

# Introdução ao Teste de Software

## Automatização

Vinicius H. S. Durelli

✉ [durelli@ufsj.edu.br](mailto:durelli@ufsj.edu.br)



# Organização

- 1 Automatização
- 2 Componentes de um caso de teste (como artefatos de software)
- 3 Framework para automatização de casos de teste: JUnit
  - Casos de teste
  - Asserções
  - Modelo RIPR no JUnit
- 4 Considerações finais

- 1 Automatização
- 2 Componentes de um caso de teste (como artefatos de software)
- 3 Framework para automatização de casos de teste: JUnit
  - Casos de teste
  - Asserções
  - Modelo RIPR no JUnit
- 4 Considerações finais

# O que é automatização?

## Definição → Automatização

Consiste no uso de software para controlar diversos aspectos relacionados aos testes como, por exemplo, **execução** dos testes, **comparação dos resultados esperados com os resultados obtidos**, **configuração de pré-condições** de teste (Ammann & Offutt 2016).

Vantagens:

- Reduz custo envolvido;
- Mitiga as chances de erros humanos;
- Reduz o custo de regressão.

- 1 Automatização
- 2 Componentes de um caso de teste (como artefatos de software)
- 3 Framework para automatização de casos de teste: JUnit
  - Casos de teste
  - Asserções
  - Modelo RIPR no JUnit
- 4 Considerações finais

## Falando sobre casos de teste...

A parte mais mencionada de um caso de teste é denominada “valor(es)/entradas de caso de teste” (do inglês, *test case values*):

### Definição → Valores/Entradas de Caso de Teste

As entradas necessárias para completar uma execução do software sendo testado (Ammann & Offutt 2016).

Tal definição é bem ampla.

- Como seriam as entradas de caso de teste para uma aplicação Web?
- E para um sistema de tempo real (e.g., sistema de controle de voo)?

➡ Porém, é importante notar que **testes incluem muito mais do que só entradas** para “exercitar” o software sendo testado.

## Prefixos e sufixos de teste...

\* Casos de teste são **artefatos de software com várias partes**.

Dependendo do software sendo testado, o testador precisa fornecer outras entradas para afetar *controllability* e *observability*.

### Definição → Prefixo de Teste

As entradas necessárias colocar o software em um estado apropriado para receber os valores de caso de teste (Ammann & Offutt 2016).

### Definição → Sufixo de Teste

As entradas que precisam ser fornecidas ao software depois dos valores de caso de teste (Ammann & Offutt 2016).

## Prefixos e sufixos de teste...

\* Casos de teste são **artefatos de software com várias partes**.

Dependendo do software sendo testado, o testador precisa fornecer outras entradas para afetar *controllability* e *observability*.

### Definição → Prefixo de Teste

As entradas necessárias colocar o software em um estado apropriado para receber os valores de caso de teste (Ammann & Offutt 2016).

Sufixos de teste podem ser divididos em **duas subcategorias**:

### Definição → Verificação

Valores necessários para visualização dos resultados do caso de teste (Ammann & Offutt 2016).

### Definição → Retorno

Valores ou comandos usados para terminar o programa ou retorná-lo para um estado estável (Ammann & Offutt 2016).



## Verificando o resultado da execução. . .

Obviamente, é necessário decidir se o resultado de uma determinada execução coincide com a saída esperada.

### Oráculo de Teste

Um oráculo (de teste) verifica se a saída de um programa é correta. Em outras palavras, o oráculo decide se um determinado caso de teste “passou” ou “falhou”.

\* O resultado que deve ser produzido pelo software sendo testado, caso o mesmo se comporte corretamente, deve ser incluído no caso de teste.

### Definição → Resultado Esperado

Resultado que deve ser produzido pelo caso de teste quando o software se comporta como esperado (Ammann & Offutt 2016).

