

INSTITUTO FEDERAL
Ceará

Campus
Tianguá

DOI: 10.5281/zenodo.10444780

TESTES DE SOFTWARE (PARTE 2)

Engenharia de Software

Profa. Cynthia Pinheiro

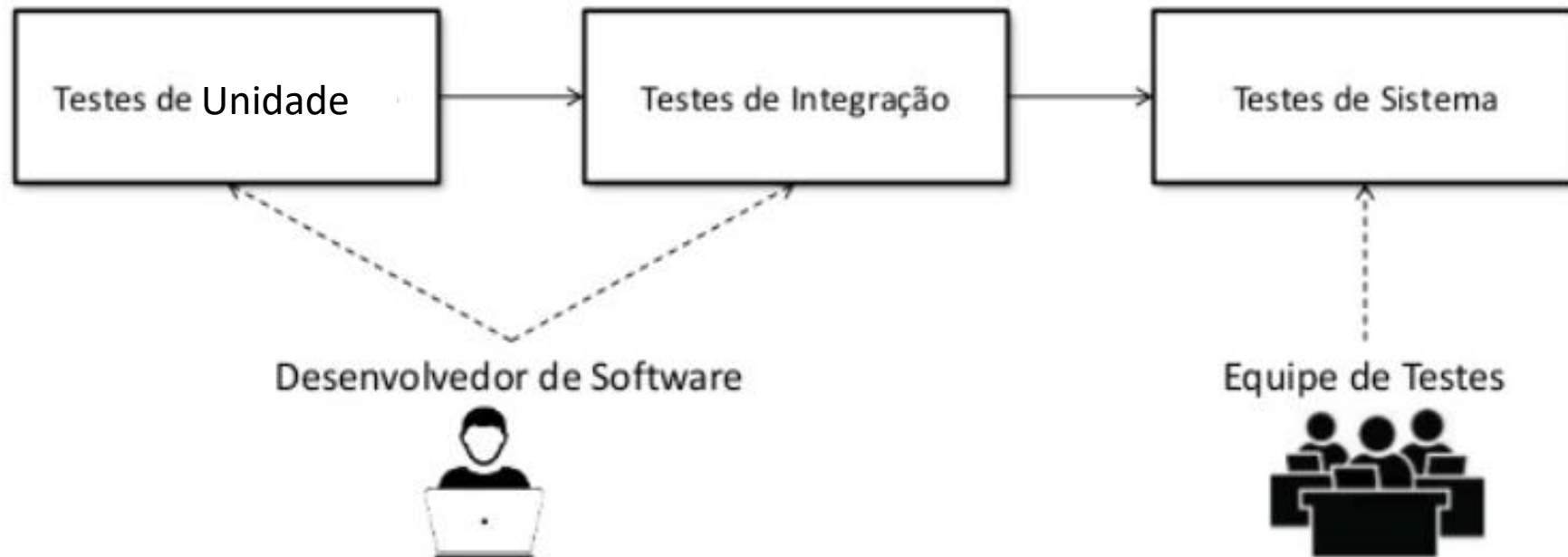


Esta obra está licenciada com uma Licença Creative Commons Atribuição 4.0 Internacional

Testes de Software

- *Processo* de Testes

- Deve ser realizado em *fases*, de forma a tornar os *testes mais eficazes*:



