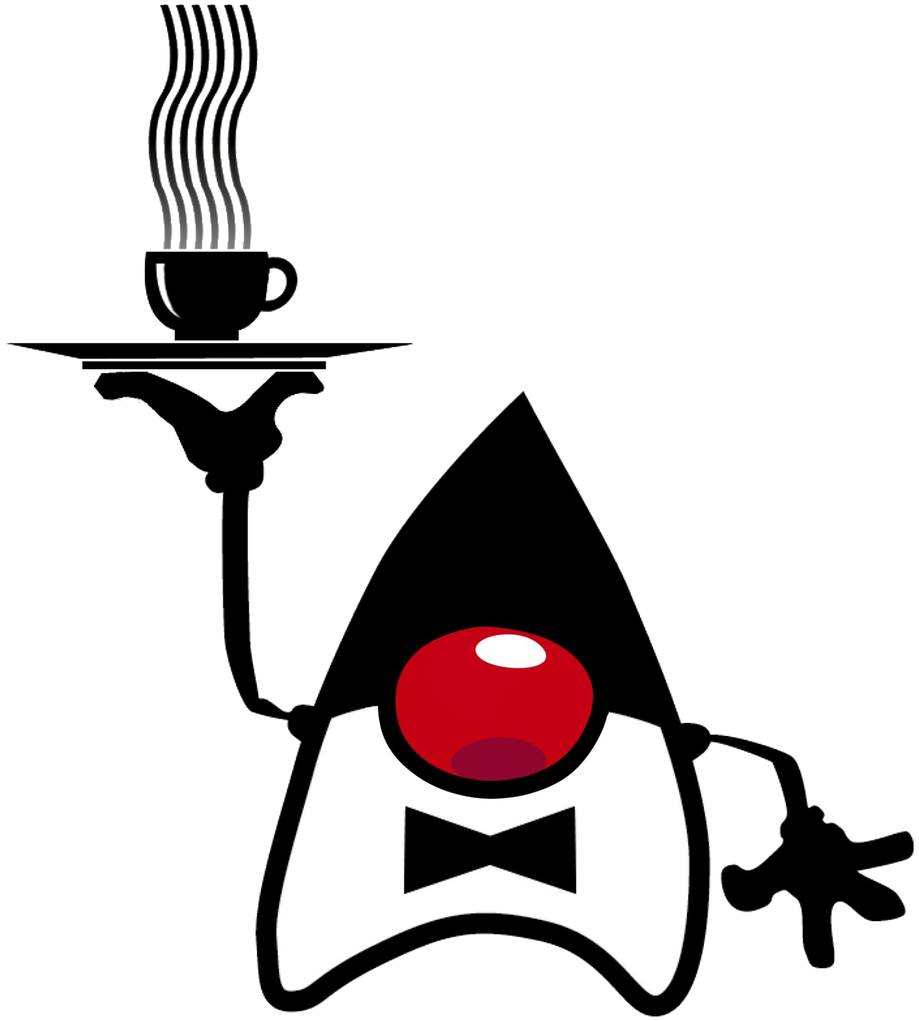
Trilha Java

Encontro 27 – Testes Automatizados



vnt/school
powered by venturus

Recapitulação

1. Interface Funcional
2. Expressões Lambda
3. Predicate
4. Consumer
5. Function
6. Stream

vnt/school
powered by venturus

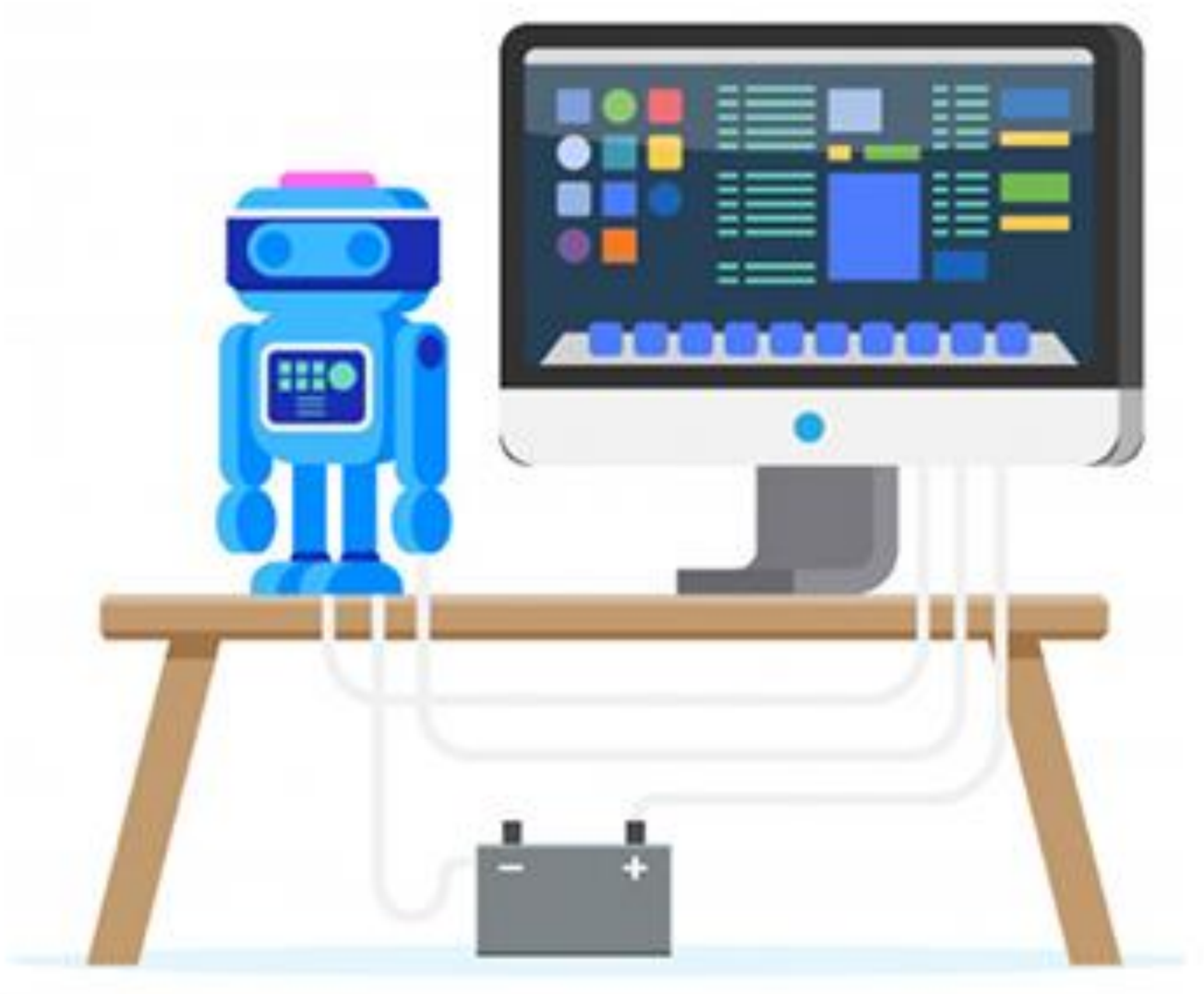
Agenda

1. Testes Automatizados
2. JUnit
3. Test Drive Development
4. Exemplos
5. Exercícios

vnt/school
powered by venturus

Porque Testes?

Validação do código



vnt/school

powered by venturus

Porque Testes?

Em programação **muitos códigos** diferentes são escritos e é preciso **entender** sua **funcionalidade**.

Às vezes é preciso **usar padrões** de projeto específicos, escrever códigos para **fazer validações** e etc.

vnt/school
powered by venturus

Porque Testes?

Em todos esses casos, podem surgir dúvidas:

Será que esse código vai funcionar como esperado?

Ele atende a todas as regras de negócio?

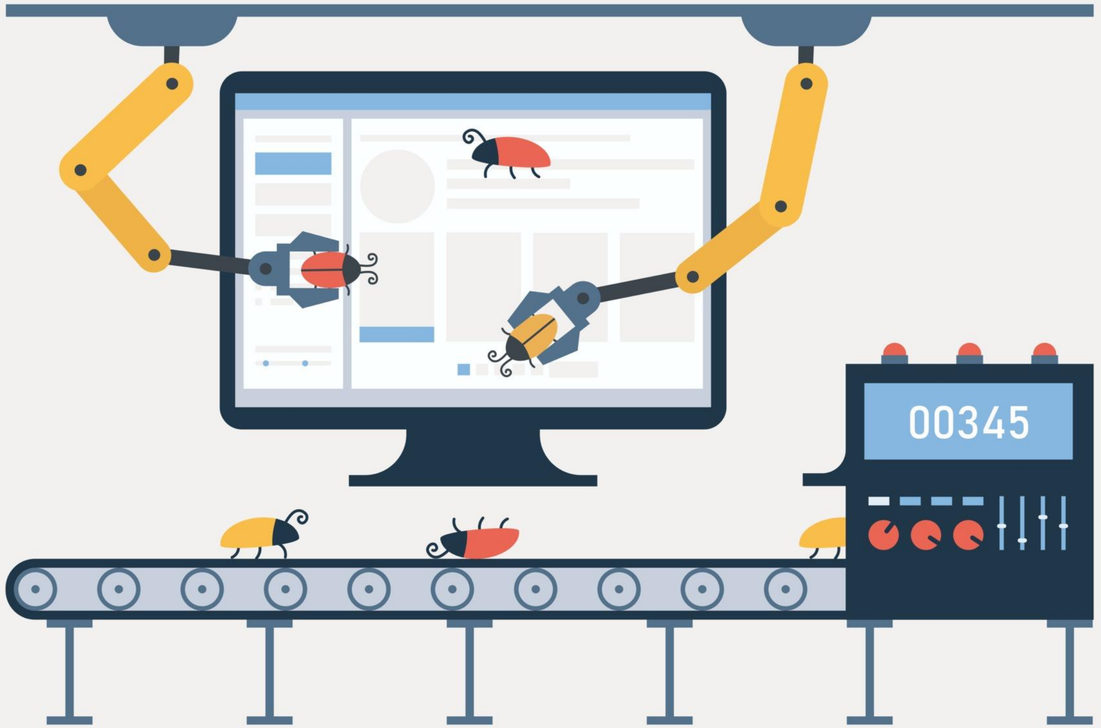
Ele abrange todos os cenários possíveis?