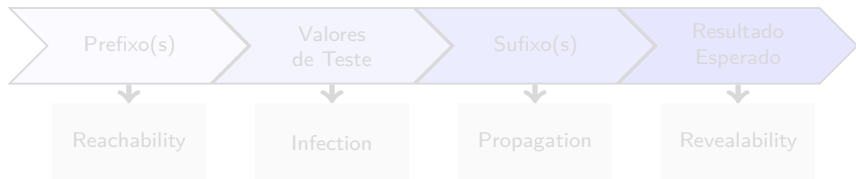
# Afinal de contas, o que é um caso de teste?

Um caso de teste inclui todos os elementos mencionados anteriormente: (i) valores de caso de teste, (ii) prefixo, (iii) sufixo e (iv) resultado esperado.

## Definição → Caso de Teste

Um caso de teste é composto dos valores de caso de teste, prefixo, sufixo e resultado esperado necessários para uma completa execução e avaliação do software sendo testado (Ammann & Offutt 2016).

Os componentes em um caso de teste são “instâncias” do modelo RIPR:



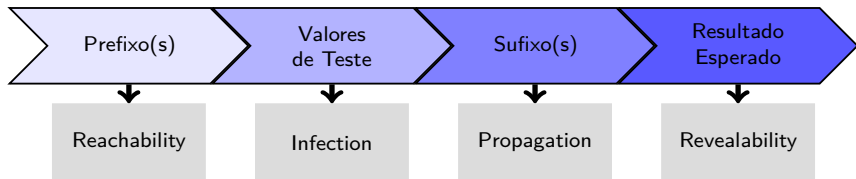
# Afinal de contas, o que é um caso de teste?

Um caso de teste inclui todos os elementos mencionados anteriormente: (i) valores de caso de teste, (ii) prefixo, (iii) sufixo e (iv) resultado esperado.

## Definição → Caso de Teste

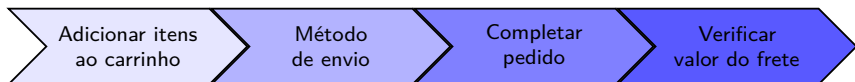
Um caso de teste é composto dos valores de caso de teste, prefixo, sufixo e resultado esperado necessários para uma completa execução e avaliação do software sendo testado (Ammann & Offutt 2016).

Os componentes em um caso de teste são “instâncias” do modelo RIPR:



## Exemplo

Considerando o método `estimateShipping()` que calcula o valor do frete considerando os itens no carrinho de compras. Um caso de teste que verifica o valor do frete poderia incluir os seguintes elementos (Ammann & Offutt 2016):



É importante notar a **complexidade** envolvida em tal caso de teste:

- Criação de objetos (e.g., carrinho de compras);
- Cada vez que o teste for executado, é importante garantir que itens não sejam retirados do depósito;
- Caso de teste deve ser executado várias vezes.

- 1 Automatização
- 2 Componentes de um caso de teste (como artefatos de software)
- 3 Framework para automatização de casos de teste: JUnit
  - Casos de teste
  - Asserções
  - Modelo RIPP no JUnit
- 4 Considerações finais

# JUnit

Framework open-source,<sup>1</sup> escrito por Kent Beck Erich Gamma, para criação de testes de unidade para a linguagem Java. A primeira versão foi lançada em 1997.

## Definição → Framework para Testes Unitários

Ferramenta cujo propósito é facilitar a criação e execução de testes de unidade (e integração).

- Faz parte da família xUnit: inclui implementações C#, C++, etc.
- Amplamente utilizado na indústria.
- Pode ser utilizado de forma *stand-alone* ou no contexto de uma IDE.<sup>2</sup>

---

<sup>1</sup><https://junit.org/junit5/>

<sup>2</sup>Integrated Development Environment

## Criando casos de teste usando JUnit (1)


Primeiro, é necessário importar os seguintes pacotes:

```
import static org.junit.jupiter.api.Assertions.assertEquals;  
import org.junit.jupiter.api.Test;
```

Uma classe de teste incluindo um método/teste de unidade pode ser definida como a seguir:<sup>3</sup>

```
class FirstJUnitTest {  
    @Test  
    void myFirstTest() {  
        assertEquals(2, 1 + 1); //1 + 1 = 2?  
    }  
}
```

---

<sup>3</sup>Classes e métodos de teste não precisam ser públicos. 

