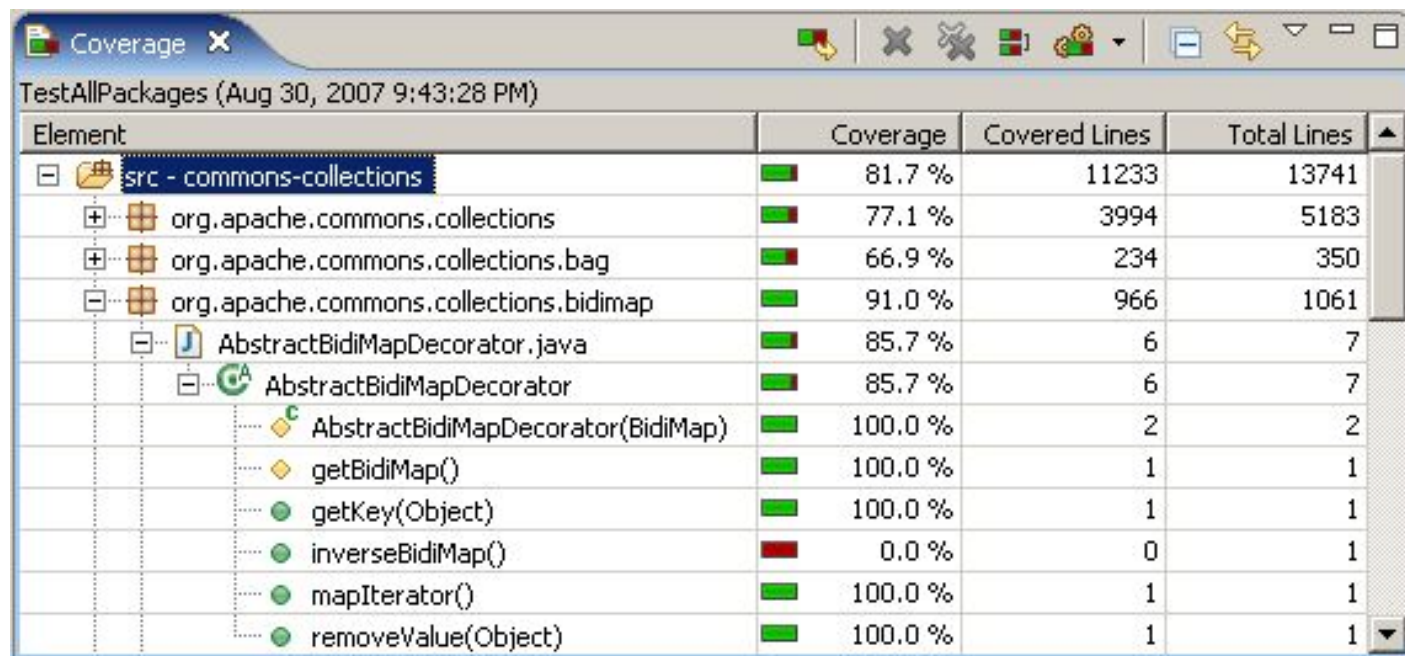


# Marcação de cobertura













```
public class Desempenho {  
  
    public boolean verificarAprovacao(float nota1,  
        float nota2, float frequencia){  
        float media = (nota1 + nota2)/3;  
        boolean resultado = false;  
  
        if (frequencia < 0.75){  
            resultado = false;  
        } else {  
            if (media >= 7.0){  
                resultado = true;  
            }  
        }  
  
        return resultado;  
    }  
}
```

# Resumo de cobertura

Coverage view: resumos de cobertura dos projetos que permite drill-down até método



The screenshot shows the 'Coverage' window for a test run named 'TestAllPackages' (Aug 30, 2007 9:43:28 PM). The table displays coverage data for various elements, including packages and methods. Each row includes a tree icon, a file icon, the element name, a progress bar, the coverage percentage, the number of covered lines, and the total number of lines.

Element	Coverage	Covered Lines	Total Lines
src - commons-collections	 81.7 %	11233	13741
org.apache.commons.collections	 77.1 %	3994	5183
org.apache.commons.collections.bag	 66.9 %	234	350
org.apache.commons.collections.bidimap	 91.0 %	966	1061
AbstractBidiMapDecorator.java	 85.7 %	6	7
AbstractBidiMapDecorator	 85.7 %	6	7
AbstractBidiMapDecorator(BidiMap)	 100.0 %	2	2
getBidiMap()	 100.0 %	1	1
getKey(Object)	 100.0 %	1	1
inverseBidiMap()	 0.0 %	0	1
mapIterator()	 100.0 %	1	1
removeValue(Object)	 100.0 %	1	1

# Prática 3: Analisar a cobertura dos testes

1. Instalar o Eclemma (Eclipse Marketplace)
2. Executar os testes usando Coverage
3. Analisar a cobertura e criar novos casos de testes para aumentar a cobertura.



