



Universidade Federal do Ceará
Centro de Ciências/Departamento de Computação
Código da Disciplina: CK0236
Professor: Ismayle de Sousa Santos

Aula 06

Técnica de Programação II

Testes Unitários



qpg4p5x



ismaylesantos@great.ufc.br



@IsmayleSantos

Agenda

- **Testes Unitários**
 - O que é?
 - Ferramentas
 - JUnit
 - Comandos
 - Boas Práticas
 - Critérios de Cobertura
 - Reports
-

O que é teste?

- Um teste de software consiste na **verificação dinâmica** do comportamento de um programa através de um **conjunto finito** de casos de teste, adequadamente selecionado a partir de um número **geralmente infinito** de execuções deste programa

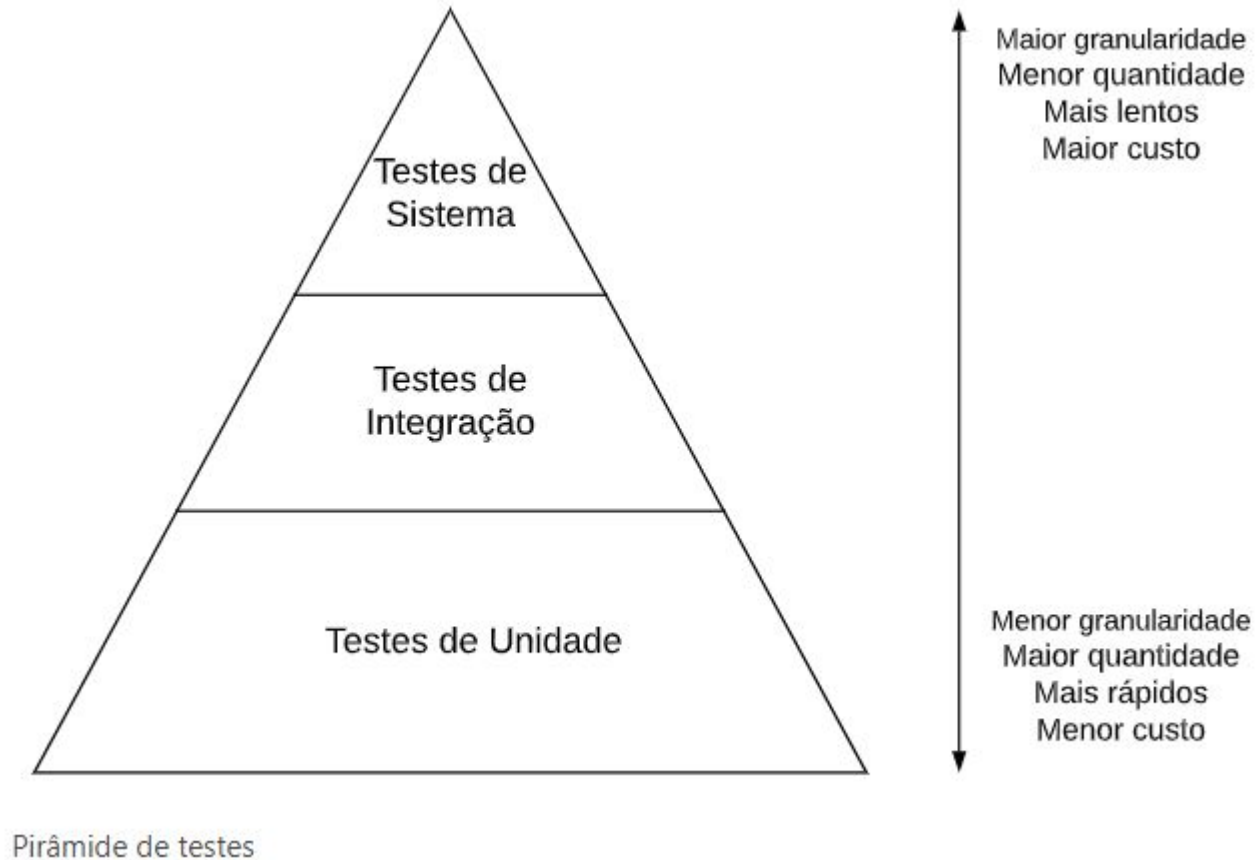


Erro x Defeito x Falha

- **Erro**
 - Ação humana que produz um resultado incorreto.
- **Defeito**
 - Quando o sistema não está de acordo com a especificação
- **Falha**
 - Quando o defeito é executado e leva o programa para um comportamento incorreto