## Criando casos de teste usando JUnit (2)

Na versão atual do framework, métodos teste devem ser anotados com `org.junit.jupiter.api.Test`.

**@Test**

```
void isEmpty() {  
    ArrayList<Integer> list = new ArrayList<Integer>();  
    assertTrue(list.isEmpty(), () -> "Deve ser vazia");  
}
```

**Asserções** (*assertions*) são usadas para verificar o comportamento esperado. Cada asserção testa se uma determinada condição é verdadeira ou não.

Quando uma condição não é verdadeira, o framework reporta uma falha.



# Asserções (1)

Algumas das asserções mais utilizadas:

| Asserções                    |   |
|------------------------------|---|
| Método                       | Funcionalidade  |
| <code>assertTrue</code>      | Verifica se a condição é verdadeira                     |
| <code>assertFalse</code>     | Verifica se a condição é falsa                          |
| <code>assertNull</code>      | Verifica se a referência é nula                         |
| <code>assertNotNull</code>   | Verifica se a referência não é nula                     |
| <code>assertEquals</code>    | Verifica se o resultado esperado é igual ao obtido      |
| <code>assertNotEquals</code> | Verifica se o resultado esperado é diferente do obtido  |
| <code>assertSame</code>      | Verifica se o objeto esperado e o obtido são iguais     |
| <code>assertNotSame</code>   | Verifica se o objeto esperado e o obtido não são iguais |

Asserções que recebem dois parâmetros seguem o seguinte padrão: o primeiro parâmetro é o valor esperado e o segundo é o valor obtido.

## Asserções (2)

Basicamente, asserções podem ser utilizadas de três formas.

```
1 @Test
2 void nullAssertionTest() {
3     String str = null;
4     ❶ assertNull(str);
5     ❷ assertNull(str, "str deveria ser null");
6     ❸ assertNull(str, () -> "str deveria ser null");
7 }
```

- A versão ❷ (linha 5), passa uma *string* como parâmetro. Tal *string* é exibida em caso de falha.
- A versão ❸ (linha 6), recebe uma expressão lambda como parâmetro. Tal expressão só é avaliada se a asserção falhar.

## Agrupando asserções

Embora recomende-se que cada método contenha somente uma asserção, é possível agrupar várias asserções em um método por meio de `assertAll`.

```
1 @Test  
2 void bunchOfAssertions() {  
3     int n = 2;  
4     boolean v = true;  
5     assertAll((() -> assertTrue(v), ❶  
6               (() -> assertEquals(2, n) ❷));  
7 }
```

- Todas as asserções são executadas como se fossem uma só.
- Cada asserção é especificada como uma expressão lambda.
- O framework reporta quantas asserções falharam.



- 1 Automatização
- 2 Componentes de um caso de teste (como artefatos de software)
- 3 Framework para automatização de casos de teste: JUnit
  - Casos de teste
  - Asserções
  - Modelo RIPR no JUnit
- 4 Considerações finais

## Considerações finais. . .

Na aula de hoje nós vimos:

- Automatização
- Framework JUnit;
  - Asserções;
  - RIPR no JUnit.

Na **próxima aula**:

- **Mais sobre automatização de teste de software;**

Ammann, Paul & Jeff Offutt (2016). *Introduction to Software Testing*. 2nd ed. Cambridge University Press, p. 364.

😊 **Próxima aula: exercício(s) sobre o conteúdo da aula de hoje!** 😊

“Box” icon by Mourad Mokrane from the Noun Project (<https://thenounproject.com/>).