Porque Testes?

É por isso que precisamos sempre **testar** os nossos códigos.

Não é à toa que **existem profissionais** cujo trabalho é dedicado a **testar as funcionalidades** escritas por outras pessoas.

A depender do tamanho da empresa e do número de projetos, pode haver até um **time de testes**.



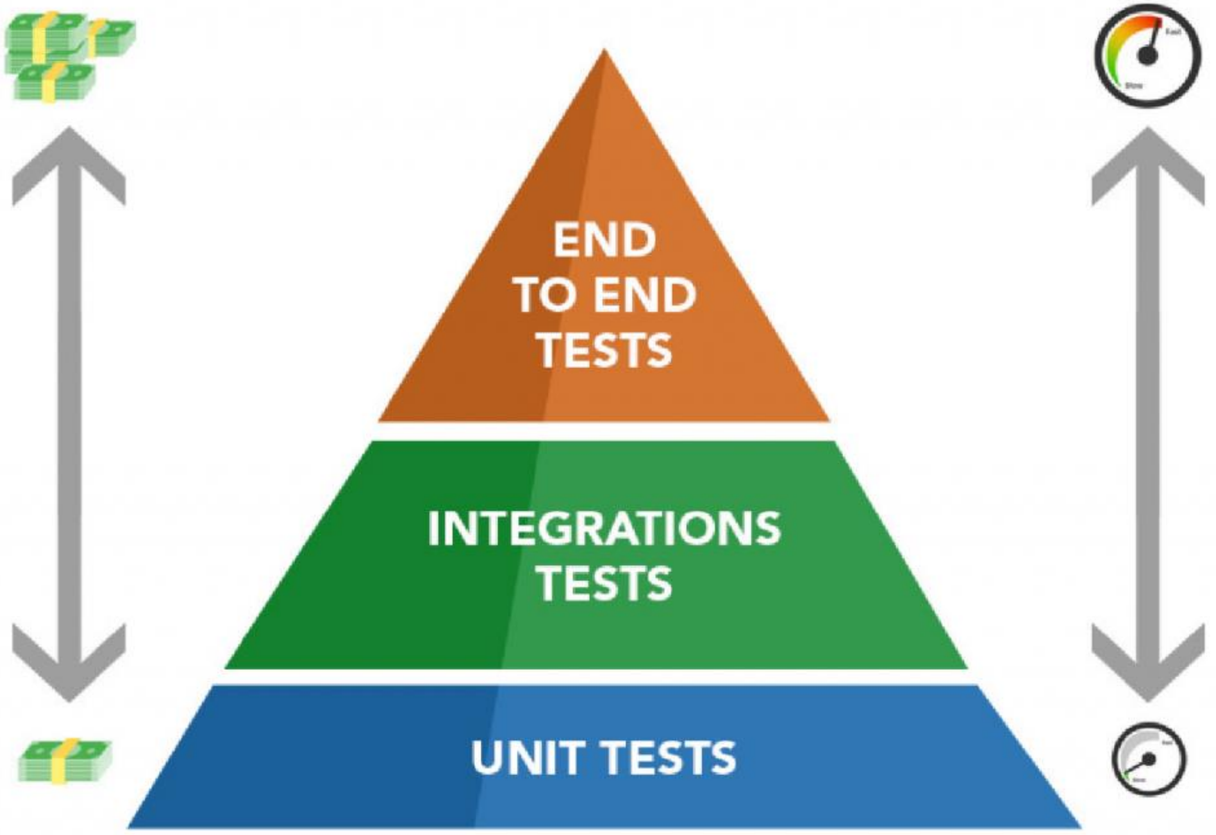
vnt/school
powered by venturus

Porque Testes?

Existem alguns tipos de testes.

Os **testes automatizados**. Ganharam força no mercado porque apresentam grandes **vantagens**:

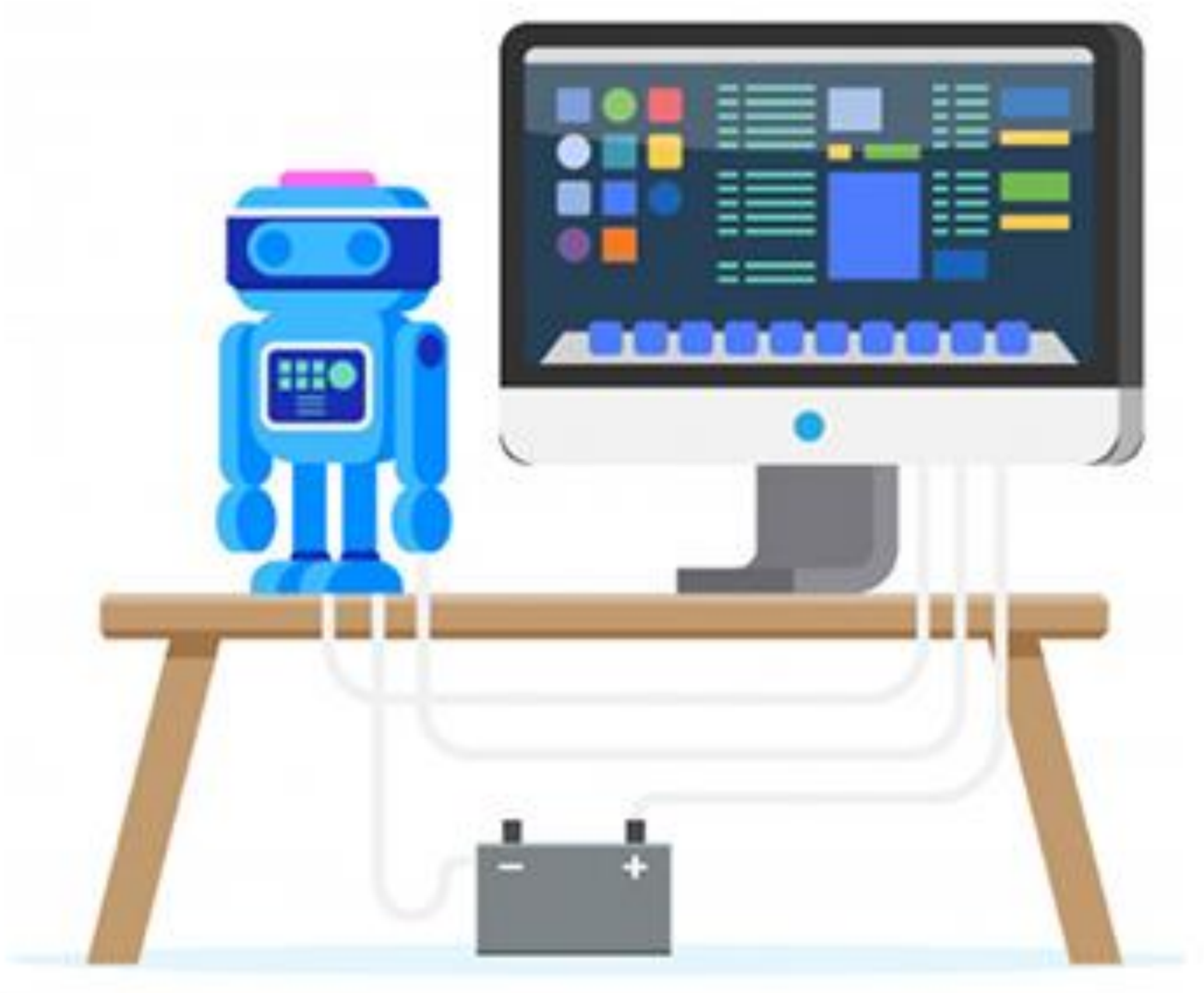
Automatização de ações repetitivas.
Feedback mais rápido.
Segurança para mudar o código.
Favorecimento de melhorias.



vnt/school
powered by venturus

Testes Automatizados

Validação do código



vnt/school

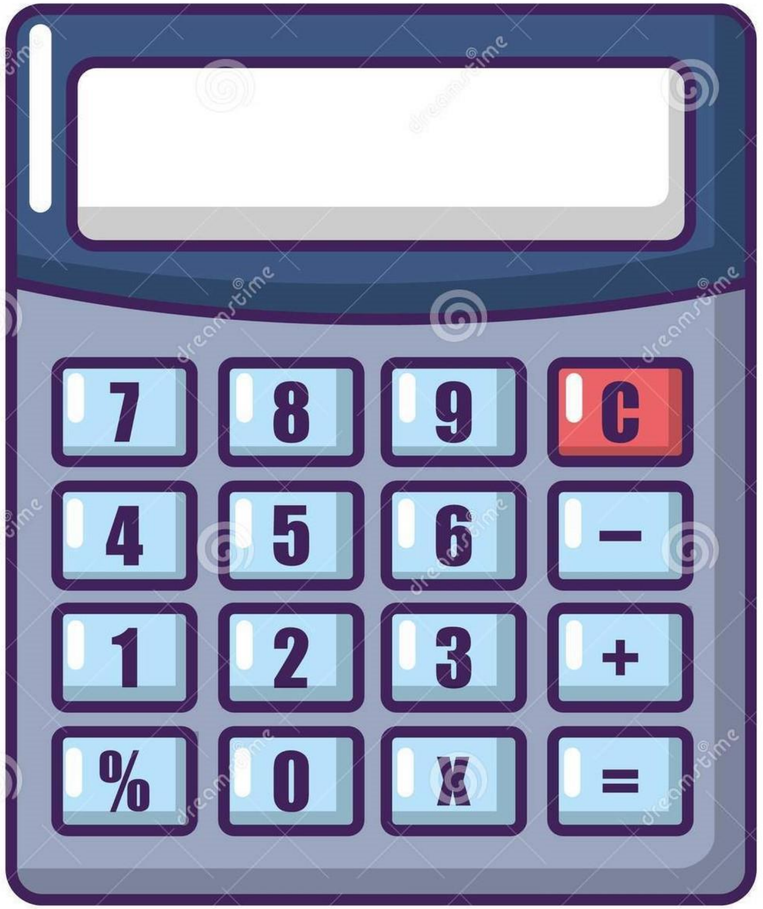
powered by venturus

Testes Automatizados

Crie um código simples simulando uma **calculadora** que realiza operações de soma.