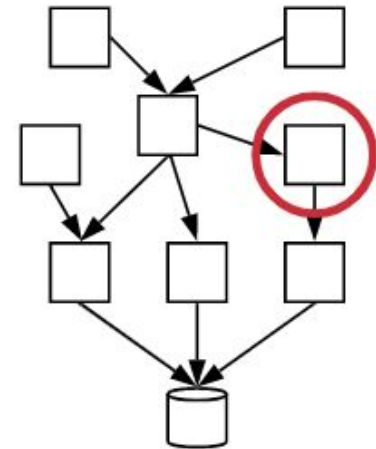


Níveis de Testes



Testes Unitários

- **Focam em**
 - verificar unidades testáveis do sistema
 - testar unidades individuais de forma independente
- Na POO, o foco é testar os métodos das classes
- **Benefícios**
 - Identificação precoce de defeitos
 - São simples
 - Fáceis de implementar
 - Executam rapidamente
 - Testes implementados na mesma linguagem do sistema



Escopo de testes de unidade

Testes Unitários

- É do tipo **caixa branca**
 - Baseado no código-fonte
- É de responsabilidade do desenvolvedor



Testes Unitários

- **Teste de uma Classe OO requer**
 - **Teste de todas as operações associadas com um objeto**
 - **Atribuir e obter valores a todos os atributos de objeto**
 - **Exercício do objeto em todos os estados possíveis**
 - **Simular todos os eventos que podem causar mudanças de estado**

Testes unitários chamam métodos de uma classe e verificam se eles retornam os resultados esperados

Ferramentas xUnit

JUnit



Ferramentas xUnit



- JMockit
 - Cobertura de código:
Cobertura de Linhas,
Cobertura de Caminhos,
Cobertura de dados
- EMMA
 - Mede cobertura de código Java
 - Suporta cobertura de classe, linha e métodos

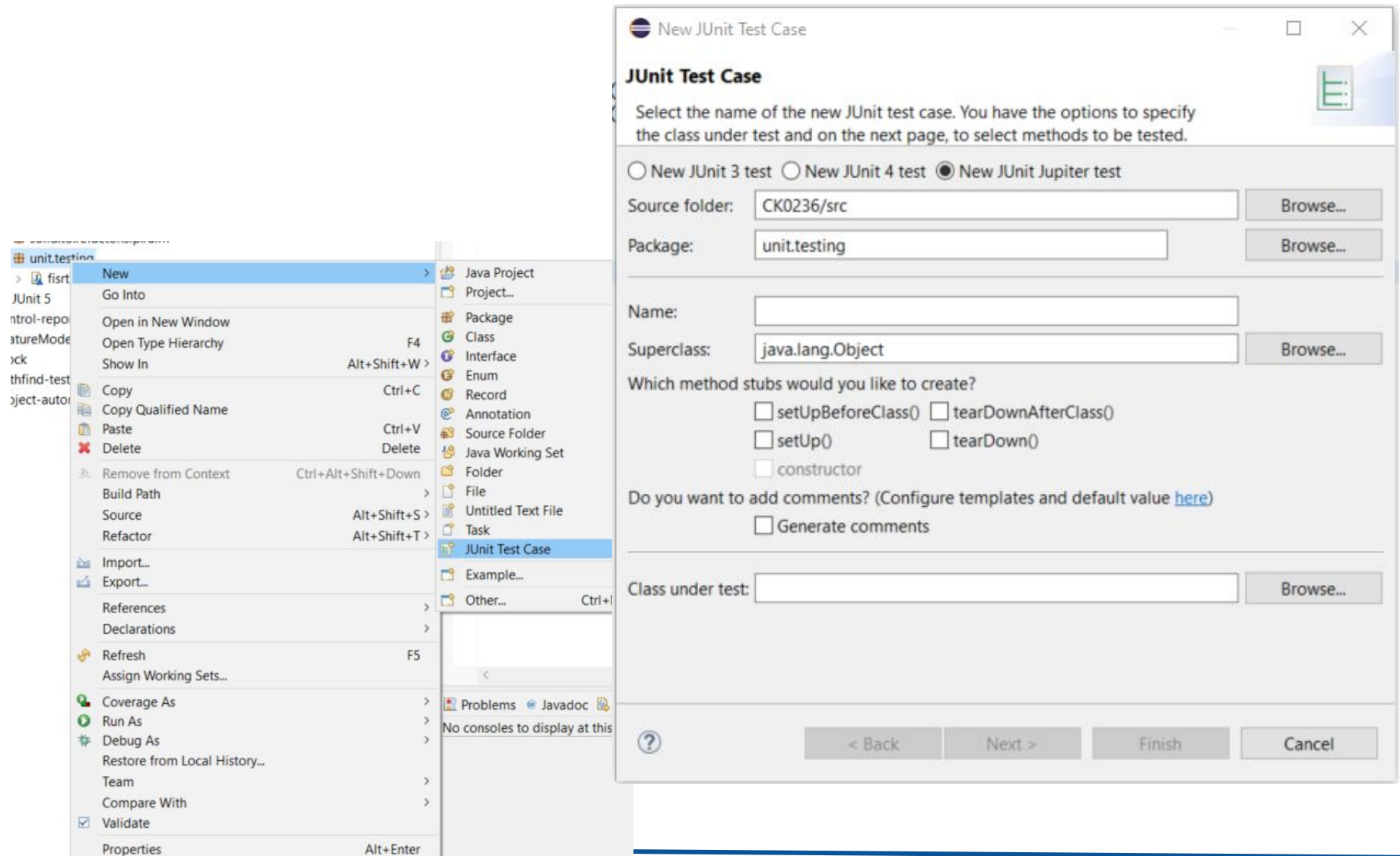


JUnit

- Framework de Testes Unitários em Java
- O Eclipse já possui o JUnit instalado
- Baseado em anotações



