Linguagem de Programação I

Aula 10
Introdução ao Desenvolvimento guiado por Testes

l.bertholdo@ifsp.edu.br

Conteúdo

- Testes de Desenvolvimento
  - Testes Unitários
  - Testes Automatizados
  - Test-Driven Development (TDD)
- Exemplo
  - Configuração do JUnit, Classes de Produção e Classe de Teste
  - Anotações
  - Test Data Builder
  - Testes com Exceções
  - Código de Testes antes do Código de Produção
  - Diferença entre Falha e Erro

l.bertholdo@ifsp.edu.br

Testes de Desenvolvimento

- Em geral, os testes de software têm como objetivo demonstrar ao desenvolvedor e ao cliente que o software atende aos **requisitos especificados**.

- Além disso, visam descobrir **defeitos de software**: situações em que o software se comporta de maneira incorreta, indesejável ou de forma diferente das especificações.

- Dentre os testes de software, estão os **testes unitários** – processo de testar individualmente pequenas unidades de um programa, como **classes** de objetos, e seus **métodos**.

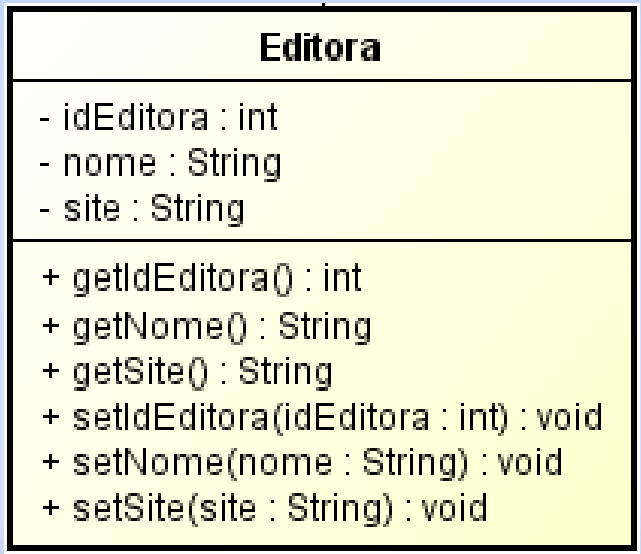
Testes unitários *se concentram em testar as* ***funcionalidades*** *dos objetos e em responder perguntas como: os objetos apresentam o comportamento esperado individualmente?*

l.bertholdo@ifsp.edu.br

Testes Unitários

- Os testes unitários são **chamadas** para os métodos da classe, com diferentes parâmetros de entrada (quando houver).

- Ao testar classes de objeto, sempre que possível, os testes devem ser projetados para:
  - Testar todas as operações (métodos) associadas ao objeto.
  - Definir e verificar o valor de todos os dados (atributos) associados ao objeto.
  - Colocar o objeto em todos os estados possíveis, o que significa simular todos os eventos que causam mudanças de estado.



Testes Automatizados

- Em geral, testes unitários feitos de forma **manual** são caros, pois consomem muito tempo. Por isso, sempre que possível, deve-se automatizá-los.

- Nos **testes automatizados**, os testes são codificados em um programa que é executado para testar o sistema em desenvolvimento.

- Essa forma é mais rápida que o teste manual, especialmente quando envolve **testes de regressão** (reexecução de testes anteriores para verificar se as alterações no software não introduziram novos erros).

Testes Automatizados

- Uma discussão muito comum sobre testes manuais versus testes automatizados é sobre a questão da **produtividade**.
- Ao implementar testes automatizados, a equipe de desenvolvimento terá mais trabalho: escrever **código de teste** e não apenas **código de produção**, tornando-a aparentemente menos produtiva.
- No entanto, a produtividade não pode se resumir ao número de **linhas de código de produção** escritas por dia. Ela deve considerar a quantidade de **linhas de código sem defeitos** escritas por dia.

*Se observarmos o dia a dia de um desenvolvedor que faz **testes manuais**, veremos que ele escreve um pouco, roda o programa, e o programa falha. Ele então corrige o problema e executa novamente o mesmo teste. **Quantas vezes por dia ele executa o mesmo teste manual**? Ao automatizar seus testes, o desenvolvedor gasta tempo apenas **uma vez** (para escrevê-los). Para executá-los posteriormente, ele simplesmente aperta um botão e a máquina realiza o trabalho, **de forma correta e eficiente**.*

Test-Driven Development (TDD)

- O TDD (desenvolvimento guiado por testes), é um método de desen-volvimento de software em que se intercalam **testes** e **codificação**.
- O código de produção é escrito de forma incremental, sempre em conjunto com um **teste automatizado** para esse incremento.
- Para implementar o **TDD**, são essenciais ambientes de testes automatizados, como o **JUnit**, um *framework open-source* que suporta testes de programas Java.

l.bertholdo@ifsp.edu.br

Test-Driven Development (TDD)

- Em ambientes com o **JUnit**, os testes são embutidos em um programa à parte, que os executa e invoca o código em desenvolvimento, possibilitando executar centenas de testes em poucos segundos.

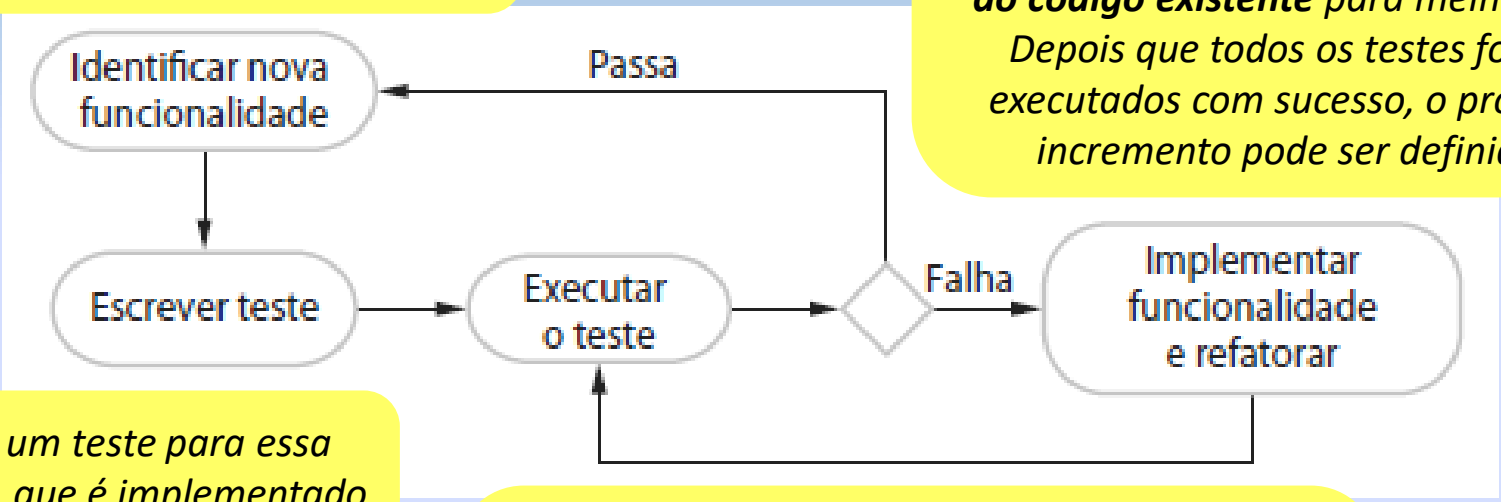
- Embora o **TDD** seja comumente associado aos **métodos ágeis**, ele pode ser usado em qualquer processo de desenvolvimento.