Feito isso, é hora de **testar** a nossa calculadora!

Como seriam os testes?



vnt /school
powered by venturus

Testes Automatizados

Um **teste automatizado** nada mais é do que **uma classe**, um código cujo objetivo é testar outro.

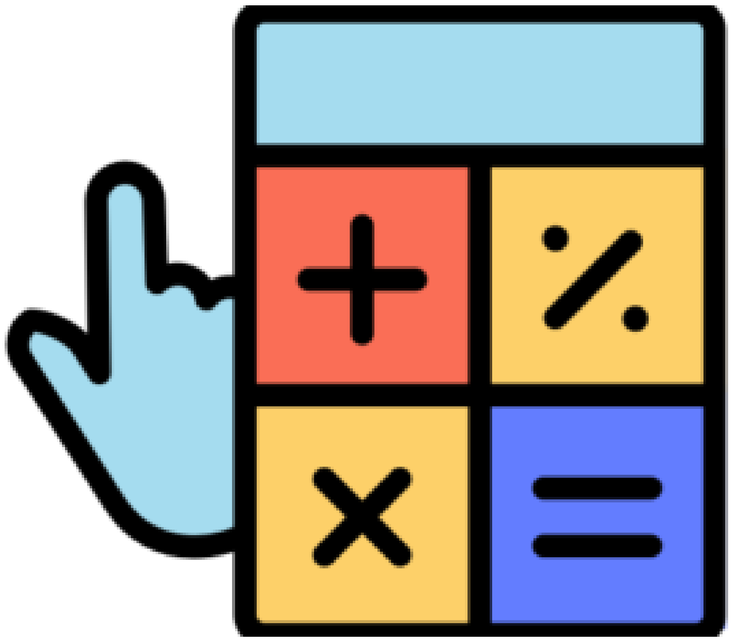
É uma **simulação do uso daquela classe**, ou seja, é **necessário instanciar**, **chamar um método**, **passar parâmetros**, **pegar o retorno** e assim por diante.

```
public class Testes {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();
        int soma;

        soma = calc.somar(9, 10);
        System.out.println(soma);

        soma = calc.somar(8, 0);
        System.out.println(soma);

        soma = calc.somar(-1, 0);
        System.out.println(soma);

        soma = calc.somar(3, -1);
        System.out.println(soma);
    }
```

vnt/school
powered by venturus

Testes Automatizados

Porém, imagine que uma **calculadora** opera com, pelo menos, **quatro operações básicas** (somar, subtrair, multiplicar e dividir), cada um com **vários cenários**.



Isso resultaria em um código muito extenso.

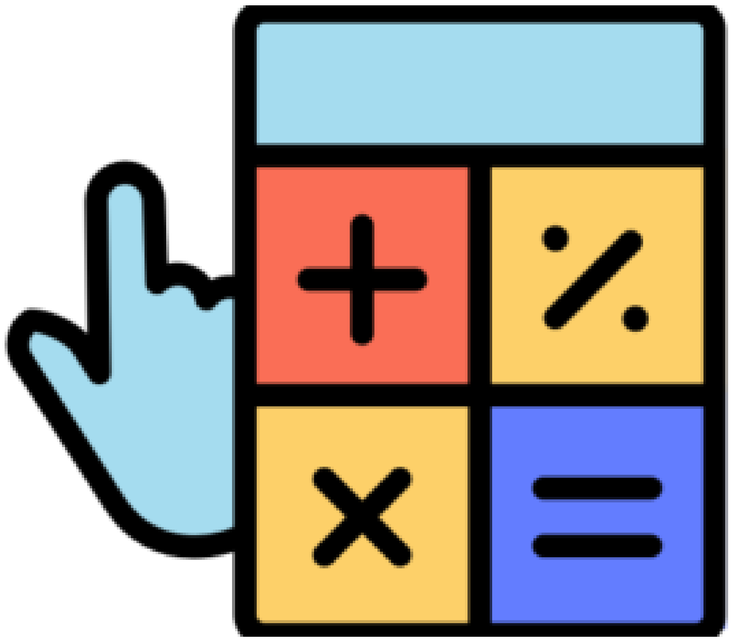
Além disso, o uso do **método main()** exige a realização de algumas etapas de maneira **manual**.

Testes Automatizados

Outro **problema** é que o console exibe apenas o número como resultado.

Se estivermos lidando com uma grande variedade de cenários, essa abordagem tende a se tornar **cansativa**.

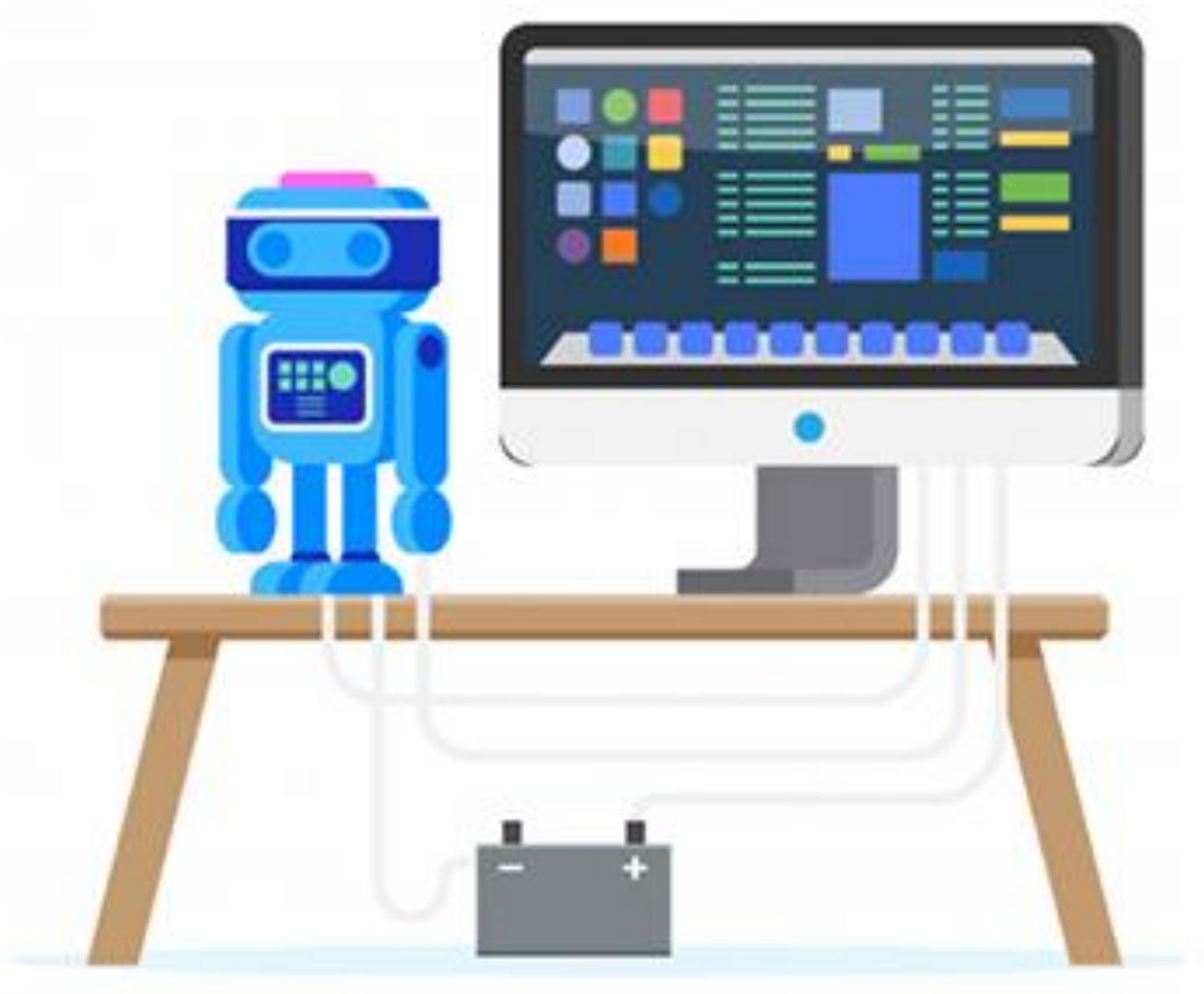
Solução: JUnit, uma biblioteca que serve para simplificar todo esse processo.



vnt/school
powered by venturus

JUnit

Testes Automatizados



vnt/school
powered by venturus

JUnit

Biblioteca padrão para a escrita de **testes automatizados** em **linguagem *Java***.

Foi **criada em 1995** por **Kent Beck e Erich Gamma**. Beck é considerado o "pai dos testes automatizados"

O **JUnit** é uma biblioteca gratuita e de **código aberto**.

vnt/school
powered by venturus

JUnit

Simplifica a escrita de **testes de unidade (ou testes unitários)**, que são a categoria mais simples dos **testes automatizados**.

Existem **outras categorias** como os testes de **integração**, de **API**, ***end-to-end***, aceitação, performance, estresse, etc.

Existem também **outras bibliotecas** e ferramentas disponíveis, mas neste curso o foco será no uso do **JUnit** para os **testes de unidade**.