Criando um Caso de Teste com JUnit



Criando um Caso de Teste com JUnit

```
class testBotão {  
    @BeforeAll  
    static void setUpBeforeClass() throws Exception {  
    }  
  
    @AfterAll  
    static void tearDownAfterClass() throws Exception {  
    }  
  
    @BeforeEach  
    void setUp() throws Exception {  
    }  
  
    @AfterEach  
    void tearDown() throws Exception {  
    }  
  
    @Test  
    void test() {  
        fail("Not yet implemented");  
    }  
}
```

} Antes de todos os testes

} Depois de todos os testes

} Antes de cada teste

} Depois de cada test

Testes Unitários com JUnit

- Por convenção
 - Classes de teste têm o mesmo nome das classes testadas, mas com um sufixo **Test**
 - Métodos de testes começam com o prefixo **test**
 - São públicos e possuem a anotação *@Test*, a qual identifica métodos que deverão ser executados

```
import static org.junit.jupiter.api.Assertions.*;

class TestContaComum {

    @Test
    void testContaVazia() {
        ContaComum conta = new ContaComum();
        double saldo = conta.getSaldo();
        assertEquals(saldo, 0.0);
    }
}
```

Testes Unitários

- “Partes” de um teste unitário
 - **Configuração**
 - Iniciar o sistema com os dados de entrada
 - Instanciar o objeto que se pretende testar
 - **Chamada**
 - Chamar os métodos da classe a ser testado
 - **Afirmação**
 - Compara o resultado da chamado ao resultado esperado

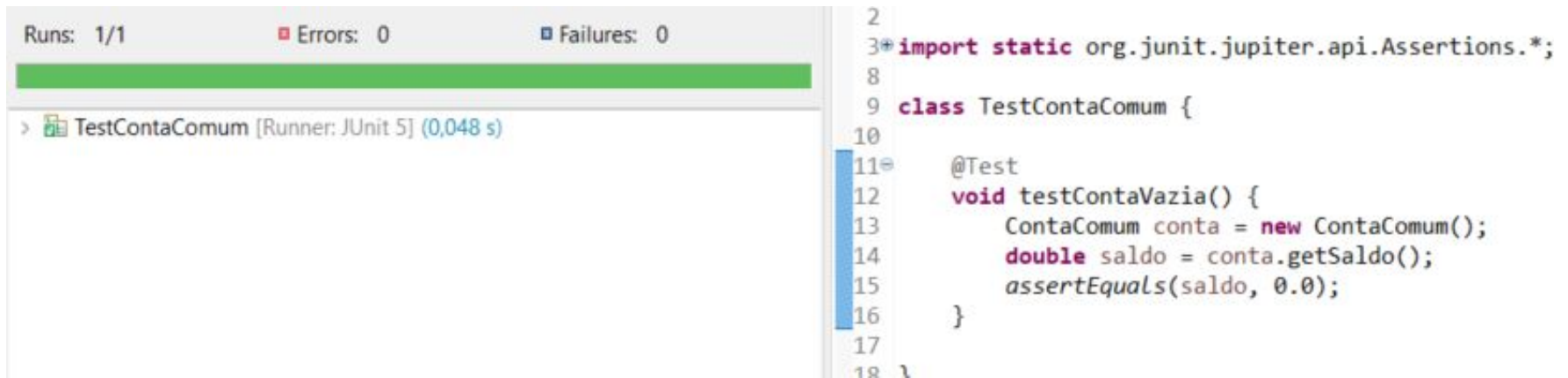
```
@Test
void testContaVazia() {
    ContaComum conta = new ContaComum();
    double saldo = conta.getSaldo();
    assertEquals(saldo, 0.0);
}
```

Contexto do teste

Chamada

Afirmação

Exemplo de Testes Unitários com JUnit



The screenshot displays an IDE interface. On the left, a test runner window shows the results of a test execution. It indicates that 1 out of 1 test runs were successful, with 0 errors and 0 failures. A green progress bar is visible. Below this, the test name 'TestContaComum' is listed, along with the runner 'JUnit 5' and the execution time '(0,048 s)'. On the right, the source code for the test is shown. It includes an import statement for JUnit's Assertions, a class definition 'TestContaComum', and a single test method 'testContaVazia'. This method creates a 'ContaComum' object, retrieves its balance using 'getSaldo()', and asserts that it is equal to 0.0 using 'assertEquals()'. The code is syntax-highlighted, and line numbers are visible on the left margin of the code editor.

```
2
3 import static org.junit.jupiter.api.Assertions.*;
8
9 class TestContaComum {
10
11     @Test
12     void testContaVazia() {
13         ContaComum conta = new ContaComum();
14         double saldo = conta.getSaldo();
15         assertEquals(saldo, 0.0);
16     }
17
18 }
```