---

# TDD - Test driven development



# Motivação

- Antes de começar a codificar, já ter em mente o que precisa ser desenvolvido e consequentemente situações válidas e inválidas para seu comportamento.
- Cientes disso, por que não implementar o teste logo?
- Mas não temos um programa ainda, como fazer o teste?
- Mudança de forma de pensar!!





## TDD - Definição

- TDD (Test Driven Development - Desenvolvimento guiado por testes) é uma técnica de programação ágil.
- Com TDD, especificamos nosso software criando testes executáveis e rodando-os de maneira que eles mesmos testem nosso software.
- Serve para diagnosticar precocemente “bugs” que poderiam vir a ser problemas na finalização do projeto.

# Ciclo de Desenvolvimento TDD

---



# Ciclo de Desenvolvimento TDD



1. Para codificar um software, vamos começar escrevendo um teste unitário que falha
2. Desenvolver/ajustar método de forma mais simples possível para que o teste unitário não falhe.
3. Remodelar o código.

Em suma, TDD é uma técnica de programação onde todo o código produzido é criado em resposta a um código que falhou.

# Exemplo



Problema: Criar um método que lê 3 inteiros e retorne o maior dos 3 valores.

Exemplo:

- Entrada: 3, 5, 1 ---> Saída: 5

# Exemplo

1º passo: Criar um teste vazio que falha

```
@Test
public void testarObterMaior1() {
    fail("Método não implementado");
}
```

Enquanto um método não for implementado, o teste continua falhando.



# Exemplo

2º passo: Fazer o teste funcionar

- Criar o método:

```
public static int obterMaior(int v1, int v2, int v3){  
    return 0;  
}
```

3º Passo: criar o teste para o método **obterMaior**:

```
@Test  
public void testarObterMaior1(){  
    int resultado = Util.obterMaior(5, 3 , 1);  
    assertEquals(5, resultado);  
}
```

4º Passo: Executar o teste



# Exemplo

Se o teste falhou, teremos que ajustar o método até que o teste obtenha sucesso.

5º passo: Fazer o teste funcionar

```
public static int obterMaior(int v1, int v2, int v3){  
    return 5;  
}  
  
@Test  
public void testarObterMaior1(){  
    int resultado = Util.obterMaior(5, 3 , 1);  
    assertEquals(5, resultado);  
}
```

Com esta alteração, o teste testarObterMaior1 obtém sucesso.  
Paramos por aí?



# Exemplo

Não! Vamos criar outro teste que falhe!

```
@Test
public void testarObterMaior2() {
    int resultado = Util.obterMaior(-5, 20, 1);
    assertEquals(20, resultado);
}
```

Executar todos os testes:

- testarObterMaior1 : Sucesso
- testarObterMaior2 : Falha

Qual o próximo passo?





# Exemplo

Teremos que ajustar o método para que o teste obtenha sucesso.

```
public static int obterMaior(int v1, int v2, int v3){  
    int resultado = v1;  
    if (v1 < v2) { resultado = v2; }  
    return resultado;  
}
```

Executar todos os testes:

- testarObterMaior1: Sucesso
- testarObterMaior2: Sucesso

E pronto? Não! A ideia é fazer novos testes que falhem e torná-los bem sucedidos até que não haja mais novas situações.



## Prática 4: Implementar usando TDD



Uma fábrica de software implantou uma regra para realizar o pagamento de bônus de final de ano aos seus funcionários. Um funcionário contém nome, e-mail, salário-base, cargo e o gerente. De acordo com seu cargo, o valor do bônus será diferente:

- **DESENVOLVEDOR:** 20% do salário-base, caso o salário-base seja menor que R\$ 3.000,00; ou R\$ 600,00, caso contrário.
- **DBA:** 15% do salário-base, caso o salário-base seja até R\$ 2.000,00; ou 5%, caso contrário.
- **TESTADOR:** valor fixo de R\$ 320,00.
- **GERENTE:** 2% da soma dos bônus dos seus subordinados.

# Referências



Material: [bit.ly/wteia-ts](https://bit.ly/wteia-ts)

<https://www.caelum.com.br/apostila-java-testes-jsf-web-services-design-patterns/testes-automatizados/#3-5-junit>

JUnit 5 x JUnit 4 : <https://howtoprogram.xyz/2016/08/10/junit-5-vs-junit-4/>

**Eclemma:** <https://www.eclemma.org/>  
<https://imasters.com.br/linguagens/java/cobertura-de-testes-unitarios-para-java-com-eclemma/?trace=1519021197&source=single>

Hamcrest:

<http://blog.caelum.com.br/melhorando-a-legibilidade-dos-seus-testes-com-o-hamcrest/>