Testes de Software

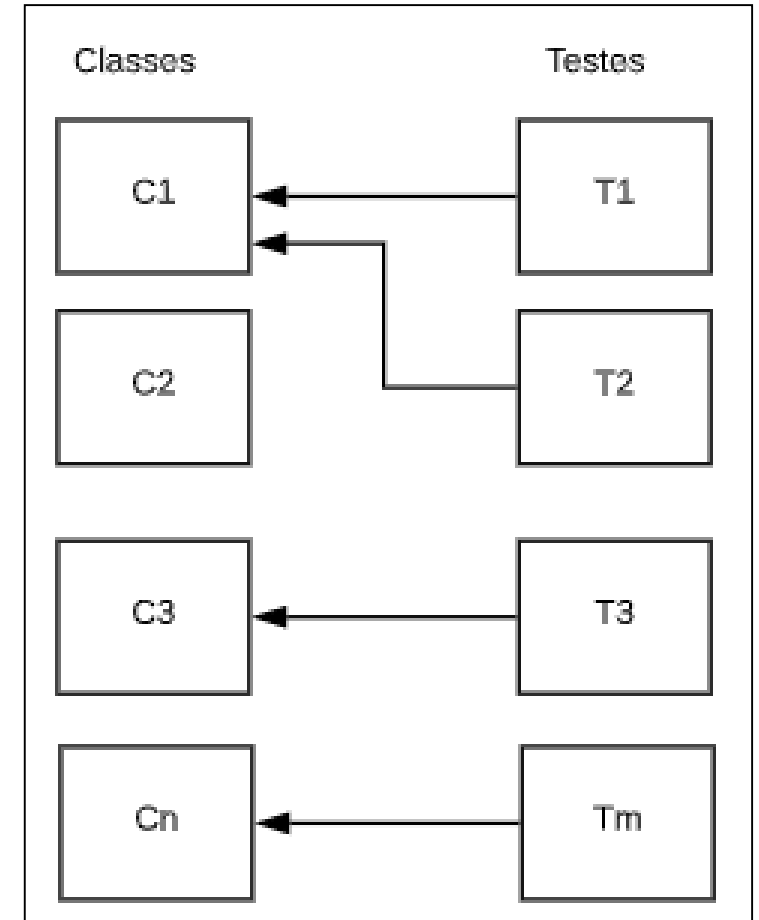
- *Teste de Unidade:*

- *Unidade:* pode ser um *método* ou procedimento, uma *classe* completa, um *pacote* de funções ou classes.
- São realizados pelo *próprio programador* sendo inclusive aconselhável desenvolver o *driver de teste* da unidade *antes* da própria unidade.
- *Driver de teste:* programa que *obtem ou gera um conjunto de dados* que serão usados para *testar sistematicamente* o componente que ainda vai ser desenvolvido.
 - Chama métodos de uma classe e *verifica se eles retornam os resultados esperados*.
- *Teste estrutural:* garantir que pelo menos *todos os comandos e decisões* do componente implementado *foram testados* para checar se eles têm defeitos.

Testes de Software

- *Teste de Unidade:*

- Quando se usa testes de unidade, o código de um sistema pode ser dividido em *dois grupos*:
 - Um conjunto de *classes de sistema*, que implementam os requisitos
 - Um conjunto de *classes de teste*, que testam as classes do sistema.
 - Uma classe pode ter *um, vários ou nenhum teste*.
- Testes de unidade são implementados usando-se *frameworks*.
 - Mais conhecidos: de frameworks *xUnit*, onde o x indica a linguagem usada na implementação.
 - Exemplo: *Junit* para *Java*.



Testes de Software

- *Teste de Unidade:*

- JUnit permite implementar classes que vão *testar*, de forma *automática*, as classes da aplicação.
- Por convenção, classes de teste têm o *mesmo nome* das classes testadas, mas com o *sufixo Test*.
- Os métodos de teste começam com o *prefixo test* e devem atender às seguintes condições:
 - serem *públicos*, já que serão chamados pelo JUnit
 - *Não* possuir *parâmetros*
 - Possuir a anotação *@Test*, que identifica métodos que deverão ser executados durante um teste.

```
import java.util.ArrayList;
import java.util.EmptyStackException;

public class Stack<T> {

    private ArrayList<T> elements = new ArrayList<T>();
    private int size = 0;

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return (size == 0);
    }

    public void push(T elem) {
        elements.add(elem);
        size++;
    }

    public T pop() throws EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException();
        T elem = elements.remove(size-1);
        size--;
        return elem;
    }
}
```

Testes de Software

• *Teste de Unidade:*

```
import org.junit.Test;
import static org.junit.Assert.assertTrue;

public class StackTest {

    @Test
    public void testEmptyStack() {
        Stack<Integer> stack = new Stack<Integer>();
        boolean empty = stack.isEmpty();
        assertTrue(empty);
    }
}
```



Este método *cria uma pilha* (Stack) e *testa se ela está vazia*.

Contexto do teste (fixture):
Deve-se *instanciar* os objetos que se pretende testar e, se for o caso, *inicializá-los*.

O teste deve *chamar um dos métodos* da classe que está *sendo testada* e seu resultado é armazenado em uma variável local.