Exemplo de Testes Unitários com JUnit

Runs: 1/1 Errors: 0 Failures: 1

TestContaComum [Runner: JUnit 5] (0,036 s)
testContaVazia() (0,036 s)

```
2
3 import static org.junit.jupiter.api.Assertions.*;
8
9 class TestContaComum {
10
11     @Test
12     void testContaVazia() {
13         ContaComum conta = new ContaComum();
14         double saldo = conta.getSaldo();
15         assertEquals(saldo, 1.0);
16     }
17
18 }
```

Failure Trace

org.opentest4j.AssertionFailedError: expected: <0.0> but was: <1.0>
at unit.testing.TestContaComum.testContaVazia(TestContaComum.java:15)
at java.base/java.util.ArrayList.forEach(ArrayList.java:1507)
at java.base/java.util.ArrayList.forEach(ArrayList.java:1507)

Exemplo de Testes Unitários com JUnit

```
class TestContaComum {  
  
    ContaComum conta;  
  
    @BeforeEach  
    public void init() {  
        conta = new ContaComum();  
    }  
  
    @Test  
    public void testContaVazia() {  
        double saldo = conta.getSaldo();  
        assertEquals(0.0, saldo);  
    }  
  
    @Test  
    public void testDeposita() {  
        conta.deposita(3.0);  
        assertEquals(3.0, conta.getSaldo());  
    }  
  
    @Test  
    public void testeRende() {  
        conta.deposita(5.0);  
        conta.rende();  
        double novoValor = 5.0 * 1.1;  
        assertEquals(novoValor, conta.getSaldo());  
    }  
}
```

@BeforeEach anota um método que deve ser executado antes de cada método @Test

Tratamento de Exceção com JUnit 5

- Exemplo com exceção

```
public void rende() throws Exception {  
    if(saldo <= 0.0) {  
        throw new Exception("valor menor ou igual a 0");  
    }  
    this.saldo *= 1.1;  
}
```

```
@Test  
public void testeRende() throws Exception {  
    conta.deposita(5.0);  
    conta.rende();  
    double novoValor = 5.0 * 1.1;  
    assertEquals(novoValor, conta.getSaldo());  
}
```

```
@Test  
public void testeRendeComException() throws Exception {  
    Assert.assertThrows(Exception.class, () -> {  
        conta.rende();  
    });  
}
```

Definições relacionadas

- **Teste**
 - Método que implementa um teste e possuir anotação @Test
- **Fixture**
 - Estado do sistema que será testado por um ou mais métodos de teste, incluindo dados, objetos, etc.
- **Suíte de Testes (Test Suite):**
 - Conjunto de casos de teste, os quais são executados pelo framework de testes de unidade (que no nosso caso é o JUnit).
- **Sistema sob Teste (System Under Test, SUT):**
 - Sistema que está sendo testado

Quando Escrever Testes de Unidade?

- **Incremental**
 - Programar um pouco, escrever testes... programar mais um pouco, escrever novos testes
- **TDD**
 - Criar primeiro os testes, antes do código
- **Durante análise de bugs**
 - Escrever um teste que reproduz o bug. Se corrigirmos o código, o teste deve passar
 - Ao invés de depurar usando por exemplo `System.out.println`, prefira escrever um método de testes

Princípios do Teste de Unidade

- **FIRST**
 - Rápidos (FAST)
 - Independentes (Independent)
 - Determinísticos (Repeatable)
 - Auto-verificáveis (Self-checking)
 - Escritos o quanto antes (Timely)

Princípios do Teste de Unidade

- **Rápidos (FAST)**
 - Testes de unidade devem ser executados rapidamente, em questões de milisegundos
 - Se não for possível, dividir a suíte de testes
 - Testes rápidos
 - Uso frequente
 - Testes mais demorados
 - Executados com certa frequência (e.g., 1 vez ao dia)

Princípios do Teste de Unidade

- **Independentes (Independent)**
 - A ordem de execução dos testes de unidade não deve ser importante
 - A execução de T1 seguida de T2 ou T2 e depois T1 deve ter o mesmo resultado

Princípios do Teste de Unidade

- **Determinísticos (Repeatable)**
 - Testes de unidade devem ter sempre o mesmo resultado

*Testes com resultados não-determinísticos
são chamados de **Testes Flaky** (ou Testes
Erráticos)*

Princípios do Teste de Unidade

- **Auto-verificáveis (Self-checking)**
 - O resultado de um teste de unidades deve ser facilmente verificável.
 - Não deve depender de verificação manual
 - quando um teste falha, deve ser possível identificar essa falha de forma rápida, incluindo a localização do comando assert que falhou.