Uma grande vantagem do TDD é que ele ajuda os programadores a clarear suas ideias sobre o que um trecho de código deve fazer. Isso acontece porque, para escrever um teste, é preciso entender a que ele se destina. Esse entendimento torna mais fácil escrever o código da funcionalidade.

Test-Driven Development (TDD)

- Processo básico do TDD

1) *Começa-se identificando o* ***incremento de funcionalidade****, que deve ser pequeno e implementável em poucas linhas de código.*



2) *Escreve-se um teste para essa funcionalidade, que é implementado como um* ***teste automatizado****.*

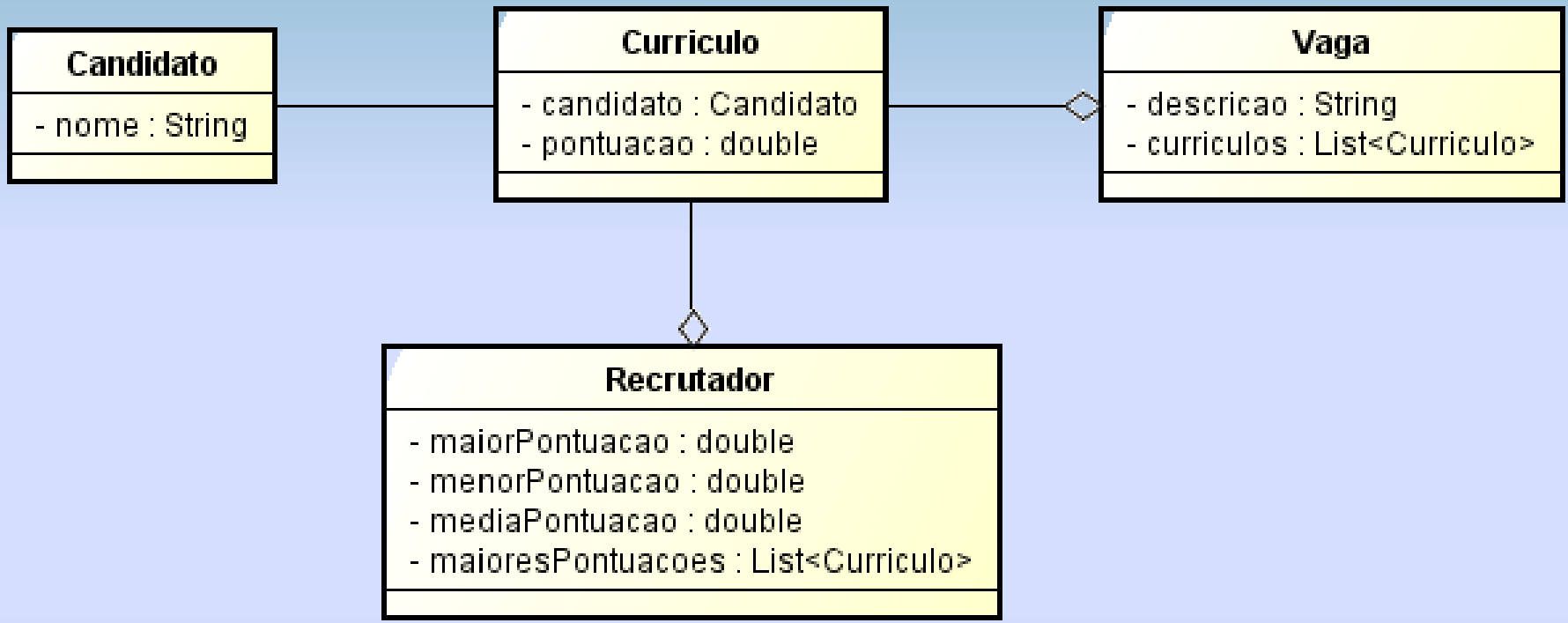
3) *Executa-se o teste, junto com todos os outros testes implementados. Inicialmente, a funcionalidade não estará implementada, logo,* ***o teste falhará****. Isso é proposital, pois mostra que o teste acrescenta algo ao conjunto de testes.*

4) *Implementa-se a funcionalidade e executa-se novamente o teste. A implementação pode envolver a* ***adição de um novo código*** *ou a* ***refatoração do código existente*** *para melhorá-lo. Depois que todos os testes forem executados com sucesso, o próximo incremento pode ser definido.*