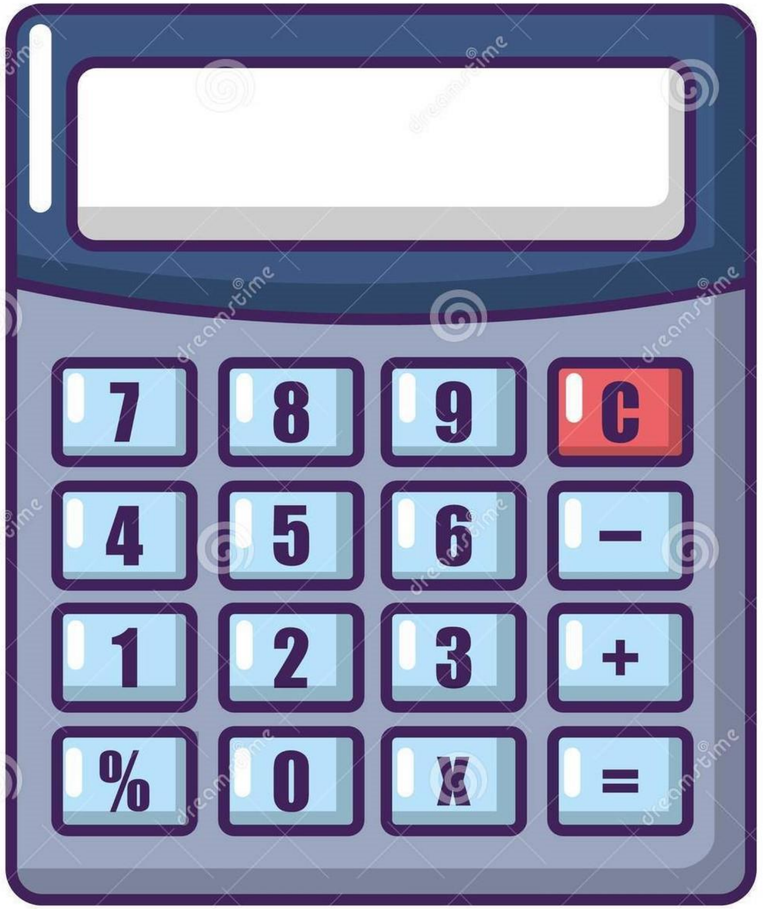
vnt/school
powered by venturus

JUnit

Crie um código simples simulando uma **calculadora** que realiza operações de soma.

Feito isso, é hora de **testar** a nossa calculadora!

Como seriam os testes com JUnit?

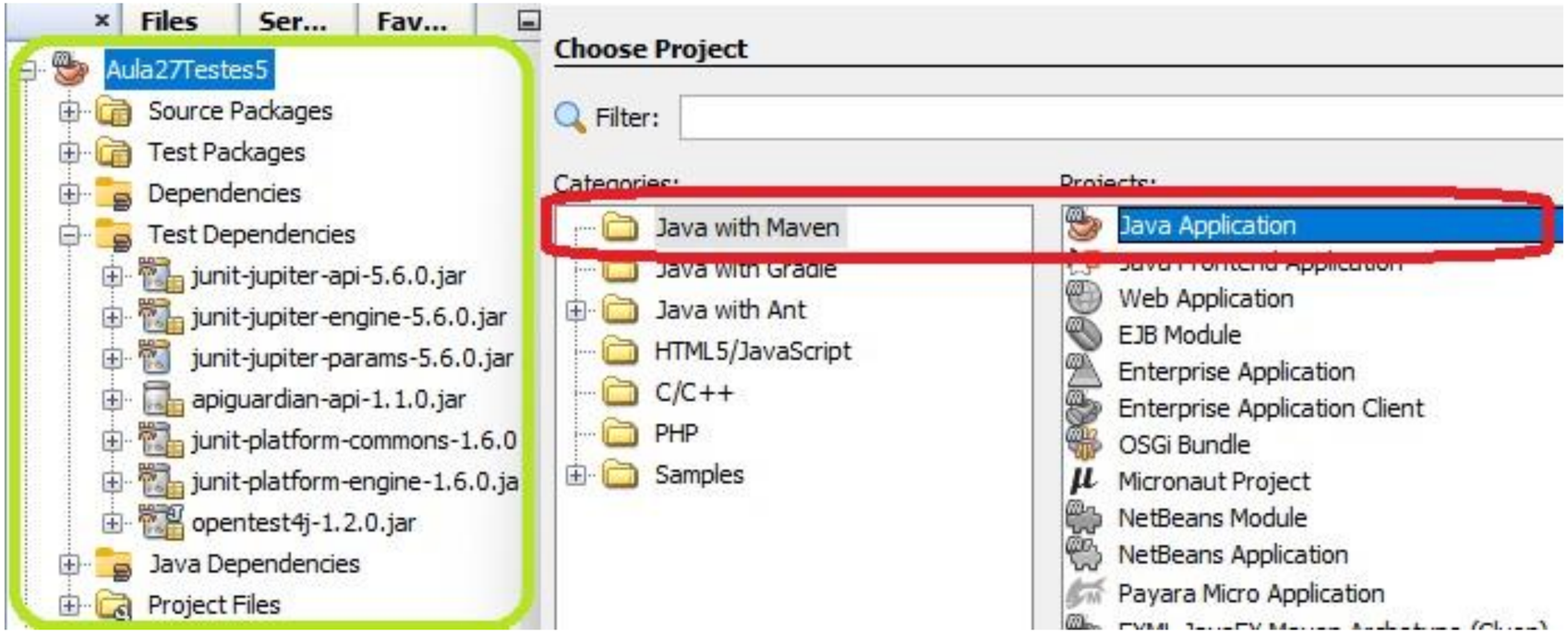


vnt /school
powered by venturus

JUnit

Vermelho: cria um arquivo Java com Maven. Opçao para fazer testes.

Verde: apresenta os pacotes de Dependências. Bibliotecas do JUnit.



vnt/school
powered by venturus

JUnit

Antes de iniciar os testes é importante baixar o **Maven** e fazer a **integração** com a **IDE Netebeans**.

Acesse o site abaixo e faça o **download** do pacote **Maven**.

https://maven.apache.org/download.cgi

vnt/school
powered by venturus

JUnit

Siga o passo a passo do vídeo indicado abaixo. Ele te ajudará na **instalação do Maven.**

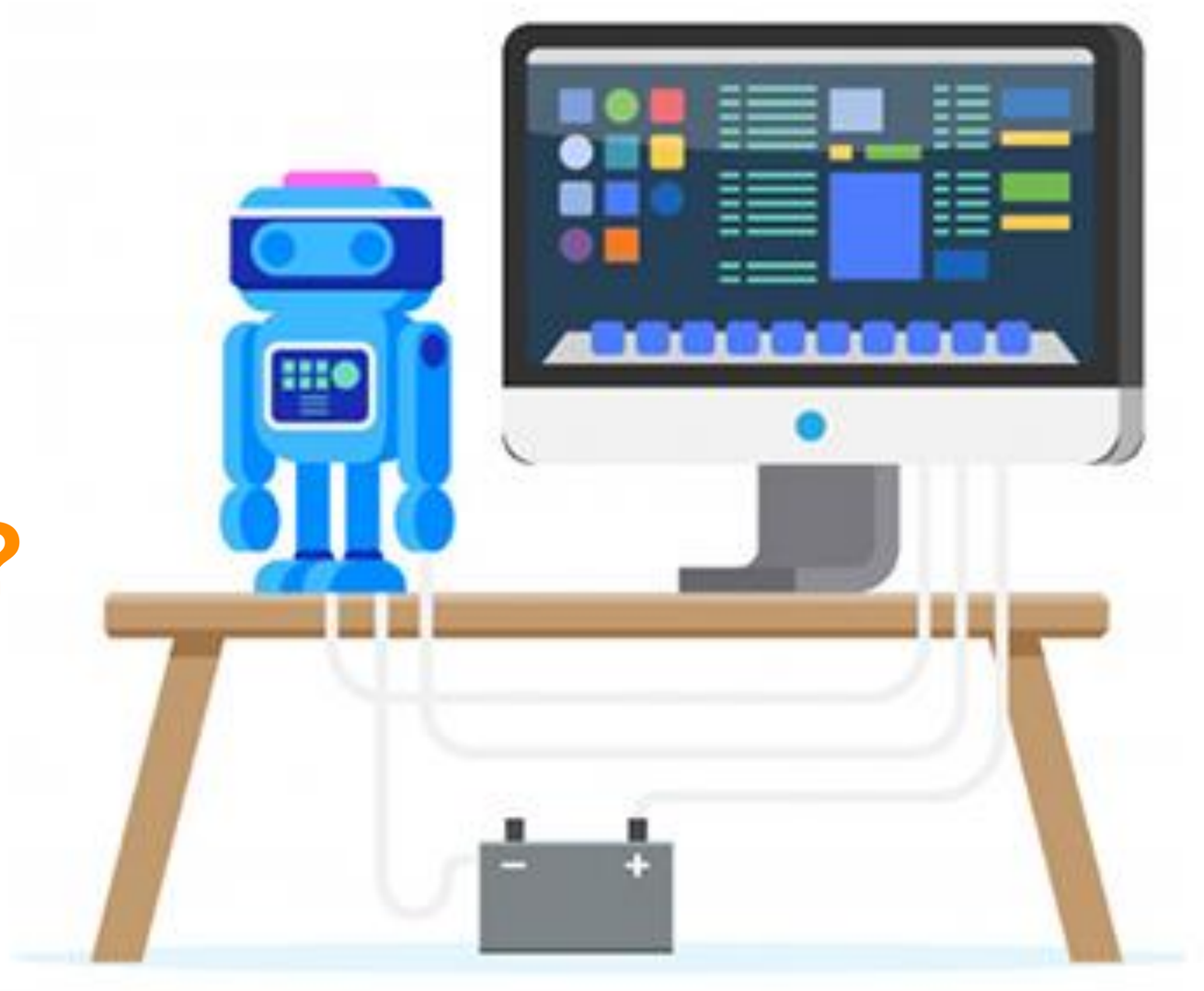
https://www.youtube.com/watch?v=mZY9aoAY2vk

É fundamental que a **integração** seja feita com a **IDE Netbeans.**
Inclusive, acrescentar o **path** na **variável de ambiente** do sistema.

vnt/school
powered by venturus

Vamos Praticar?