Princípios do Teste de Unidade

- **Escritos o quanto antes (Timely)**
 - Quanto mais cedo os testes forem escritos, melhor!



Test Smells

- São características “preocupantes” no código de testes de unidade
- Exemplos
 - **Teste Obscuro**
 - Teste longo, difícil de entender
 - **Teste com Lógica Condicional**
 - Teste que possui código que pode ou não ser executado
 - **Duplicação de Código em Testes**
 - Código duplicado em vários testes

Número de asserts por teste

- Em geral, recomenda-se ter no máximo **um assert** por teste

```
@Test
public void testEmptyStack() {
    assertTrue(stack.isEmpty());
}

@Test
public void testNotEmptyStack() {
    stack.push(10);
    assertFalse(stack.isEmpty());
}
```

```
@Test
public void testEmptyStack() {
    assertTrue(stack.isEmpty());
    stack.push(10);
    assertFalse(stack.isEmpty());
}
```

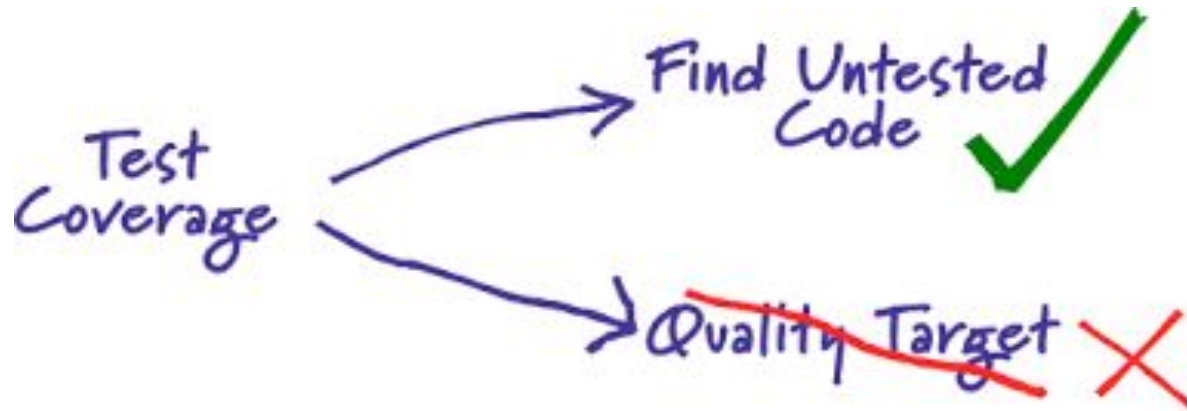
Número de asserts por teste

- mas sempre existem exceções...

```
@Test
public void testBookService() {
    BookService bs = new BookService();
    Book b = bs.getBook(1234);
    assertEquals("Engenharia Software Moderna", b.getTitle());
    assertEquals("Marco Tulio Valente", b.getAuthor());
    assertEquals("2020", b.getYear());
    assertEquals("ASERG/DCC/UFGM", b.getPublisher());
}
```

Cobertura dos Testes

- Mede o percentual de cobertura do código de testes

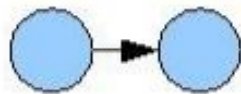


Grafo de fluxo de controle (CFG)

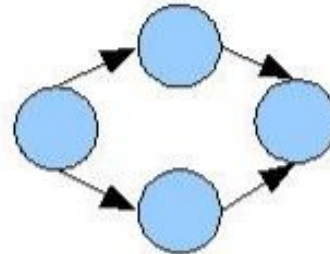
- **Composto por**
 - **Nós**
 - Um ou mais instruções executadas em sequência
 - **Arcos**
 - Fluxo de controle entre bloco de comandos (nós)
-

Sintaxe do Grafo de fluxo de controle (CFG)

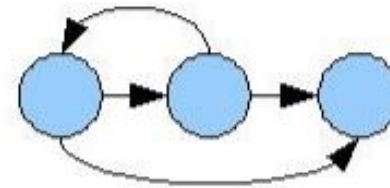
As Construções Estruturadas em
forma de grafo de fluxo