l.bertholdo@ifsp.edu.br

Exemplo

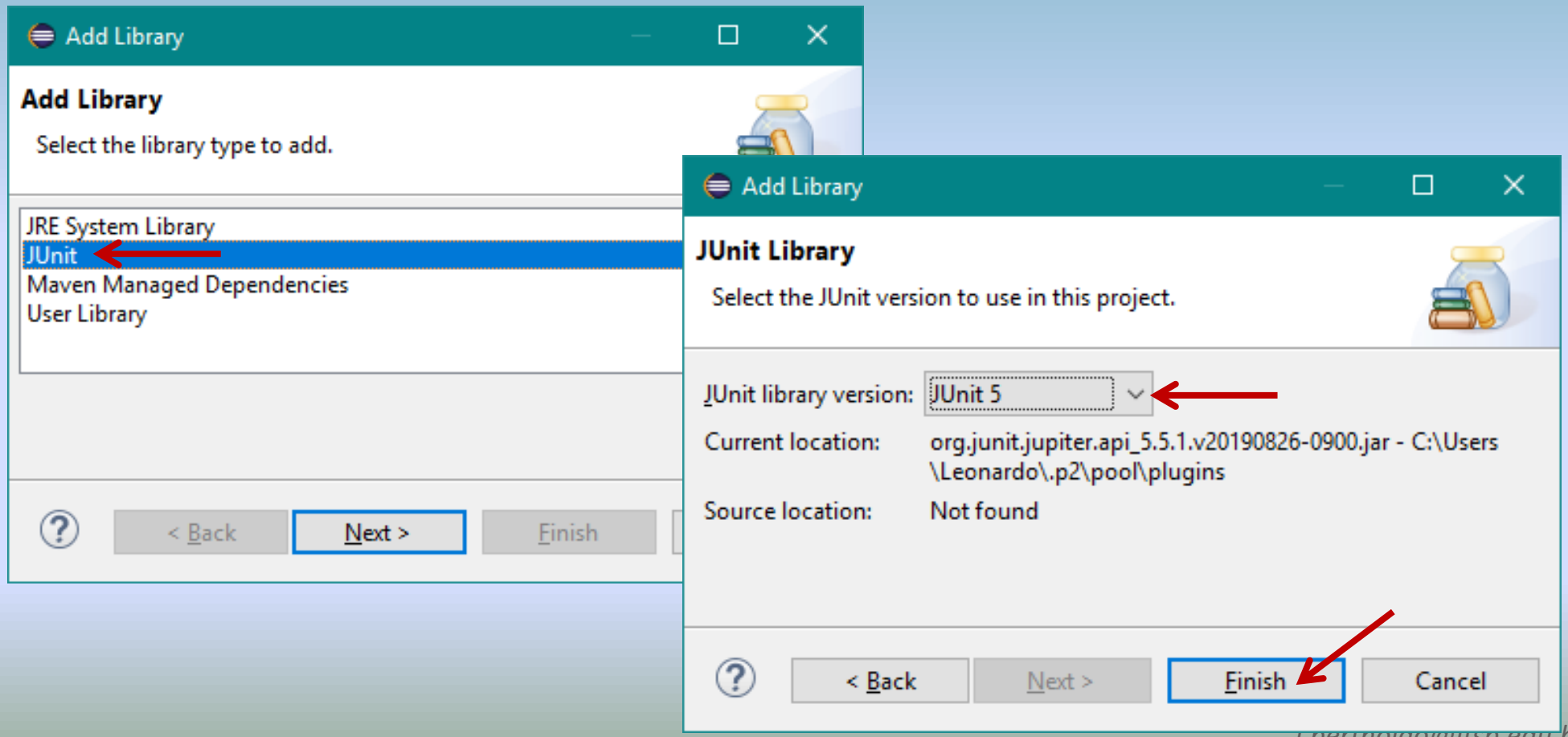
- Agência de Empregos



Configuração do JUnit

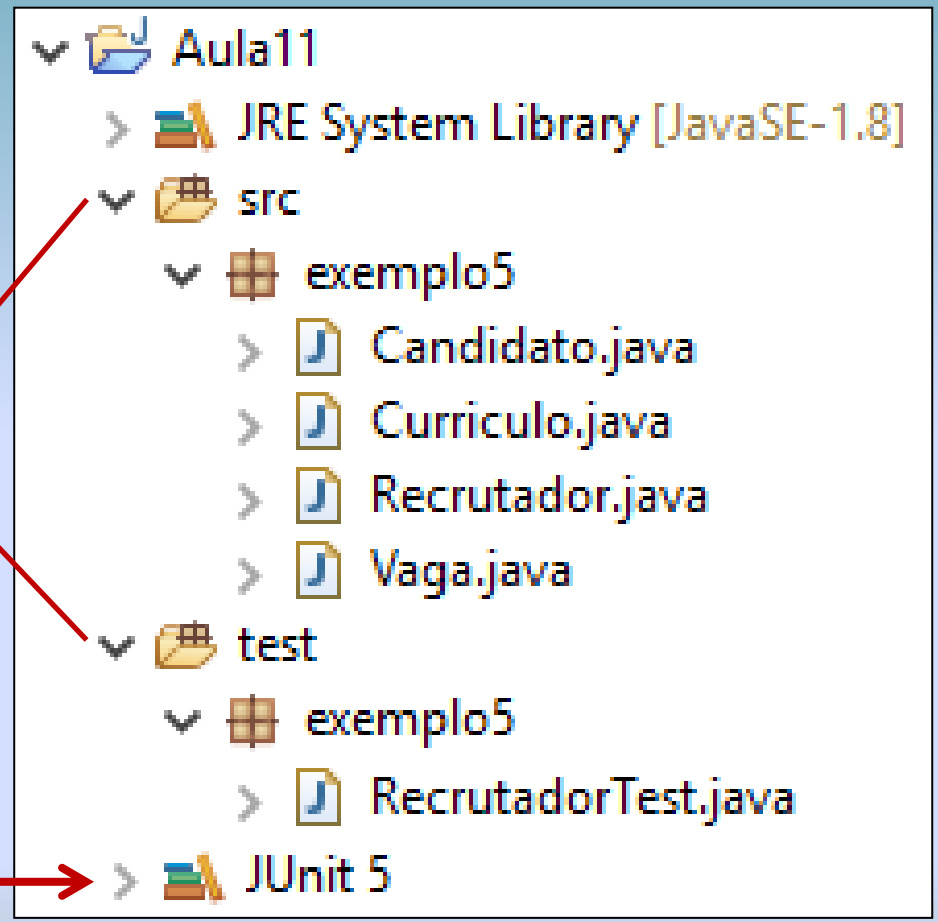
- Para usar o **JUnit** no **Eclipse**, é preciso adicioná-lo ao projeto.

Clique com o botão direito sobre o projeto >> **Build Path** >> **Add Libraries**.



Configuração do JUnit

Note que as **classes de produção** (código em desenvolvimento) e as **classes de teste** (código de teste) são colocadas em **Source Foulders** separadas.



JUnit incluído no projeto

Classes de Produção

*Verificar **imports** das classes no último slide!*

```java
public class Candidato {
    private String nome;

    public Candidato(String nome) { this.nome = nome; }

    public String getNome() { return nome; }
}
```

```java
public class Curriculo {
    private Candidato candidato;
    private double pontuacao;

    public Curriculo(Candidato candidato, double pontuacao) {
        this.candidato = candidato;
        this.pontuacao = pontuacao;
    }

    public Candidato getCandidato() { return candidato; }

    public double getPontuacao() { return pontuacao; }
}
```

Classes de Produção

List é uma interface que define o comportamento das listas da API Java. **ArrayList** é uma das implementações desta interface.

```
public class Vaga {
    private String descricao;
    private List<Curriculo> curriculos;

    public Vaga(String descricao) {
        this.descricao = descricao;
        this.curriculos = new ArrayList<Curriculo>();
    }

    public void recebe(Curriculo curriculo) { curriculos.add(curriculo); }

    public String getDescricao() { return descricao; }

    public List<Curriculo> getCurriculos() { return curriculos; }
}
```

*O ideal é sempre declarar listas como **List** e deixar para definir a implementação apenas no momento da instanciação. Assim, se for preciso alterar o tipo de implementação, bastará instanciar este novo tipo.*
***Por exemplo:** this.curriculos = new LinkedList<Curriculo>().*

l.bertholdo@ifsp.edu.br

Classes de Produção

```
public class Recrutador {
    private double maiorPontuacao = Double.NEGATIVE_INFINITY; // Negativo "infinito".
    private double menorPontuacao = Double.POSITIVE_INFINITY; // Positivo "infinito".
    private double mediaPontuacao = 0;
    private List<Curriculo> maioresPontuacoes;

    public void avaliaCurriculos(Vaga vaga) {
        double total = 0;

        for (Curriculo curriculo : vaga.getCurriculos()) {
            if (curriculo.getPontuacao() > maiorPontuacao)
                maiorPontuacao = curriculo.getPontuacao();
            if (curriculo.getPontuacao() < menorPontuacao)
                menorPontuacao = curriculo.getPontuacao();
            total += curriculo.getPontuacao();
        }
        if (total == 0)
            return;

        mediaPontuacao = total / vaga.getCurriculos().size();

        verificaTresMaioresPontuacoes(vaga);
    }
```

Como **maiorPontuacao** tem o valor "negativo infinito", qualquer pontuação será a **maior** no 1º laço do **for**. Da mesma forma, como **menorPontuacao** tem o valor "positivo infinito", qualquer pontuação será a **menor** no 1º laço.

Se **total** for zero, significa que não há nenhum currículo com pontuação para a vaga. Logo, não é preciso calcular a média de pontuação, nem verificar as três maiores pontuações.