JUnit



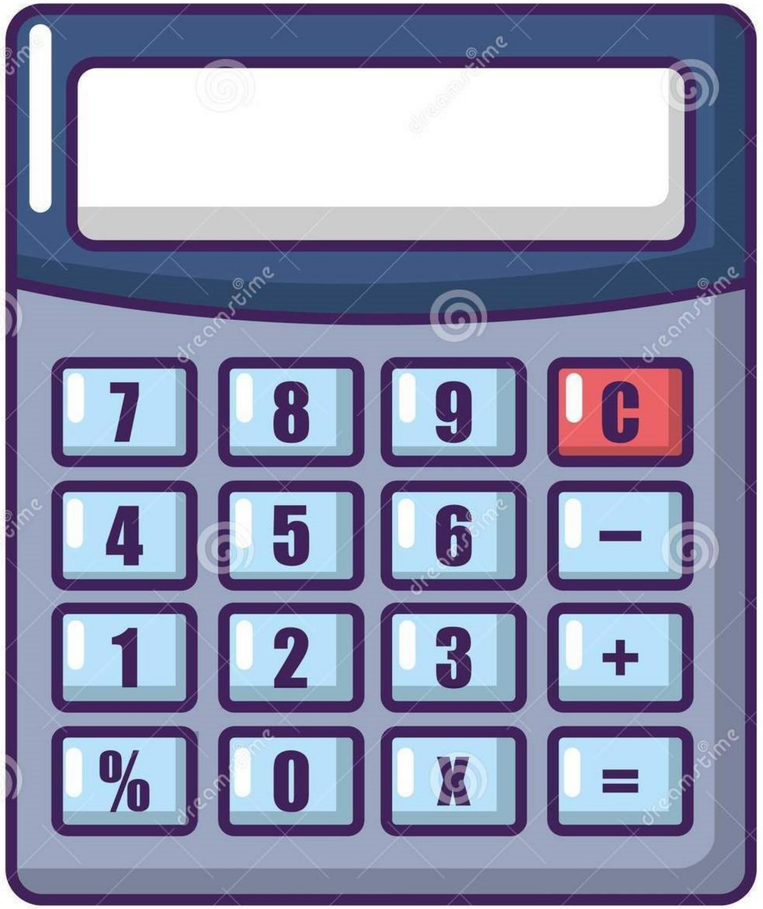
vnt/school
powered by venturus

JUnit

Crie um código simples simulando uma **calculadora** que realiza operações de soma.

Feito isso, é hora de **testar** a nossa calculadora!

Como seriam os testes com JUnit?

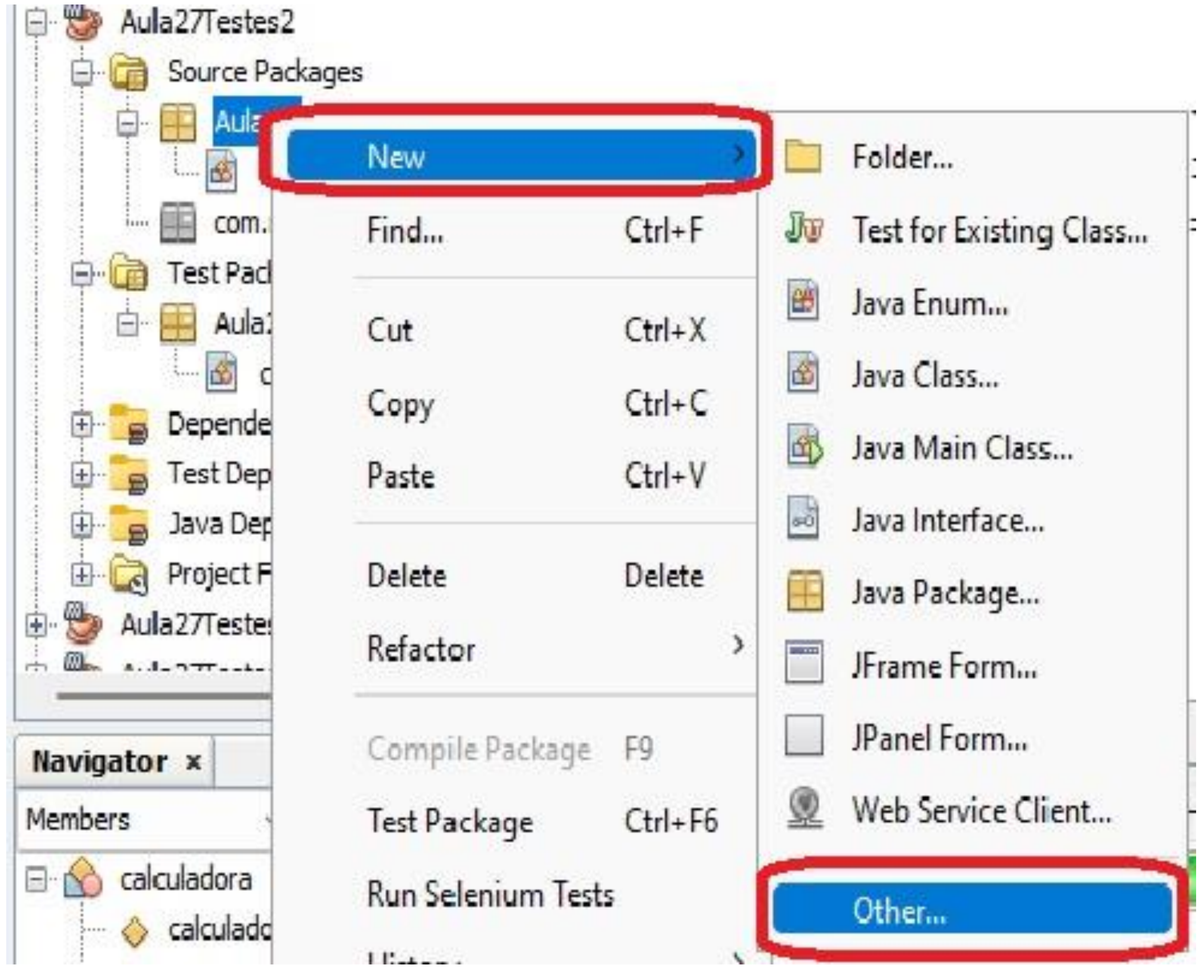


vnt /school
powered by venturus

JUnit

Após ter criado a **classe Calculadora** e ter desenvolvido toda a lógica proposta no trabalho, é hora de fazer o teste.

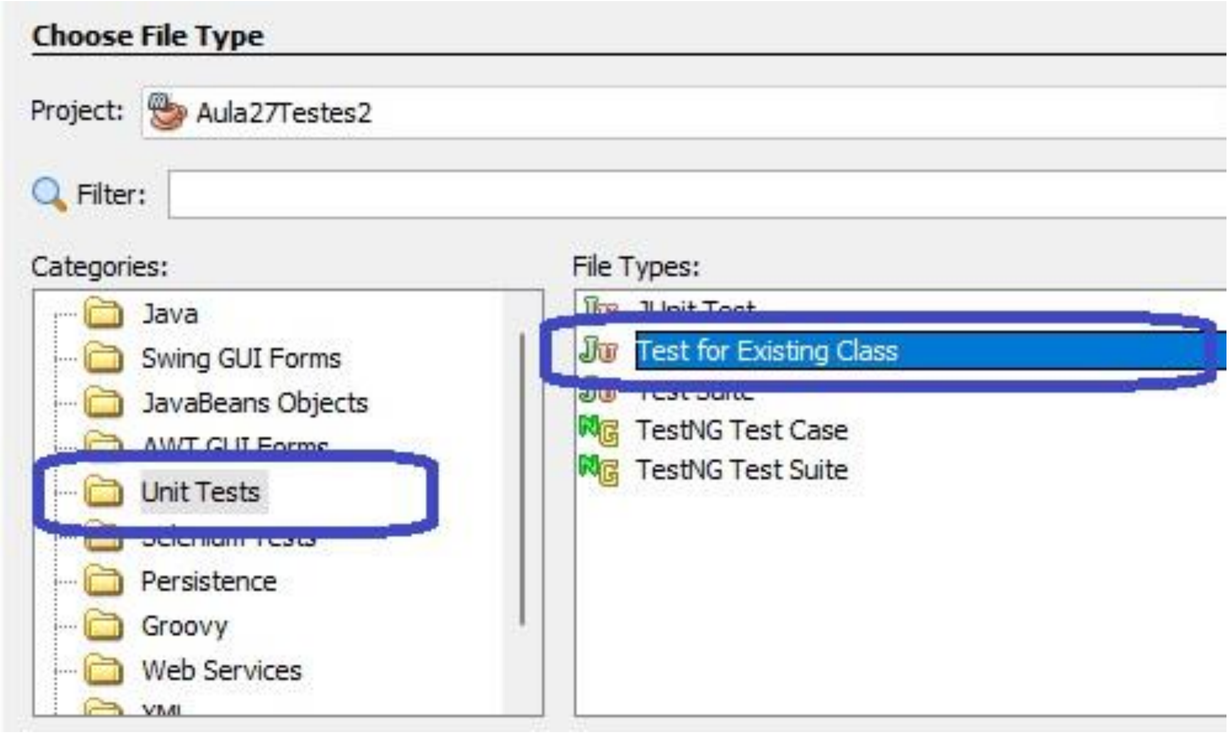
Clique com o **botão direito** do mouse sobre o **pacote principal** e New -> Other.



vnt/school
powered by venturus

JUnit

Em seguida selecione a categoria **Unit Tests** e o tipo de arquivo Test for Existing Class.



Veja que será criado testes para classes que já existem no seu **código**.

vnt/school
powered by venturus

JUnit

Sabendo que a **classe Calculadora** já havia sido criado, obviamente esta é a indicação do NetBeans para que se crie o teste.

vnt/school
powered by venturus

JUnit

Veja que a classe **TestSomeMethod()** foi criada.

Basta fazer a lógica do que deve ser testado.