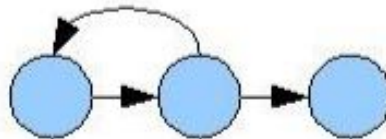
Seqüência



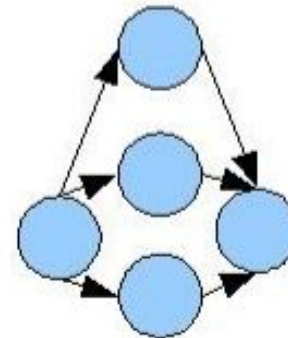
Se-então-senão



Enquanto



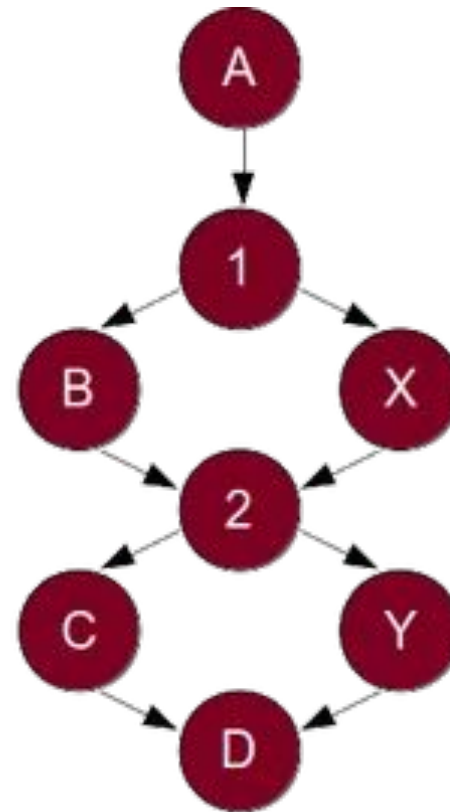
Repita



Caso

Exemplo de Grafo de fluxo de controle (CFG)

```
print("A")  
if (condition1)  
    print("X")  
else  
    print("B")  
if (condition2)  
    print("Y")  
else  
    print("C")  
print("D")
```



Tipos de Cobertura dos Testes unitários

- **Cobertura de Comandos**
 - percentual de cobertura de instruções do código
 - **Cobertura de branches (ramificações)**
 - percentual de branches de um programa que são executados por um teste
 - Comando IF sempre gera 2 branches (verdadeiro, falso)
 - **Todos os nós, todas arestas ...**
-

Complexidade Ciclomática

- **Complexidade Ciclomática mede a complexidade de um programa**
 - Mede quantidade de caminhos linearmente independentes
 - Forte relação com testabilidade
 - Indica a dificuldade de se construir casos de testes de unidade
 - Auxilia a identificar o que precisa ser testado
 - Pode ser usado para garantir que todas as instruções sejam executadas pelo menos uma vez durante o teste unitário
-