Continua ->

Classes de Produção

new Comparator *cria uma classe anônima (sem nome) que implementa o método* ***compare*** *da interface* ***Comparator****.*

Continuação

```
    public double getMaiorPontuacao() { return maiorPontuacao; }

    public double getMenorPontuacao() { return menorPontuacao; }

    public double getMediaPontuacao() { return mediaPontuacao; }

    public List<Curriculo> getTresMaioresPontuacoes() { return maioresPontuacoes; }

    private void verificaTresMaioresPontuacoes(Vaga vaga) {
        maioresPontuacoes = new ArrayList<Curriculo>(vaga.getCurriculos());

        maioresPontuacoes.sort(new Comparator<Curriculo>() {
            public int compare(Curriculo c1, Curriculo c2) {
                if (c1.getPontuacao() < c2.getPontuacao())
                    return 1;
                if (c1.getPontuacao() > c2.getPontuacao())
                    return -1;
                return 0;
            }
        });
        maioresPontuacoes = maioresPontuacoes.subList(0,
                maioresPontuacoes.size() > 3 ? 3 : maioresPontuacoes.size());
    }
}
```

O método **sort** recebe uma instância de uma classe anônima que implementa a interface **Comparator**, e ordena (de forma decrescente) os elementos da lista, com base da pontuação do currículo.

O método **subList** retorna um intervalo de elementos da lista com base em uma posição inicial (1º argumento) e uma posição final (2º argumento).

Operador ternário (sintaxe): *condição* ***?*** *se verdadeiro* ***:*** *se falso*

Aqui, se o tamanho da lista é maior que 3, a posição final é 3. Senão (:), a posição final é o tamanho da lista.

Classe de Teste

A anotação **@Test** indica que o método em seguida é um método de teste e deve ser executado pelo **JUnit**. Com isso, a classe não precisa ter um método **main** para ser executada.

```
public class RecrutadorTest {
    @Test
    public void deveAvaliarPontuacoesEmOrdemCrescente() {
        Candidato joao = new Candidato("João");
        Candidato maria = new Candidato("Maria");
        Candidato jose = new Candidato("José");

        Vaga vaga = new Vaga("Analista de Sistemas");
        vaga.recebe(new Curriculo(maria, 9));
        vaga.recebe(new Curriculo(joao, 8));
        vaga.recebe(new Curriculo(jose, 6));

        Recrutador recrutador = new Recrutador();
        recrutador.avaliaCurriculos(vaga);

        assertEquals(9, recrutador.getMaiorPontuacao(), 0.001);
        assertEquals(6, recrutador.getMenorPontuacao(), 0.001);
    }
```

Cenário: Aqui, é definido o conjunto de dados a ser usado no teste. No caso, uma vaga com três currículos de três candidatos.

Ação: Aqui, ocorre a execução do trecho de código a ser testado.

Validação: assertEquals é um método estático da classe **Assert**, presente na biblioteca **JUnit**. O **1º** parâmetro é o valor numérico esperado; o **2º** é o valor retornado; e o **3º** é a diferença máxima entre os valores esperado e real, para a qual os dois valores ainda são considerados iguais.

*Uma **anotação (annotation)** é uma metainformação incluída no código do programa, com o objetivo de fornecer diretivas para compiladores, IDEs ou mesmo outros programas que irão manipulá-lo. As anotações são de extrema utilidade, pois permitem reduzir o código escrito e otimizar várias tarefas repetitivas presentes no desenvolvimento. Em Java, toda anotação é iniciada com "@".*

Classe de Teste

```java
@Test
public void deveAvaliarPontuacoesEmOrdemDecrescente() {
    Candidato joao = new Candidato("João");
    Candidato maria = new Candidato("Maria");
    Candidato jose = new Candidato("José");
    Candidato ana = new Candidato("Ana");

    Vaga vaga = new Vaga("Analista de Sistemas");
    vaga.recebe(new Curriculo(joao, 9));
    vaga.recebe(new Curriculo(maria, 8));
    vaga.recebe(new Curriculo(jose, 7));
    vaga.recebe(new Curriculo(ana, 5));

    Recrutador recrutador = new Recrutador();
    recrutador.avaliaCurriculos(vaga);

    assertEquals(9, recrutador.getMaiorPontuacao(), 0.001);
    assertEquals(5, recrutador.getMenorPontuacao(), 0.001);
}
```

Classe de Teste

```java
@Test
public void deveAvaliarPontuacoesEmOrdemAleatoria() {
    Candidato joao = new Candidato("João");
    Candidato maria = new Candidato("Maria");
    Candidato jose = new Candidato("José");
    Candidato ana = new Candidato("Ana");
    Candidato pedro = new Candidato("Pedro");

    Vaga vaga = new Vaga("Analista de Sistemas");
    vaga.recebe(new Curriculo(joao, 10));
    vaga.recebe(new Curriculo(maria, 7));
    vaga.recebe(new Curriculo(jose, 6));
    vaga.recebe(new Curriculo(ana, 10));
    vaga.recebe(new Curriculo(pedro, 8));

    Recrutador recrutador = new Recrutador();
    recrutador.avaliaCurriculos(vaga);

    assertEquals(10, recrutador.getMaiorPontuacao(), 0.001);
    assertEquals(6, recrutador.getMenorPontuacao(), 0.001);
}
```

Classe de Teste

```java
@Test
public void deveCalcularMediaDePontuacoes() {
    Candidato joao = new Candidato("João");
    Candidato maria = new Candidato("Maria");
    Candidato jose = new Candidato("José");

    Vaga vaga = new Vaga("Analista de Sistemas");
    vaga.recebe(new Curriculo(maria, 9));
    vaga.recebe(new Curriculo(joao, 8));
    vaga.recebe(new Curriculo(jose, 6));

    Recrutador recrutador = new Recrutador();
    recrutador.avaliaCurriculos(vaga);

    assertEquals(7.666, recrutador.getMediaPontuacao(), 0.001);
}
```

```java
@Test
public void deveRetornarMediaZeroParaVagaSemCurriculos(){
    Vaga vaga = new Vaga("Analista de Sistemas");

    Recrutador recrutador = new Recrutador();
    recrutador.avaliaCurriculos(vaga);

    assertEquals(0, recrutador.getMediaPontuacao(), 0.001);
}
```

Classe de Teste

```java
@Test
public void deveRetornarAsTresMaioresPontuacoes() {
    Candidato joao = new Candidato("João");
    Candidato maria = new Candidato("Maria");
    Candidato jose = new Candidato("José");
    Candidato ana = new Candidato("Ana");

    Vaga vaga = new Vaga("Analista de Sistemas");
    vaga.recebe(new Curriculo(joao, 5));
    vaga.recebe(new Curriculo(maria, 6));
    vaga.recebe(new Curriculo(jose, 8));
    vaga.recebe(new Curriculo(ana, 10));

    Recrutador recrutador = new Recrutador();
    recrutador.avaliaCurriculos(vaga);

    List<Curriculo> maiores = recrutador.getTresMaioresPontuacoes();

    assertEquals(3, maiores.size());
    assertEquals(10, maiores.get(0).getPontuacao(), 0.001);
    assertEquals(8, maiores.get(1).getPontuacao(), 0.001);
    assertEquals(6, maiores.get(2).getPontuacao(), 0.001);
}
```