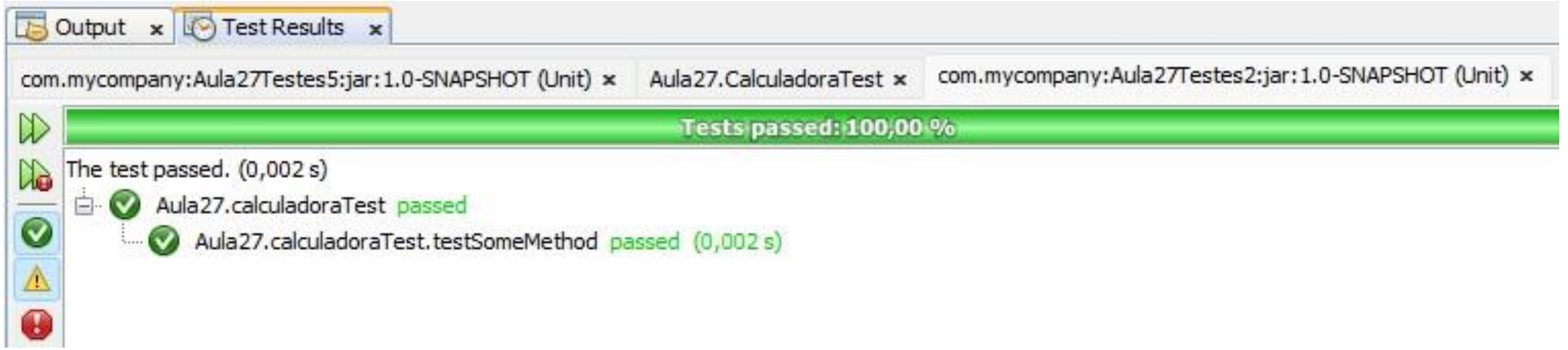
Vários cenários de teste podem ser criados.

```
package Aula27;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class calculadoraTest {
    @Test
    public void testSomeMethod() {
        calculadora calc = new calculadora();
        int soma = calc.somar(3, 7);
    }
}
```

vnt/school
powered by venturus

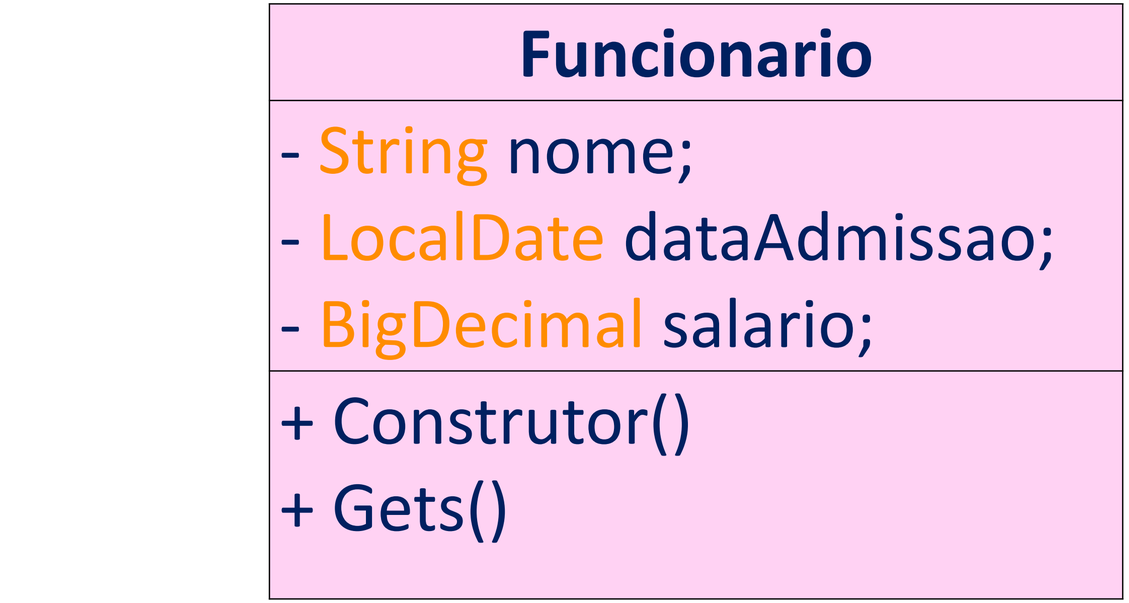
JUnit

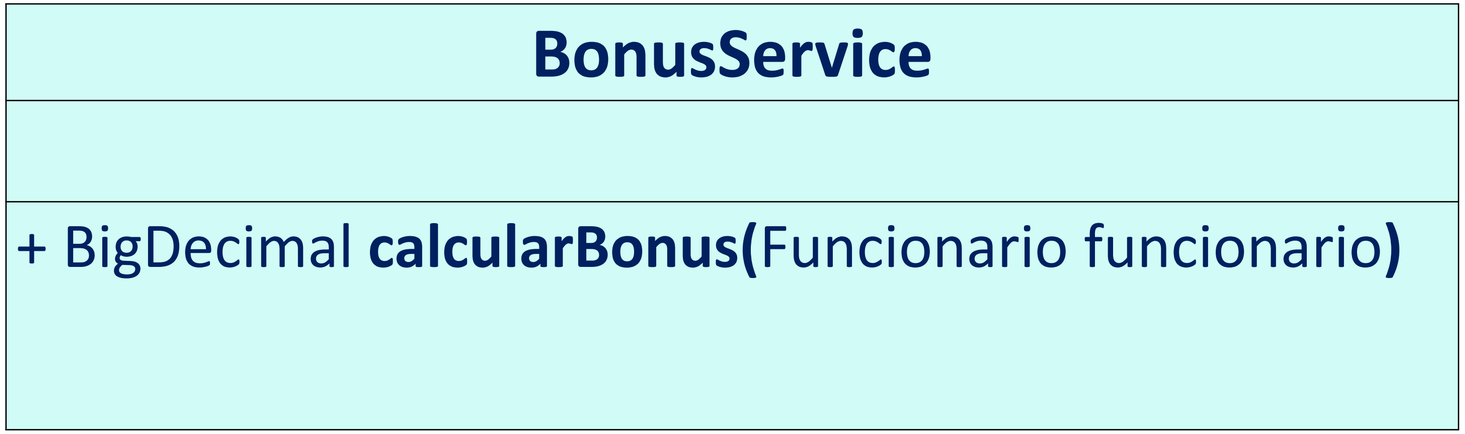
Para **rodar o teste**, clique com o botão direito no código do teste, e clique em Teste File.



JUnit

Exercício 1: crie uma aplicação que simule o controle de funcionários, reajustes salariais e cálculo de bônus. Crie as duas classes como no diagrama. **Funcionario** **BonusService.**





vnt/school
powered by venturus

JUnit

A primeira classe **`Funcionario`** representa um funcionário dentro de uma empresa, reunindo dados como **`nome`**, **`dataAdmissao`** e **`salario`**. Insira o construtor para receber essas três informações e os métodos **`get`**s para recuperá-las.

A segunda classe **`BonusService`** é uma classe de serviço que executa uma regra de negócio da aplicação, que é a de calcular o bônus a ser recebido por um funcionário.

O método **`calcularBonus`** recebe um objeto (**`funcionario`**) e devolve um **`BigDecimal`**, que é o valor do bônus que esse funcionário vai receber. O bônus equivale a 10% do salário do funcionário. Se os 10% ultrapassar o valor de 1000 reais o **bônus** deve ser **zero**.

vnt/school
powered by venturus

JUnit

Esta é a lógica da classe **BonusService** e que deve ser testada.

```
public class BonusService {
    public BigDecimal calcularBonus(Funcionario funcionario) {
                BigDecimal valor = funcionario.getSalario().multiply(new BigDecimal("0.1"));
                if (valor.compareTo(new BigDecimal("1000")) > 0) {
                        valor = BigDecimal.ZERO;
                }
                return valor.setScale(2, RoundingMode.HALF_UP);
        }
}
```

vnt/school
powered by venturus

JUnit

Vários cenários são testados pra validar o código.

```
public class BonusServiceTest {
    @Test
    public void testSomeMethod() {
        BonusService service = new BonusService();
                BigDecimal bonus = service.calcularBonus(new Funcionario("Rodrig
                assertEquals(new BigDecimal("0.00"), bonus);
    }
    @Test
    public void testSomeMethod2() {
        BonusService service = new BonusService();
                BigDecimal bonus = service.calcularBonus(new Funcionario("Rodrig
                assertEquals(new BigDecimal("250.00"), bonus);
    }
    @Test
    public void testSomeMethod3() {
        BonusService service = new BonusService();
                BigDecimal bonus = service.calcularBonus(new Funcionario("Rodrig
                assertEquals(new BigDecimal("1000.00"), bonus);
    }
```

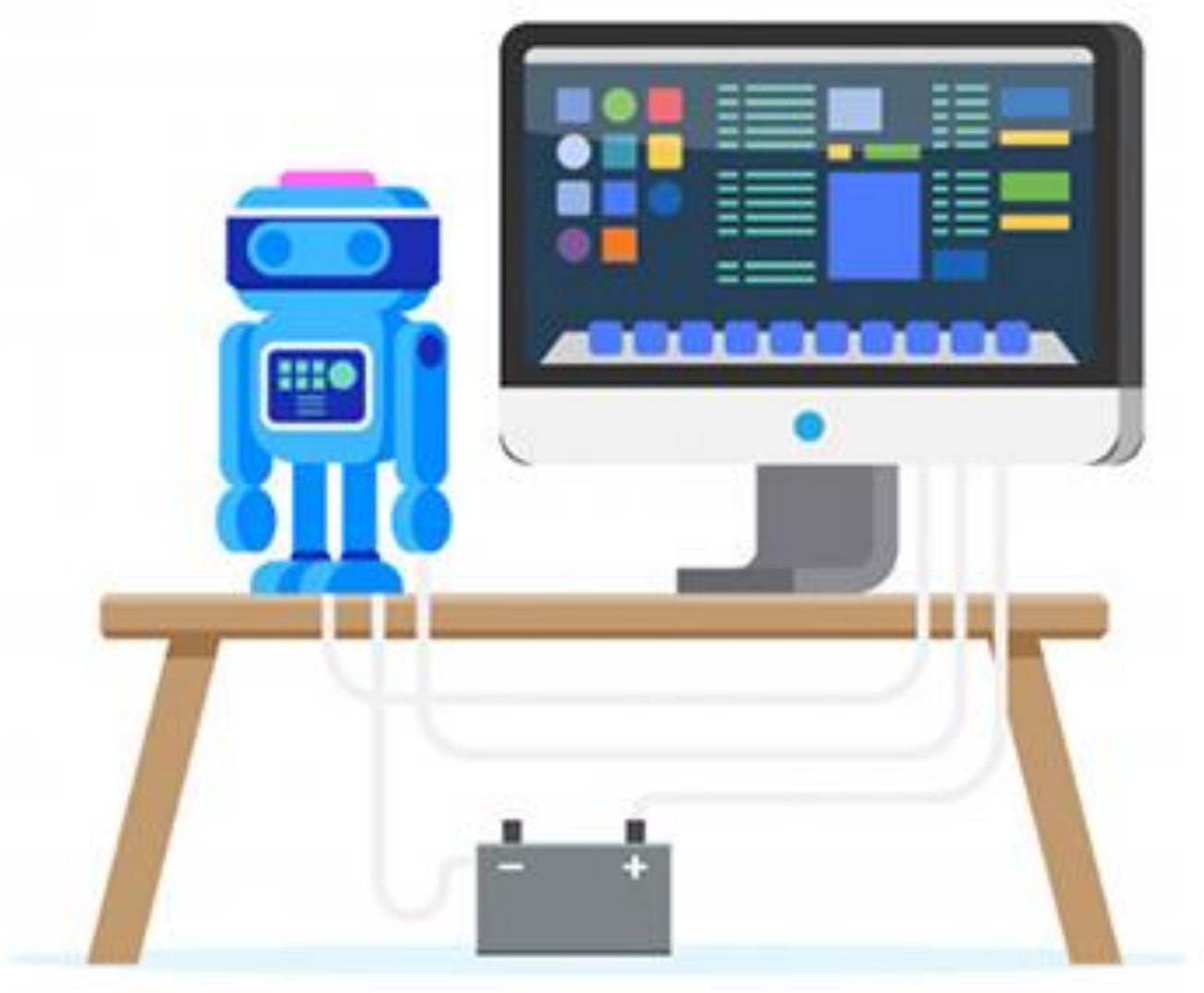
??