Comandos "assert": testa se um determinado resultado é igual a um valor esperado.
Exemplo: *assertTrue*, que verifica se o *valor passado como parâmetro é verdadeiro*.

```
import java.util.ArrayList;
import java.util.EmptyStackException;

public class Stack<T> {

    private ArrayList<T> elements = new ArrayList<T>();
    private int size = 0;

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return (size == 0);
    }

    public void push(T elem) {
        elements.add(elem);
        size++;
    }

    public T pop() throws EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException();
        T elem = elements.remove(size-1);
        size--;
        return elem;
    }
}
```

Testes de Software

- *Teste de Unidade:*

```
import org.junit.Test;
import static org.junit.Assert.assertTrue;

public class StackTest {

    @Test
    public void testEmptyStack() {
        Stack<Integer> stack = new Stack<Integer>();
        boolean empty = stack.isEmpty();
        assertTrue(empty);
    }

}
```



IDEs oferecem opções para rodar *apenas os testes de unidade* um sistema (*“Run as Test”*).

O código fonte completo de Stack e StackTest está disponível em <https://gist.github.com/mtov/3601acd0b32a1d0a85b4a81a43af4284>.

```
import java.util.ArrayList;
import java.util.EmptyStackException;

public class Stack<T> {

    private ArrayList<T> elements = new ArrayList<T>();
    private int size = 0;

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return (size == 0);
    }

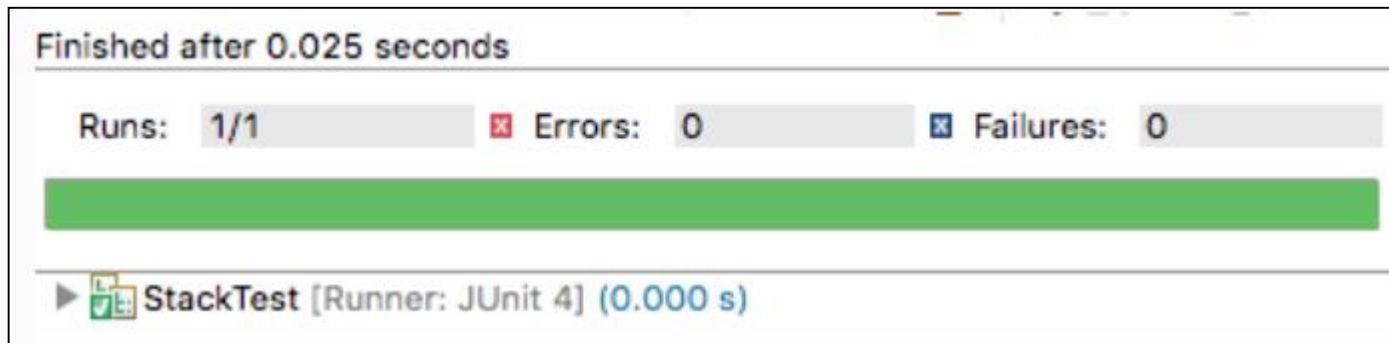
    public void push(T elem) {
        elements.add(elem);
        size++;
    }

    public T pop() throws EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException();
        T elem = elements.remove(size-1);
        size--;
        return elem;
    }

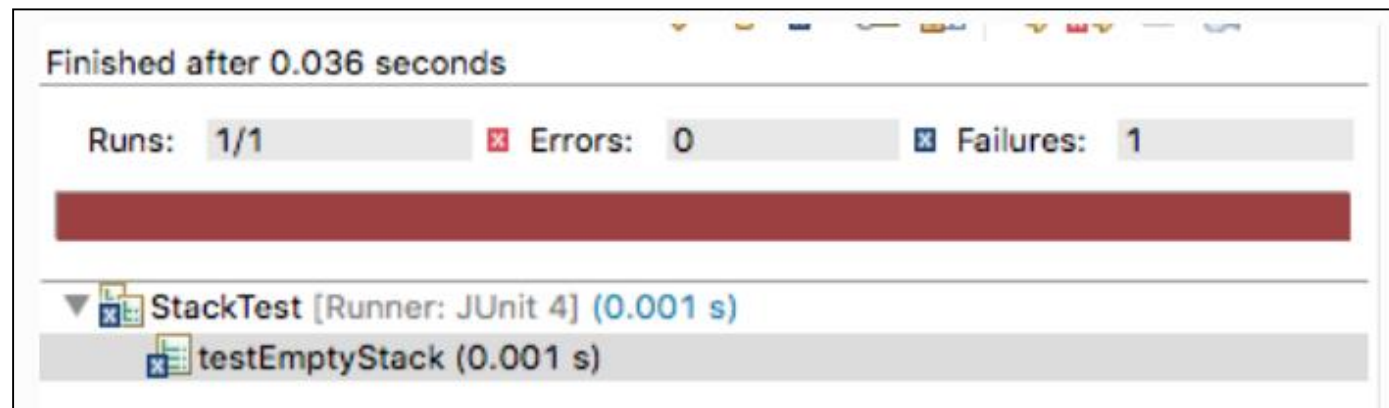
}
```