Classe de Teste

```java
@Test
public void deveRetornarAsTresMaioresPontuacoesEmVagaComMenosDeTresCurriculos() {
    Candidato joao = new Candidato("João");
    Candidato maria = new Candidato("Maria");

    Vaga vaga = new Vaga("Analista de Sistemas");
    vaga.recebe(new Curriculo(joao, 9));
    vaga.recebe(new Curriculo(maria, 8));

    Recrutador recrutador = new Recrutador();
    recrutador.avaliaCurriculos(vaga);

    List<Curriculo> maiores = recrutador.getTresMaioresPontuacoes();

    assertEquals(2, maiores.size());
    assertEquals(9, maiores.get(0).getPontuacao(), 0.001);
    assertEquals(8, maiores.get(1).getPontuacao(), 0.001);
}
```

Classe de Teste

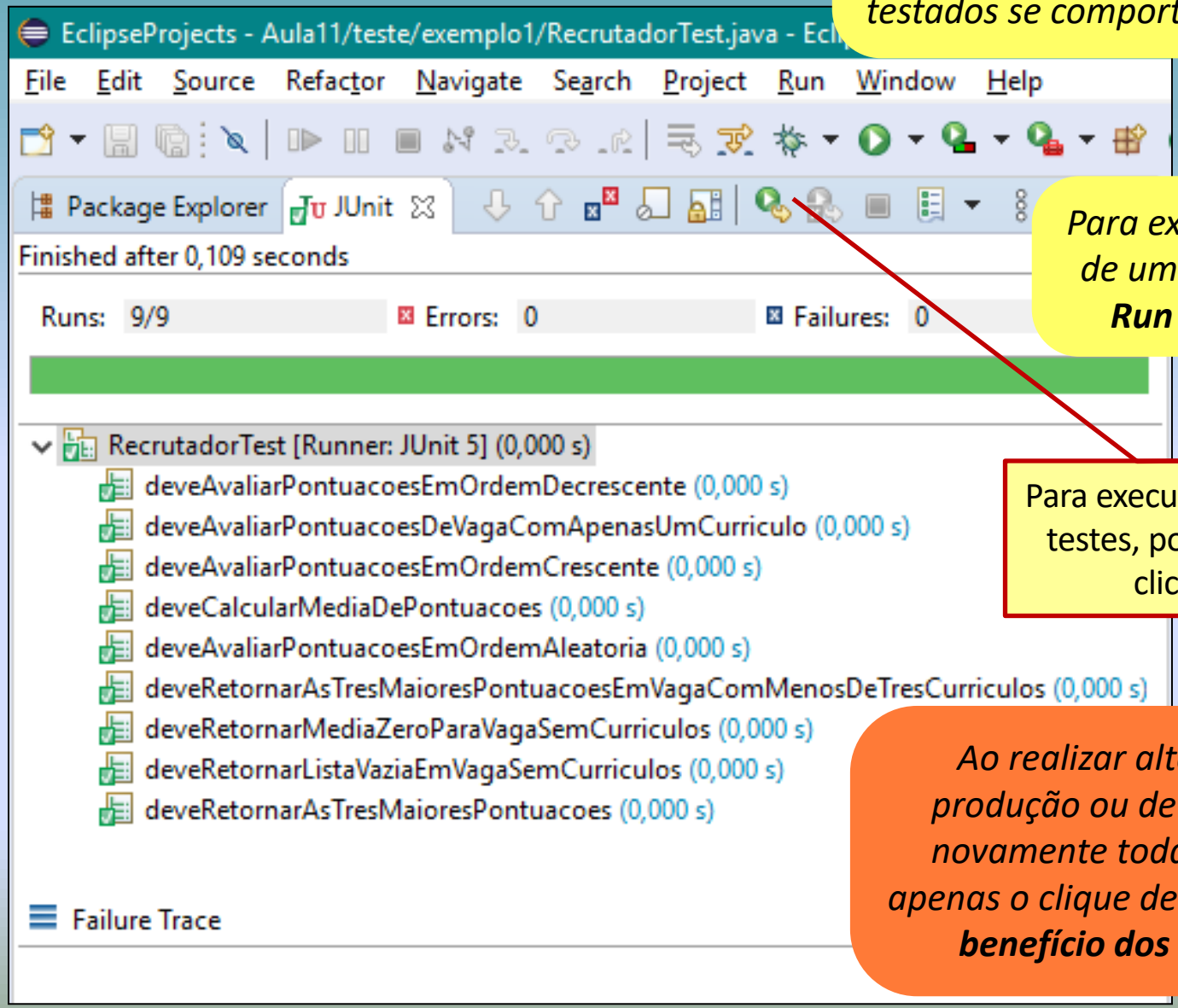
```java
@Test
public void deveAvaliarPontuacoesDeVagaComApenasUmCurriculo() {
    Candidato joao = new Candidato("João");

    Vaga vaga = new Vaga("Analista de Sistemas");
    vaga.recebe(new Curriculo(joao, 8));

    Recrutador recrutador = new Recrutador();
    recrutador.avaliaCurriculos(vaga);

    List<Curriculo> maiores = recrutador.getTresMaioresPontuacoes();

    assertEquals(1, maiores.size());
    assertEquals(8, recrutador.getMaiorPontuacao(), 0.001);
    assertEquals(8, recrutador.getMenorPontuacao(), 0.001);
}
```

```java
    @Test
    public void deveRetornarListaVaziaEmVagaSemCurriculos() {
        Vaga vaga = new Vaga("Analista de Sistemas");

        Recrutador recrutador = new Recrutador();
        recrutador.avaliaCurriculos(vaga);

        List<Curriculo> maiores = recrutador.getTresMaioresPontuacoes();

        assertEquals(0, maiores == null ? 0 : maiores.size());
        assertEquals(null, maiores); // Outra forma de validação.
    }
} // Fim da classe RecrutadorTest
```

Operador ternário: Se a lista é nula (vazia), o resultado é **0**. Senão, o resultado é o **tamanho da lista**.

Execução do Teste

Os ícones verdes, ao lado dos métodos, indicam que os testes "passaram", ou seja, os códigos testados se comportaram conforme o esperado.



*Para executar os métodos de teste de uma classe, selecione o menu **Run >> Run As >> JUnit Test**.*

Para executar novamente, uma bateria de testes, pode-se também simplesmente clicar no botão **Rerun Test**.

*Ao realizar alterações nos códigos de produção ou de testes, pode-se executar novamente toda a bateria de testes com apenas o clique de um botão. **Esse é o grande benefício dos testes automatizados!***

Anotações

- Testes automatizados nada mais são do que códigos. E códigos mal escritos são mais passíveis de erros e difíceis de serem mantidos.

```java
public class RecrutadorTest {
    @Test
    public void deveAvaliarPontuacoesEmOrdemCrescente() {
        Candidato joao = new Candidato("João");
        Candidato maria = new Candidato("Maria");
        Candidato jose = new Candidato("José");

        Vaga vaga = new Vaga("Analista de Sistemas");
        vaga.recebe(new Curriculo(maria, 9));
        vaga.recebe(new Curriculo(joao, 8));
        vaga.recebe(new Curriculo(jose, 6));

        Recrutador recrutador = new Recrutador();
        recrutador.avaliaCurriculos(vaga);

        assertEquals(9, recrutador.getMaiorPontuacao(), 0.001);
        assertEquals(6, recrutador.getMenorPontuacao(), 0.001);
    }
```

*Todos os nove métodos de teste da classe **RecrutadorTest** instanciam as classes **Candidato**, **Vaga** e **Recrutador**. Seria interessante que isso fosse feito uma **única vez**, para todos os métodos.*

Códigos de teste devem ser refatorados constantemente.
Testes mal escritos podem deixar de ajudar e começar a atrapalhar.