vnt/school
powered by venturus

coffee
time

Test Drive Development

JUnit

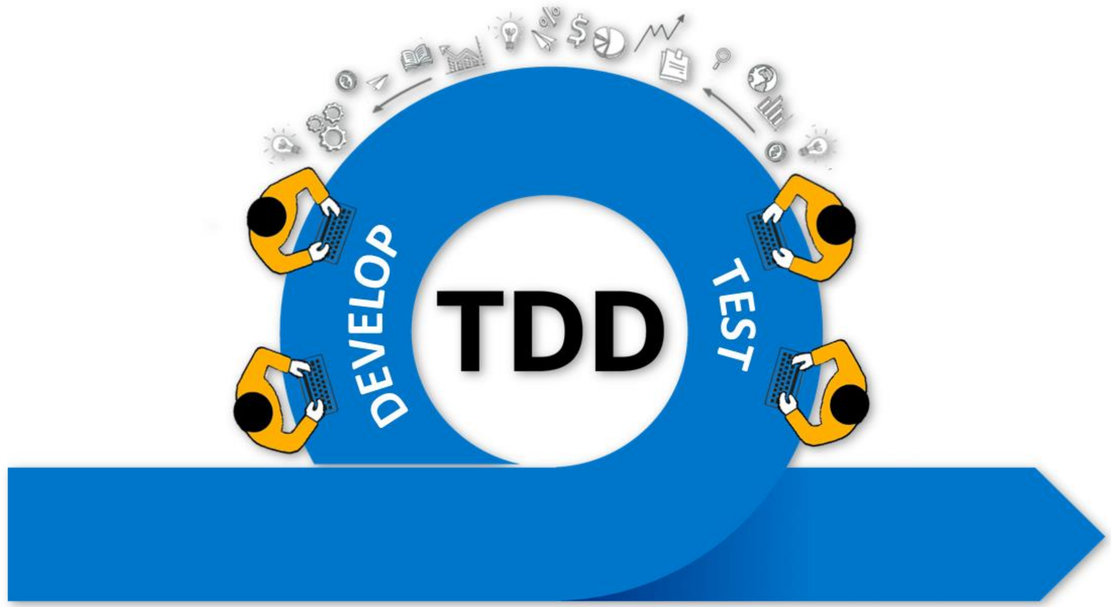


vnt/school
powered by venturus

Test Drive Development

Quando aprendemos a fazer essa modalidade de teste, é mais comum seguirmos um fluxo em que fazemos **primeiro a implementação** do código para **depois testá-lo**.

Porém, a outra abordagem possível é **inverter o processo**: começar escrevendo o **teste automatizado** para **depois implementar o código**.



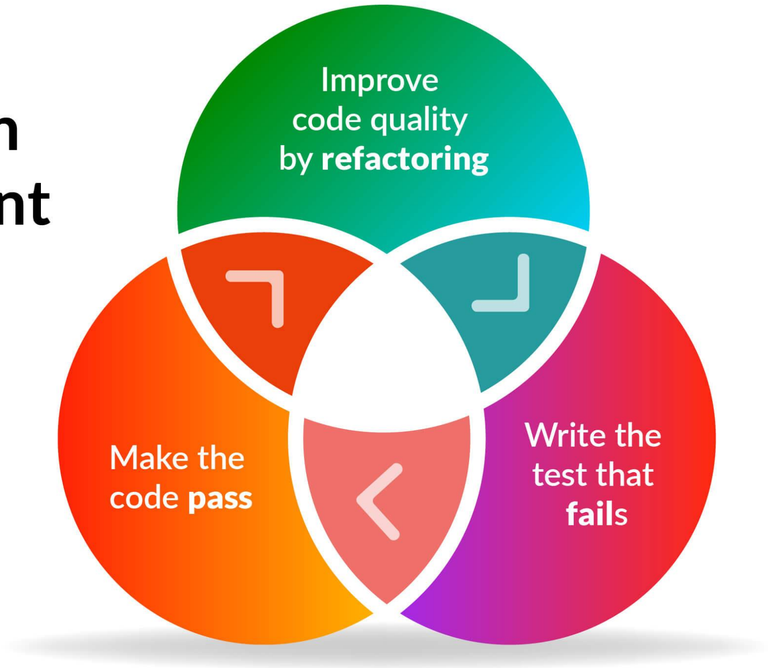
vnt/school
powered by venturus

Test Drive Development

***Test Driven Development* (TDD)**

Prioriza testes automatizados ainda na fase de projeto com o objetivo de obter software de qualidade, isto é, com **código limpo** e que **funcione**.

Os testes guiam todo o desenvolvimento.



vnt/school
powered by venturus

Test Drive Development

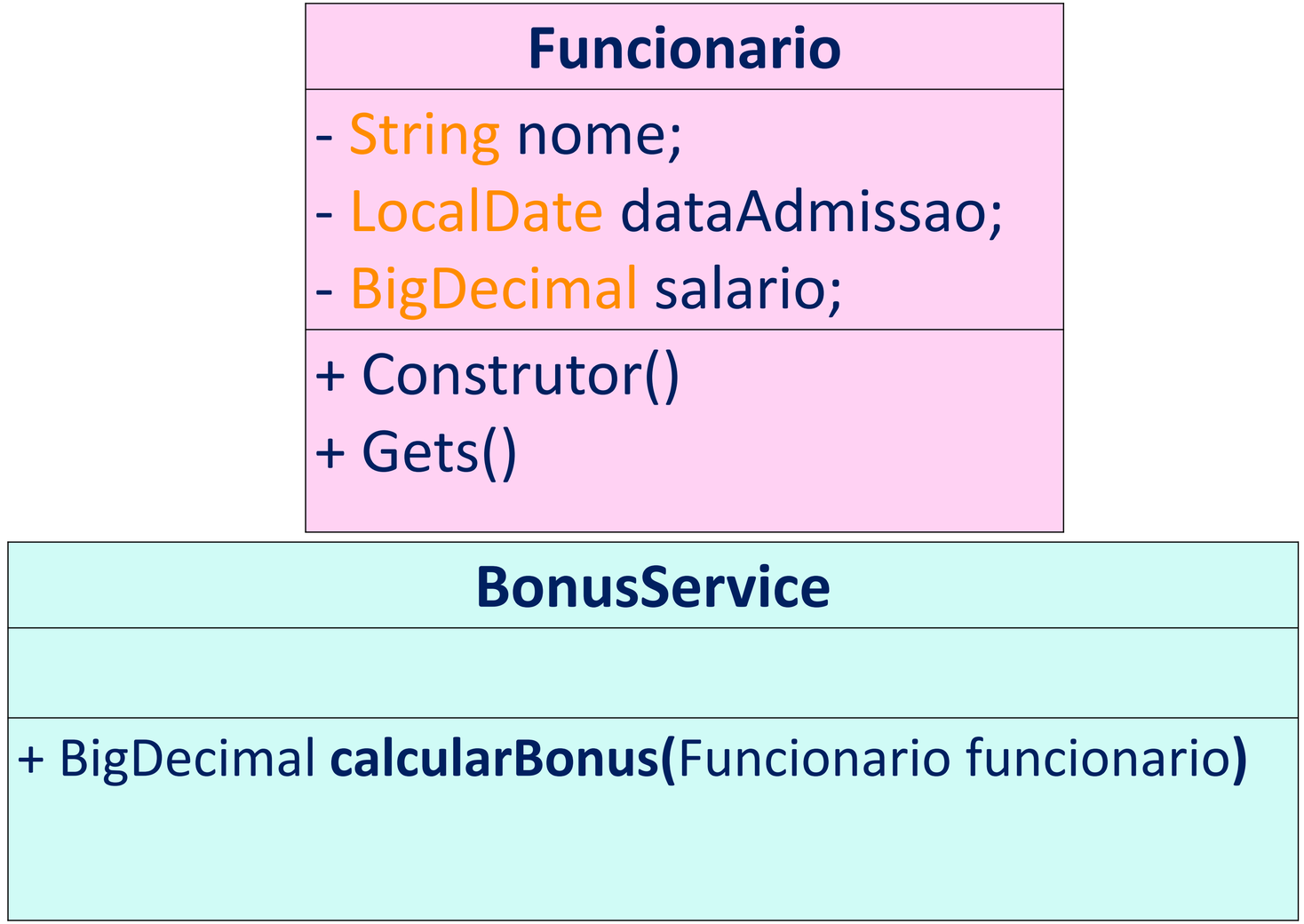
Exercício 2:

Desenvolva uma funcionalidade referente ao **reajuste anual de salário dos funcionários** de uma empresa. O sistema deve permitir que os funcionários recebam um reajuste salarial anual baseado em seu desempenho, obedecendo às seguintes regras:

- Desempenho classificado como "**A desejar**", o reajuste será de **3% do salário** atual.
- Se o desempenho for "**Bom**", o reajuste será de **15% do salário**.
- Se o desempenho for "**Ótimo**", o reajuste será de **20% do salário**.