Complexidade Ciclomática

- **Fórmulas de Cálculo**

- **$V(G) = E - N + 2$**

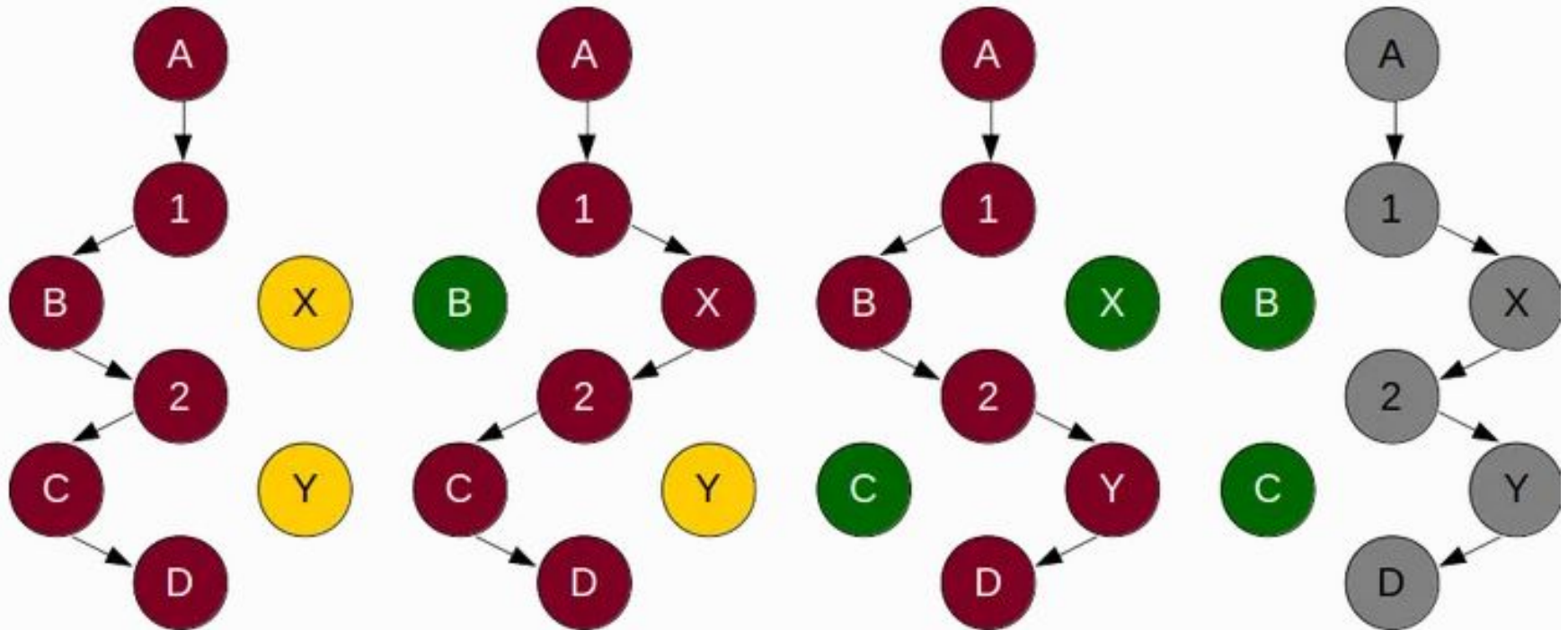
- E é o número de arestas do grafo

- N é o número de nós do grafo

- **$V(G) = P + 1$**

- P é o número de nós que podem desviar o fluxo da execução
(if, while, switch)