Anotações

- A fim de evitar a repetição de códigos, o **JUnit** possui anotações para indicar o momento em que um método deve ser executado:
  - **@Before:** Indica que o método "anotado" deve ser executado uma vez antes de cada método de teste.
  - **@After:** Indica que o método "anotado" deve ser executado uma vez depois de cada método de teste.
  - **@BeforeClass:** Indica que o método "anotado" deve ser executado uma única vez antes de todos os métodos de teste da classe.
  - **@AfterClass:** Indica que o método "anotado" deve ser executado uma única vez depois de todos os métodos de teste da classe.

*As anotações **@After** e **@AfterClass** são usadas quando os testes acessam recursos que precisam ser finalizados, como conexões com bancos de dados, fluxos de arquivos etc. Estes testes não são considerados **testes unitários**, pois precisam se integrar com outros sistemas. No entanto, desenvolvedores também utilizam o **JUnit** para escrever **testes de integração**.*

Anotações

```java
public class RecrutadorTest {
    private Recrutador recrutador;
    private Candidato joao, maria, jose, ana, pedro;
    private Vaga vaga;

    @Before // Indica que este método será executado uma vez antes de cada método de teste.
    public void executaAntesDeCadaMetodo() {
        this.recrutador = new Recrutador();

        this.joao = new Candidato("João");
        this.maria = new Candidato("Maria");
        this.jose = new Candidato("José");
        this.ana = new Candidato("Ana");
        this.pedro = new Candidato("Pedro");

        this.vaga = new Vaga("Analista de Sistemas");

        System.out.println("Início do teste");
    }

    @After // Indica que este método será executado uma vez depois de cada método de teste.
    public void executaDepoisDeCadaMetodo() { System.out.println("Fim do teste"); }

    @BeforeClass // Indica que este método será executado uma única vez
                 // antes de todos os métodos de teste.
    public static void executaAntesDaClasse() { System.out.println("Antes da classe"); }

    @AfterClass // Indica que este método será executado uma única vez
                // depois de todos os métodos de teste.
    public static void executaDepoisDaClasse() { System.out.println("Depois da classe"); }
```

Os métodos precedidos pelas anotações devem ser **públicos**, para que o **JUnit** possa acessá-los.

Os métodos "anotados" com **@Before** e **@After** serão executados nove vezes, uma vez para cada método.

Console

<terminated> Recruta

```
Antes da classe
Início do teste
Fim do teste
Início do teste
Fim do teste
Início do teste
Fim do teste
Início do teste
Fim do teste
Início do teste
Fim do teste
Início do teste
Fim do teste
Início do teste
Fim do teste
Início do teste
Fim do teste
Início do teste
Fim do teste
Depois da classe
```

Anotações

Método de Teste (Antes)

```
public class RecrutadorTest {
    @Test
    public void deveAvaliarPontuacoesEmOrdemCrescente() {
        Candidato joao = new Candidato("João");
        Candidato maria = new Candidato("Maria");
        Candidato jose = new Candidato("José");

        Vaga vaga = new Vaga("Analista de Sistemas");
        vaga.recebe(new Curriculo(maria, 9));
        vaga.recebe(new Curriculo(joao, 8));
        vaga.recebe(new Curriculo(jose, 6));

        Recrutador recrutador = new Recrutador();
        recrutador.avaliaCurriculos(vaga);

        assertEquals(9, recrutador.getMaiorPontuacao(), 0.001);
        assertEquals(6, recrutador.getMenorPontuacao(), 0.001);
    }
```

Método de Teste (Agora)

```
@Test
public void deveAvaliarPontuacoesEmOrdemCrescente() {
    vaga.recebe(new Curriculo(maria, 9));
    vaga.recebe(new Curriculo(joao, 8));
    vaga.recebe(new Curriculo(jose, 6));

    recrutador.avaliaCurriculos(vaga);

    assertEquals(9, recrutador.getMaiorPontuacao(), 0.001);
    assertEquals(6, recrutador.getMenorPontuacao(), 0.001);
}
```

Test Data Builder

- Durante a escrita dos testes, é comum termos que criar **instâncias complexas**, que envolvem várias classes de produção e são usadas por diversas classes de teste.

*Uma **vaga** é composta por **currículos**, que por sua vez são compostos por **candidatos**.*

- Nesses casos, o ideal é isolar o código de criação dessas instâncias em um único lugar, para que mudanças na estrutura das classes de produção não impactem nos métodos de teste.
- Classes como essas são conhecidas como **Test Data Builders** e consistem em um padrão de projeto para código de testes.

Test Data Builder

*Criar uma instância de **Vaga** não é tão simples. Além de instanciar a classe e invocar seu método **recebe**, é preciso criar instâncias de **Curriculo**.*

*Todas as classes de teste que criam vagas deverão ter esse trabalho. Além disso, se houver uma alteração no método **recebe** ou no construtor da classe **Curriculo**, vários trechos de código de teste precisarão ser alterados.*

Uma boa ideia é isolar o código de criação de vagas em uma classe específica.

```
public class RecrutadorTest {
    private Recrutador recrutador;
    private Candidato joao, maria, jose, ana, pedro;
    private Vaga vaga;

    @Before
    public void executaAntesDeCadaMetodo() {
        this.recrutador = new Recrutador();

        this.joao = new Candidato("João");
        this.maria = new Candidato("Maria");
        this.jose = new Candidato("José");
        this.ana = new Candidato("Ana");
        this.pedro = new Candidato("Pedro");

        this.vaga = new Vaga("Analista de Sistemas");

        System.out.println("Início do teste");
    }
```

```
@Test
public void deveAvaliarPontuacoesEmOrdemCrescente() {
    vaga.recebe(new Curriculo(maria, 9));
    vaga.recebe(new Curriculo(joao, 8));
    vaga.recebe(new Curriculo(jose, 6));

    recrutador.avaliaCurriculos(vaga);

    assertEquals(9, recrutador.getMaiorPontuacao(), 0.001);
    assertEquals(6, recrutador.getMenorPontuacao(), 0.001);
}
```

Test Data Builder

Como a classe **CriadorDeVaga** instancia a classe **Vaga**, a classe **RecrutadorTest** não precisa mais fazer este trabalho.

```java
public class CriadorDeVaga {
    private Vaga vaga;

    public CriadorDeVaga para(String descricao) {
        this.vaga = new Vaga(descricao);
        return this;
    }

    public CriadorDeVaga curriculo(Candidato candidato, double pontuacao) {
        vaga.recebe(new Curriculo(candidato, pontuacao));
        return this;
    }

    public Vaga cria() { return vaga; }
}
```

```java
public class RecrutadorTest {
    private Recrutador recrutador;
    private Candidato joao, maria, jose, ana, pedro;
    private Vaga vaga;

    @Before
    public void executaAntesDeCadaMetodo() {
        this.recrutador = new Recrutador();

        this.joao = new Candidato("João");
        this.maria = new Candidato("Maria");
        this.jose = new Candidato("José");
        this.ana = new Candidato("Ana");
        this.pedro = new Candidato("Pedro");

        System.out.println("Início do teste");
    }
```

Test Data Builder

Método Antes

```
@Test
public void deveAvaliarPontuacoesEmOrdemCrescente() {
    vaga.recebe(new Curriculo(maria, 9));
    vaga.recebe(new Curriculo(joao, 8));
    vaga.recebe(new Curriculo(jose, 6));

    recrutador.avaliaCurriculos(vaga);

    assertEquals(9, recrutador.getMaiorPontuacao(), 0.001);
    assertEquals(6, recrutador.getMenorPontuacao(), 0.001);
}
```

A classe **RecrutadorTest** também não precisa mais chamar o método **recebe** e instanciar a classe **Curriculo**.