Test Drive Development

Reutilize o **exercício 1**, onde foi feito uma aplicação que simula o controle de funcionários, reajustes salariais e cálculo de bônus. E as duas classes foram criadas conforme o diagrama.
Funcionario
BonusService.



vnt/school
powered by venturus

Test Drive Development

De acordo com as informações podemos supor que nosso código de implementação vai se chamar **`ReajusteService`** e ficará localizado no pacote **`service`**.

Service
`ReajusteService`

Seguindo a abordagem do **TDD**, pode-se criar primeiro a classe de teste **`ReajusteServiceTest`**.

Teste
`ReajusteServiceTest`

vnt/school
powered by venturus

Test Drive Development

Com base nos objetivos dessa funcionalidade, temos pelo menos três cenários possíveis. Começaremos por aquele no qual o desempenho foi "**A desejar**" e, portanto, o reajuste será de apenas **3% do salário atual**.

```
public class ReajusteServiceTest {
    @Test
    public void testDesempenhoADesejar() {
        ReajusteService service = new ReajusteService();
        Funcionario func = new Funcionario("Ana", LocalDate.now(), new BigDecimal("1000"));
        service.concederReajuste(func, Desempenho.A_DESEJAR);
        assertEquals(new BigDecimal("1030.00"), func.getSalario());
    }
```

vnt/school
powered by venturus

Test Drive Development

Uma vez escrito o código do teste, vamos para a **implementação**.

Crie a classe `ReajusteService`.

Basta seguir a indicação feita pelo próprio **Netebeans**.

```
class ReajusteService {
    void concederReajuste(Funcionario func, Desempenho desempenho) {
        if (desempenho == Desempenho.A_DESEJAR){
            BigDecimal reajuste = func.getSalario().multiply(new BigDecimal("0.03"));
            func.reajustarSalario(reajuste);
        }
```

Test Drive Development

Crie **enum Desempenho** sugerido pelo Netebeans.

Coloque-o no pacote **modelo**.

Após o **código mínimo** ser implementado. Faça o **teste rodar** e a**nalise o código**.

```
package modelo;
import java.math.BigDecimal;
public enum Desempenho {
    A_DESEJAR, BOM, OTIMO
}
```

vnt/school
powered by venturus

Test Drive Development

Se a tarefa de rodar o primeiro teste, foi realizada com sucesso, **adapte** esse código aos dois **cenários restantes** (desempenhos "**Bom**" e "**Ótimo**").

Lembre-se que ambos possuem porcentagens diferentes de reajuste (**15% e 20%**, respectivamente) e terão valores distintos como resultado.

Test Drive Development

```java
public class ReajusteServiceTest {
    @Test
    public void testDesempenhoADesejar() {
        ReajusteService service = new ReajusteService();
        Funcionario func = new Funcionario("Ana", LocalDate.now(), new BigDecimal("1000"));
        service.concederReajuste(func, Desempenho.A_DESEJAR);
        assertEquals(new BigDecimal("1030.00"), func.getSalario());
    }
    @Test
    public void testDesempenhoBom() {
        ReajusteService service = new ReajusteService();
        Funcionario func = new Funcionario("Ana", LocalDate.now(), new BigDecimal("1000"));
        service.concederReajuste(func, Desempenho.BOM);
        assertEquals(new BigDecimal("1150.00"), func.getSalario());
    }
    @Test
    public void testDesempenhoOtimo() {
        ReajusteService service = new ReajusteService();
        Funcionario func = new Funcionario("Ana", LocalDate.now(), new BigDecimal("1000"));
        service.concederReajuste(func, Desempenho.OTIMO);
        assertEquals(new BigDecimal("1200.00"), func.getSalario());
    }
```

Test Drive Development

```
class ReajusteService {
    void concederReajuste(Funcionario func, Desempenho desempenho) {
        if (desempenho == Desempenho.A_DESEJAR){
            BigDecimal reajuste = func.getSalario().multiply(new BigDecimal("0.03"));
            func.reajustarSalario(reajuste);
        }else if (desempenho == Desempenho.BOM) {
            BigDecimal reajuste = func.getSalario().multiply(new BigDecimal("0.15"));
            func.reajustarSalario(reajuste);
        }else{
            BigDecimal reajuste = func.getSalario().multiply(new BigDecimal("0.20"));
            func.reajustarSalario(reajuste);
        }
```

Test Drive Development

Vantagens de usar o TDD:

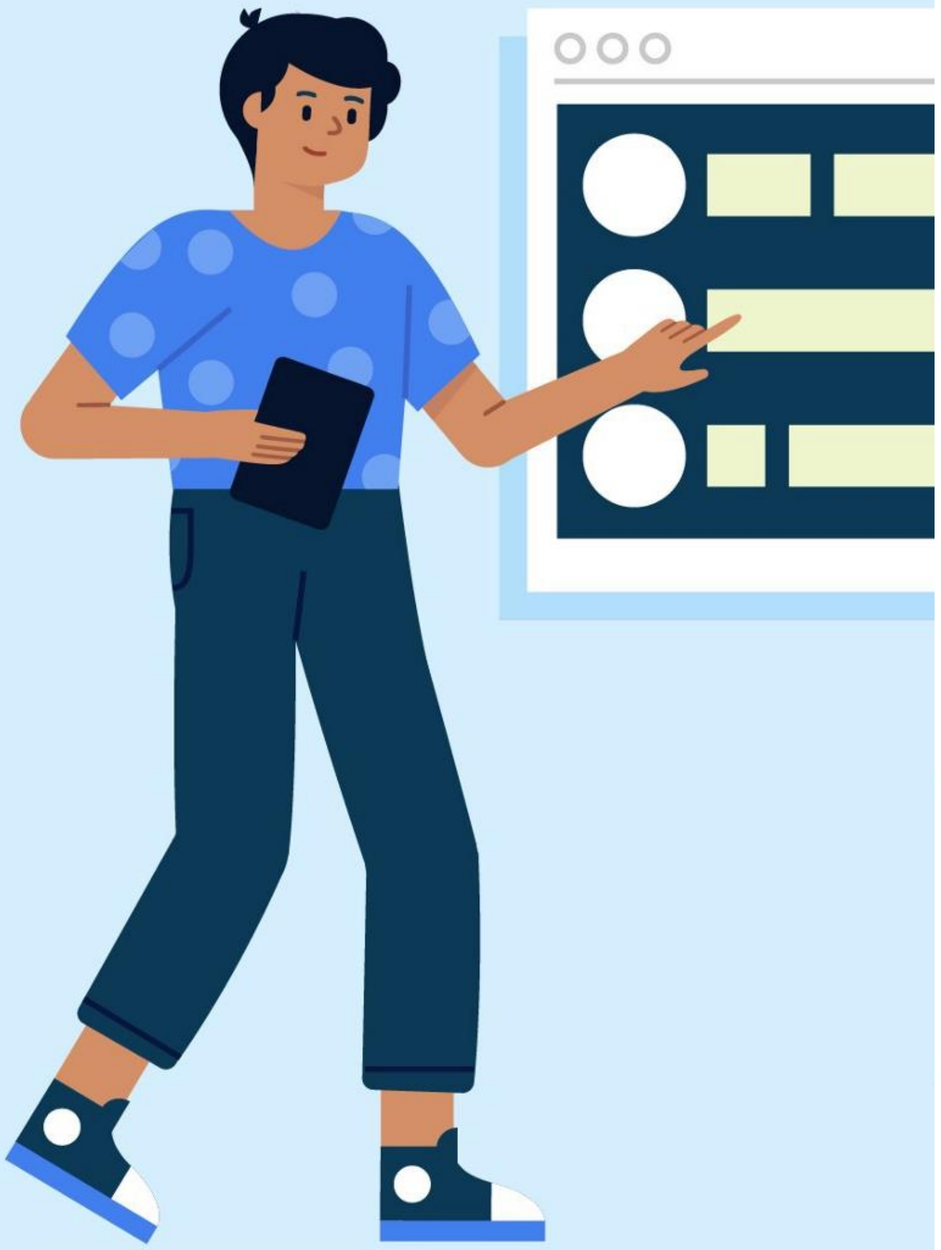
Código já sai "**testado**" A primeira delas é que você termina de programar e já sai com o código "testado".

Refatorar faz parte do processo. A refatoração permite melhorias no código de maneira contínua.

Tendência a escrever um **código mais simples** quando usamos o TDD.

vnt/school
powered by venturus


Review
e
Preview



Comunidade VNT

Dica de hoje

O **JUnit** é uma biblioteca gratuita e de código aberto.
Seu código-fonte pode ser encontrado no link abaixo.

JUnit:

https://github.com/junit-team/junit5

Referências

[1] A. Goldman, F. Kon, Paulo J. S. Silva; Introdução à Ciência da Computação com Java e Orientação a Objetos (USP). 2006. Ed. USP.
[2] Algoritmo e lógica de programação. Acessado julho/2022: https://visualg3.com.br/
[3] G. Silveira; Algoritmos em Java; Ed. Casa do Código.
[4] M. T. Goodrich, R. Tamassia; Estrutura de dados e algoritmos em Java. Ed Bookman. 2007.
[5] Algoritmo e lógica de programação. Acessado julho/2022: https://www.cursoemvideo.com/
[6] P. Silveira, R. Turini; Java 8 Pratico: lambdas, streams e os novos recursos da linguagem. Ed. Casa do Código.
[7] Linguagem Java: Curso acessado em agosto/2022: https://www.udemy.com/
[8] Linguagem Java: Curso acessado em setembro/2022: https://www.cursoemvideo.com/