Complexidade Ciclomática



Independentes: A-1-B-2-C-D A-1-X-2-C-D A-1-B-2-Y-D

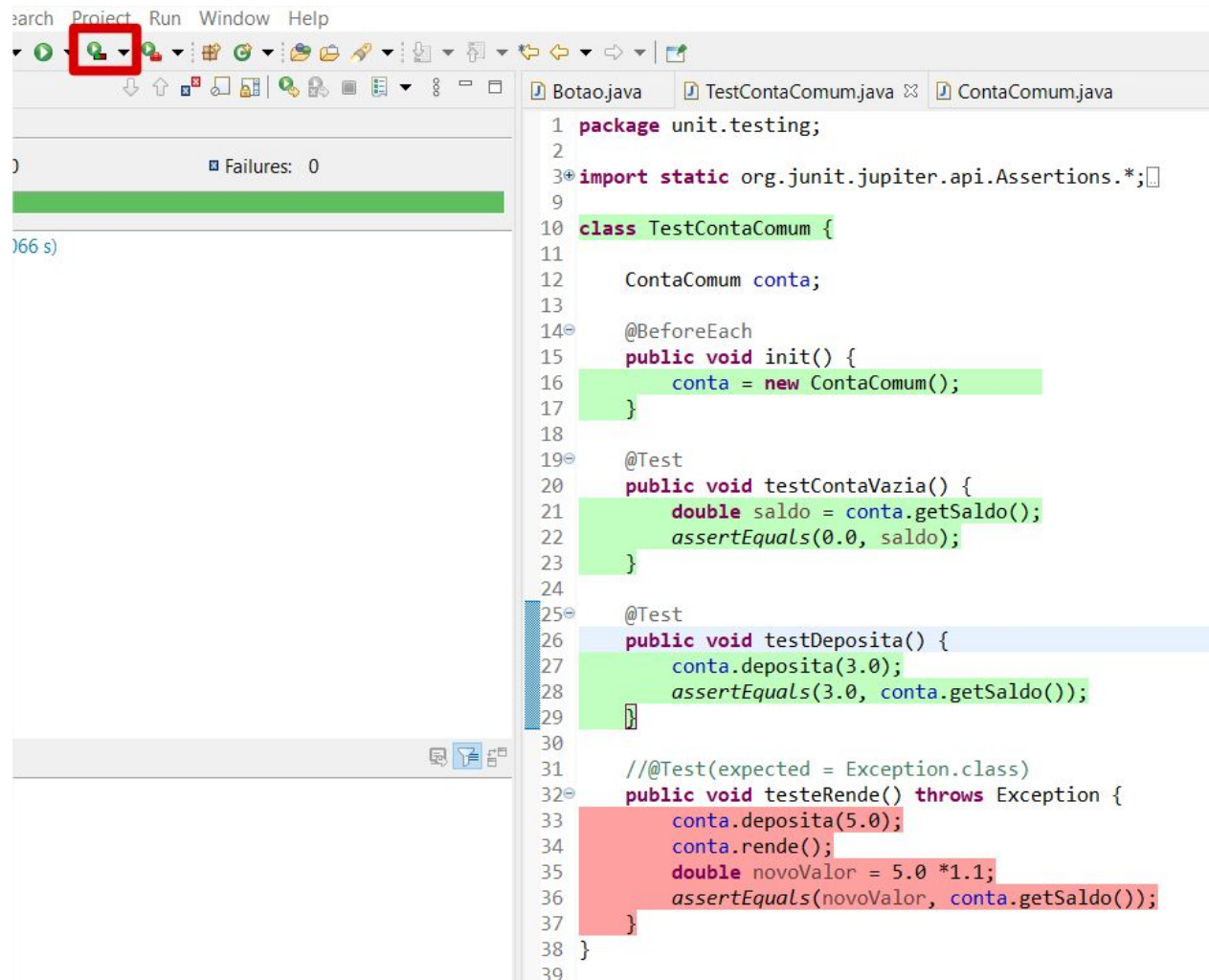
Não Independente: A-1-X-2-Y-D

Caminhos possíveis = 4

Complexidade Ciclomática = 3

Cobertura dos Testes

- Exemplo



The screenshot shows an IDE with a toolbar at the top. A red box highlights the 'Run' button (a green play icon). Below the toolbar, the 'TestContaComum.java' file is open. The code defines a `TestContaComum` class with three test methods: `init`, `testContaVazia`, and `testDeposita`. The `testDeposita` method is currently selected. The bottom panel shows the test results, indicating that all tests passed successfully with 0 failures and a total execution time of 0.66 seconds.

```
1 package unit.testing;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 class TestContaComum {
6     ContaComum conta;
7
8     @BeforeEach
9     public void init() {
10         conta = new ContaComum();
11     }
12
13     @Test
14     public void testContaVazia() {
15         double saldo = conta.getSaldo();
16         assertEquals(0.0, saldo);
17     }
18
19     @Test
20     public void testDeposita() {
21         conta.deposita(3.0);
22         assertEquals(3.0, conta.getSaldo());
23     }
24
25     // @Test(expected = Exception.class)
26     public void testeRende() throws Exception {
27         conta.deposita(5.0);
28         conta.rende();
29         double novoValor = 5.0 * 1.1;
30         assertEquals(novoValor, conta.getSaldo());
31     }
32 }
```

Failures: 0
0.66 s

Cobertura dos Testes

- Exemplo

```
package solid.to.refactor.lsp.ruim;


public class ContaComum {

    protected double saldo;

    public ContaComum() {
        this.saldo = 0;
    }

    public void deposita(double valor) {
        if (valor > 5.0)
            this.saldo += valor;
    }

    public double getSaldo() {
        return saldo;
    }
}
```

Coverage					
TestContaComum (8 de dez de 2020 15:24:35)					
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions	
> CK0236	 16,6 %	70	352	422	

Qual a cobertura ideal?

- Não existe um número mágico
 - Existem métodos triviais (gettes, setters)
- Recomenda-se monitorar a evolução da cobertura ao longo do tempo e avaliar trechos não cobertos
- Cobertura abaixo de 50% é preocupante

Testabilidade

- **Testabilidade**
 - É uma medida de quão fácil é implementar testes para um sistema
- **Usar as princípios de projeto ajudam na testabilidade**
 - responsabilidade única
 - inversão de dependências
 - Lei de Demeter
 - ...

Mocks e Stubs

```
import poo.aula04.Aluno;
```

```
//aula 06
```

```
public class ImprimirNota {
```

```
    Sigaa sigaa;
```

```
    Aluno aluno = new Aluno();
```

```
    public ImprimirNota(Aluno aluno, Sigaa sigaa) {
```

```
        this.aluno = aluno;
```

```
        this.sigaa = sigaa;
```

```
    }
```

```
    public void print() {
```

```
        System.out.println(sigaa.getMedia(aluno.getMatricula()));
```

```
    };
```

```
}
```

```
//aula 06
```

```
public interface Sigaa {
```

```
    public double getMedia(int matricula);
```

```
}
```

“Sigaa é uma interface que pode ser implementada por um serviço externo”

Como testar a classe ImprimirNota com essa dependência?

Mocks e Stubs

- **Mock**
 - Objeto que 'emula' o objeto real
 - Ele retorna os valores desejados para isolar a classe sob teste
 - Verificam o estado e comportamento
 - **Stub**
 - Mocks que verificam apenas o estado
-

Mocks e Stubs

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import poo.aula04.Aluno;

class TestImprimirNota {

    ImprimirNota imprimirNota;

    @BeforeEach
    public void init() {
        MockUFC mock = new MockUFC();
        Aluno aluno = new Aluno();
        aluno.setMatricula(5);
        imprimirNota = new ImprimirNota(aluno, mock);
    }

    @Test
    void testPrint() {
        assertEquals("10.0 - 5",imprimirNota.print());
    }
}

public class MockUFC implements Sigaa{

    @Override
    public double getMedia(int matricula) {
        // Cálculo da Média da UFC
        // acesso a banco de dados etc..
        return 10;
    }
}
```

O teste aqui foca em verificar se a impressão final é correta

Mocks e Stubs

- Framework de Mocks



Mocks e Stubs


```
import poo.aula04.Aluno;

class TestImprimirNotaComMockito {

    ImprimirNota imprimirNota;

    @BeforeEach
    public void init() {
        Sigaa sigaaMock = Mockito.mock(Sigaa.class);
        Mockito.when(sigaaMock.getMedia(5)).thenReturn(10.0);
        Aluno aluno = new Aluno();
        aluno.setMatricula(5);
        imprimirNota = new ImprimirNota(aluno, sigaaMock);
    }

    @Test
    void testPrint() {
        assertEquals("10.0 - 5", imprimirNota.print());
    }
}
```



Agora não preciso
escrever classes de mock
manualmente!

Obrigado!

Por hoje é só pessoal...

Dúvidas?



qpg4p5x



ismaylesantos@great.ufc.br



@IsmayleSantos