Método Agora

```
@Test
public void deveAvaliarPontuacoesEmOrdemCrescente() {
    vaga = new CriadorDeVaga().para("Analista de Sistemas")
            .curriculo(maria, 9)
            .curriculo(joao, 8)
            .curriculo(jose, 6)
            .cria();

    recrutador.avaliaCurriculos(vaga);

    assertEquals(9, recrutador.getMaiorPontuacao(), 0.001);
    assertEquals(6, recrutador.getMenorPontuacao(), 0.001);
}
```

Test Data Builder

O método **para** cria uma instância de **Vaga**, contendo a descrição da vaga. Cada método **curriculo** adiciona uma instância de **Curriculo** à instância de **Vaga**, que por sua vez contém uma instância de **Candidato**. Por fim, o método **cria** retorna a instância de **Vaga** criada contendo todas essas informações. Essas chamadas sequenciais de métodos são possíveis, porque os métodos **para** e **currículo** retornam **this**, ou seja, a própria instância **CriadorDeVaga** que chamou estes métodos.

```
@Test
public void deveAvaliarPontuacoesEmOrdemCrescente() {
    vaga = new CriadorDeVaga().para("Analista de Sistemas")
            .curriculo(maria, 9)
            .curriculo(joao, 8)
            .curriculo(jose, 6)
            .cria();
```

```
public class CriadorDeVaga {
    private Vaga vaga;

    public CriadorDeVaga para(String descricao) {
        this.vaga = new Vaga(descricao);
        return this;
    }

    public CriadorDeVaga curriculo(Candidato candidato, double pontuacao) {
        vaga.recebe(new Curriculo(candidato, pontuacao));
        return this;
    }

    public Vaga cria() { return vaga; }
}
```

Agora, se houver uma alteração no método **recebe** ou no construtor da classe **Curriculo**, basta alterar a classe **CriadorDeVaga**, em vez de alterar cada método de teste.

Testes com Exceções

- Durante os testes, também é interessante que o desenvolvedor teste **situações de exceção**.

```
public void avaliaCurriculos(Vaga vaga) {
    double total = 0;

    for (Curriculo curriculo : vaga.getCurriculos()) {
        if (curriculo.getPontuacao() > maiorPontuacao)
            maiorPontuacao = curriculo.getPontuacao();
        if (curriculo.getPontuacao() < menorPontuacao)
            menorPontuacao = curriculo.getPontuacao();
        total += curriculo.getPontuacao();
    }
    if (total == 0)
        return;

    mediaPontuacao = total / vaga.getCurriculos().size();

    verificaTresMaioresPontuacoes(vaga);
}
```

*Quando a **vaga** recebida pelo método **avaliaCurriculos** da classe **Recrutador** não tem nenhum currículo associado, não faz sentido avaliar seus currículos. Nesse caso, podemos então **lançar uma exceção** sempre que a vaga não tiver nenhum currículo.*

Testes com Exceções

Ao chamar o método **avaliaCurriculos**, se a vaga não possuir nenhum currículo associado, lança uma exceção **RuntimeException**.

```
public void avaliaCurriculos(Vaga vaga) {
    if (vaga.getCurriculos().size() == 0)
        throw new RuntimeException("Esta vaga não tem currículos associados!");

    double total = 0;

    for (Curriculo curriculo : vaga.getCurriculos()) {
        if (curriculo.getPontuacao() > maiorPontuacao)
            maiorPontuacao = curriculo.getPontuacao();
        if (curriculo.getPontuacao() < menorPontuacao)
            menorPontuacao = curriculo.getPontuacao();
        total += curriculo.getPontuacao();
    }
    if (total == 0)
        return;
```

*Para executar este teste, é preciso comentar os métodos **deveRetornarMediaZeroParaVagaSemCurriculos** e **deveRetornarListaVaziaEmVagaSemCurriculos**, pois eles não estão configurados para esperar uma exceção **RuntimeException**.*

```
@Test(expected=RuntimeException.class)
public void naoDeveAvaliarVagasSemNenhumCurriculo() {
    vaga = new CriadorDeVaga()
            .para("Analista de Sistemas")
            .cria();

    recrutador.avaliaCurriculos(vaga);
}
```

O atributo **expected** da anotação **@Test** indica que o teste passará apenas se a exceção **RuntimeException** for lançada para esta vaga.

l.bertholdo@ifsp.edu.br

Código de Testes antes do Código de Produção

- Relembrando o processo básico do TDD:

1. Identificar um cenário de teste para uma **funcionalidade** ou **regra de negócio** simples (implementável em poucas linhas de código):
2. Implementar um teste automatizado para essa funcionalidade.
3. Executar o teste e confirmar a falha, visto que a funcionalidade ainda não foi implementada (ou refatorada).
4. Implementar (ou refatorar) a funcionalidade. Em seguida, executar novamente o teste e verificar se ele passou, ou seja, se o resultado esperado pelo **código de teste** é o mesmo retornado pela execução do **código de produção**.

Código de Testes antes do Código de Produção

- **Regra de negócio:** Uma vaga não deve aceitar dois currículos do mesmo candidato.

1º) *Identificar o cenário de teste.*

2º) *Escrever um teste (com cenário, ação e validação) para verificar a regra de negócio.*

3º) *Executar o teste e confirmar a falha.*

```
public class VagaTest {
    private Candidato joao;
    private Vaga vaga;

    @Before
    public void executaAntesDeCadaMetodo() { this.joao = new Candidato("João"); }

    @Test
    public void naoDeveAceitarDoisCurriculosDoMesmoCandidato() {
        vaga = new CriadorDeVaga()
                .para("Analista de Sistemas")
                .curriculo(joao, 8)
                .curriculo(joao, 9)
                .cria();

        assertEquals(1, vaga.getCurriculos().size());
        assertEquals(8, vaga.getCurriculos().get(0).getPontuacao(), 0.001);
    }
}
```

Código de Testes antes do Código de Produção

4º) *Escrever o código da regra de negócio e executar o teste novamente.*

```
public class Vaga {
    private String descricao;
    private List<Curriculo> curriculos;

    public Vaga(String descricao) {
        this.descricao = descricao;
        this.curriculos = new ArrayList<Curriculo>();
    }

    public void recebe(Curriculo curriculo) {
        if (curriculos.isEmpty() || qtdDeCurriculosDoCandidato(curriculo.getCandidato()) == 0)
            curriculos.add(curriculo);
    }

    public String getDescricao() { return descricao; }

    public List<Curriculo> getCurriculos() { return curriculos; }

    private int qtdDeCurriculosDoCandidato(Candidato candidato) {
        int total = 0;

        for (Curriculo l : curriculos)
            if (l.getCandidato().equals(candidato))
                total++;

        return total;
    }
}
```

Se a lista de currículos for vazia, ou se a quantidade de currículos do candidato para a vaga for zero.