Testes de Software

- *Resultado do Teste de Unidade:*



Todos os testes passaram.
Resultado em 0.025 seg.



Houve uma falha (failure)
durante a execução de
testEmptyStack: algum
comando assert não foi
satisfeito.

Testes de Software

- *Benefícios dos Testes de Unidade:*

- *Encontrar bugs*, ainda na fase de desenvolvimento e antes que o código entre em produção (custos de correção maiores).
- *Proteção contra regressões* no código
 - *Regressão*: ocorre quando uma *modificação realizada no código* — seja para corrigir um bug, implementar uma nova funcionalidade ou realizar uma refatoração — acaba por *introduzir um bug ou outro problema* semelhante no código.
 - Após concluir uma mudança, o desenvolvedor deve novamente *rodar toda a bateria de testes*.
- Ajudam na *documentação e especificação* do código de produção.

Testes de Software

- *Testes de Unidade: Princípios FIRST*
 - Rápidos (**F**ast): é importante que os testes sejam *executados rapidamente*, em questões de milissegundos.
 - Independentes: a *ordem de execução* dos testes de unidade *não é importante*.
 - Determinísticos (**R**epeatable): devem ter sempre o mesmo resultado.
 - Auto verificáveis (**S**elf-checking): o *resultado* de um teste de unidades deve ser *facilmente verificável*.
 - Escritos o quanto antes (**T**imely): se possível *antes* mesmo do código que vai ser testado

Testes de Software

- *Testes de Unidade:*
Cobertura de Testes
 - Existem ferramentas para *cálculo de cobertura de testes*:
IDE Eclipse
 - *Linhas verdes*: correspondem a comandos executáveis no programa *cobertos por testes*.
 - Cobertura de testes no caso ao lado: *100%*.

```
public class Stack<T> {  
    private ArrayList<T> elements = new ArrayList<T>();  
    private int size = 0;  
  
    public int size() {  
        return size;  
    }  
  
    public boolean isEmpty(){  
        return (size == 0);  
    }  
  
    public void push(T elem) {  
        elements.add(elem);  
        size++;  
    }  
  
    public T pop() throws EmptyStackException {  
        if (isEmpty())  
            throw new EmptyStackException();  
        T elem = elements.get(size-1);  
        size--;  
        return elem;  
    }  
}
```

Testes de Software

- *Testes de Unidade:*
Cobertura Ideal

- Times que valorizam a escrita de testes costumam atingir *valores de cobertura próximos de 70%*.
- Valores *abaixo de 50%* tendem a ser *preocupantes*.
- Com *TDD* (Test Driven Development), a cobertura de testes costuma *não chegar a 100%*, embora normalmente *fique acima de 90%*

```
public class Stack<T> {  
    private ArrayList<T> elements = new ArrayList<T>();  
    private int size = 0;  
  
    public int size() {  
        return size;  
    }  
  
    public boolean isEmpty(){  
        return (size == 0);  
    }  
  
    public void push(T elem) {  
        elements.add(elem);  
        size++;  
    }  
  
    public T pop() throws EmptyStackException {  
        if (isEmpty())  
            throw new EmptyStackException();  
        T elem = elements.get(size-1);  
        size--;  
        return elem;  
    }  
}
```

Em vermelho: Comando não coberto por testes

Bibliografia

- *Esta aula foi retirada dos seguintes livros/artigos:*
 - HIRAMA, Keshi. *Engenharia de Software: qualidade e produtividade com tecnologia*. Rio de Janeiro, Elsevier, 2011.
 - Material do prof. *Ricardo A. Ramos*, ICMC – USP.
 - Valente, Marco. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. 2020. Disponível em <https://engsoftmoderna.info/>