Código de Testes antes do Código de Produção

- **Regra de negócio:** Currículos com pontuação menor que 5 e maior que 10 não devem ser aceitos.

1º) *Identificar o cenário de teste.*

2º) *Escrever os testes para verificar a regra de negócio.*

3º) *Executar o teste.*

4º) *Escrever o código da regra de negócio e executar o teste novamente.*

```
public class Curriculo {
    private Candidato candidato;
    private double pontuacao;

    public Curriculo(Candidato candidato, double pontuacao) {
        if (pontuacao < 5 || pontuacao > 10)
            throw new IllegalArgumentException();

        this.candidato = candidato;
        this.pontuacao = pontuacao;
    }

    public Candidato getCandidato() { return candidato; }

    public double getPontuacao() { return pontuacao; }
}
```

```
public class CurriculoTest {
    private Candidato joao;
    private Vaga vaga;

    @Before
    public void executaAntesDeCadaMetodo() { this.joao = new Candidato("João"); }

    @Test(expected=IllegalArgumentException.class)
    public void naoDeveAceitarCurriculosComPontuacaoMenorQueCinco() { new Curriculo(joao, 2); }

    @Test(expected=IllegalArgumentException.class)
    public void naoDeveAceitarCurriculosComPontuacaoMaiorQueDez() { new Curriculo(joao, 11); }
}
```

O teste passa apenas se a exceção **IllegalArgumentException** for lançada.

Diferença entre Falha e Erro

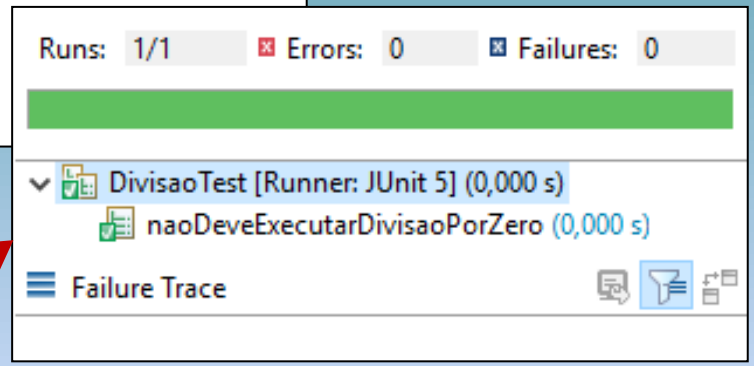
- Existem dois tipos básicos de motivos para um teste não passar:
  - **Falha:** O resultado retornado pela **código de produção** é diferente do resultado esperado pelo **código de teste**.
  - **Erro:** O **código de produção** não pôde retornar um resultado, devido a um erro durante a sua execução.

Diferença entre Falha e Erro

```java
public class Divisao {
    public static double executaDivisao(int dividendo, int divisor) {
        return dividendo / divisor;
    }
}
```

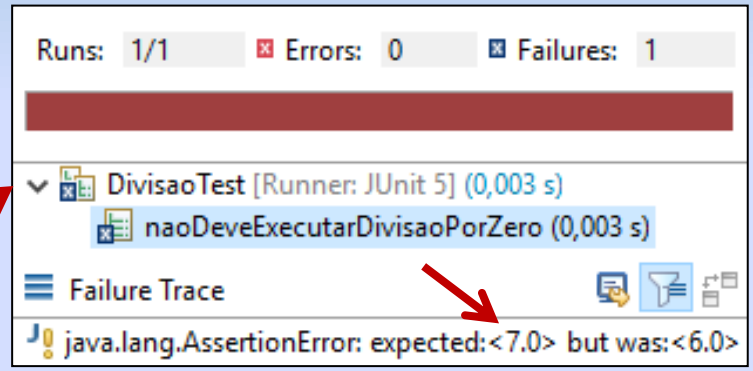
```java
public class DivisaoTest {
    @Test
    public void naoDeveExecutarDivisaoPorZero() {
        double resultado = Divisao.executaDivisao(18, 3);
        assertEquals(6, resultado, 0.001);
    }
}
```

Sucesso



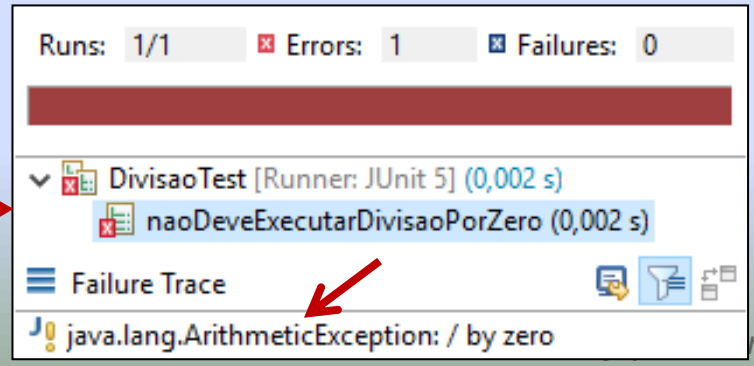

```java
public class DivisaoTest {
    @Test
    public void naoDeveExecutarDivisaoPorZero() {
        double resultado = Divisao.executaDivisao(18, 3);
        assertEquals(7, resultado, 0.001);
    }
}
```

Falha




```java
public class DivisaoTest {
    @Test
    public void naoDeveExecutarDivisaoPorZero() {
        double resultado = Divisao.executaDivisao(18, 0);
        assertEquals(6, resultado, 0.001);
    }
}
```

Erro



Imports das Classes

```java
import java.util.ArrayList;
import java.util.List;

public class Vaga {
```

```java
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class Recrutador {
```

```java
import exemplo5.Candidato;
import exemplo5.Curriculo;
import exemplo5.Vaga;

public class CriadorDeVaga {
```

```java
import static org.junit.Assert.assertEquals;

import java.util.List;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import exemplo5.Candidato;
import exemplo5.CriadorDeVaga;
import exemplo5.Curriculo;
import exemplo5.Recrutador;
import exemplo5.Vaga;

public class RecrutadorTest {
```

```java
import static org.junit.Assert.assertEquals;

import org.junit.Before;
import org.junit.Test;

import exemplo5.Candidato;
import exemplo5.CriadorDeVaga;
import exemplo5.Vaga;

public class VagaTest {
```

```java
import org.junit.Before;
import org.junit.Test;

public class CurriculoTest {
```

* As classes **Candidato** e **Curriculo** não necessitam de **imports**.

l.bertholdo@ifsp.edu.br

Referências

- ANICHE, Mauricio. Test-Driven Development: Teste e Design no Mundo Real. Casa do Código, 2017.
- JANDL JUNIOR, P. Java – Guia do Programador – 3ª Edição. São Paulo: Novatec Editora, 2015.
- PRESSMAN, Roger S.; MAXIM, Bruce R. Engenharia de software: uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016.
- SOMMERVILLE, Ian. Engenharia de software. 9. ed. São Paulo: Pearson Prentice Hall, 2011.