ECM251 – Linguagens de Programação I

Aula 19 – L1/1 e L2/1

Engenharia da Computação – 3ª série

TDD e JUnit

(L1/1 – L2/1)

2023

Horário

Terça-feira: 2 aulas/semana

- L1/1 (07h40min-09h20min): *Prof. Calvetti*;
- L2/1 (07h40min-09h20min): *Prof. Igor Silveira*;

Tópico

- Código Confiável

Código Confiável

Definição

- Um **Código Confiável**, ou funcional, é o objetivo primário e inegociável para quem desenvolve *software*;
- Seja qual for a situação, o desenvolvedor é responsável por entregar um produto que funcione;
- O **Código Confiável** deve estar de acordo com suas especificações e seus objetivos;
- Um **Código Confiável** deve passar por **Testes de Validação**, que validam se o *software* atende ao que foi solicitado, sob o ponto de vista do cliente;
- O **Código Confiável** dever passar por **Testes de Verificação**, que verificam se o *software* não possui **Erros**, **Defeitos** e **Falhas**.

Conclusão

- Quanto mais avançados e distantes da fase de desenvolvimento do *software* forem feitos seus **Testes de V&V** – Verificação e Validação e neles forem encontrados Erros, Defeitos e Falhas, maiores serão os esforços (custos, prazos e recursos) necessários para corrigi-los;
- Daí a necessidade de se ter um **Código Confiável** o mais breve possível no seu ciclo de desenvolvimento.

Tópico

- Código Limpo

Definição

- Um **Código Limpo** é um código-fonte que deve, dentre várias outras, possuir as seguintes principais características:
  1. Simples:
     - Fácil de ser entendido, utilizando lógica direta, variáveis claramente nomeadas, métodos que em seus nomes utilizam verbos demonstrando seus propósitos, com muitos comentários úteis etc., por exemplo, conforme determina o princípio de programação denominado KISS: "*Keep It Simple, Stupid*!".

Código Limpo

Definição

- Um **Código Limpo** é um código-fonte que deve, dentre várias outras, possuir as seguintes principais características:

  2. Organizado:
     - Espaçado, subdividido e indentado de maneira consistente, dando clareza ao que está sendo escrito.

Definição

- Um **Código Limpo** é um código-fonte que deve, dentre várias outras, possuir as seguintes principais características:

  3. Sem Duplicações:
     - Utilizando funções ou métodos, diminuindo o acoplamento e evitando a repetição do código e seu consequente espalhamento pelo *software* que está sendo escrito, tornando sua manutenção difícil, trabalhosa e arriscada, por exemplo, conforme determina o princípio de programação denominado DRY: *"Don't Repeat Yourself!"*.

Definição

- Um **Código Limpo** é um código-fonte que deve, dentre várias outras, possuir as seguintes principais características:

4. Funções ou Métodos Atômicos:
    - Com o mínimo de linhas de código possível e fazendo unicamente o que se destinam a fazer, apenas uma atividade por função/método e bem alinhada o seu nome, tornando-os mais objetivos e de mais fácil manutenção, potencializando o reuso e o teste.

Conclusão

- A importância de um **Código Limpo** está em, quando juntas suas principais características, se obter um código altamente legível, de entendimento rápido e que poderá ser facilmente melhorado, pois é fato que, num futuro, será necessário alterá-lo ou reutilizá-lo em outros projetos.
- Escrever um **Código Limpo**, segundo Robert Martin, em seu livro Código Limpo – Habilidades Práticas do Agile Software, vai exigir o uso disciplinado dessas várias técnicas.

Código limpo = Código de fácil manutenção!

Tópico

- *Test-Driven Development* – TDD

Definição

- O **TDD** – *Test-Driven Development*, ou Desenvolvimento Guiado por Testes é um estilo de programação, criado por Kent Beck em 2003, que guia o desenvolvimento de um *software* apoiando-se em testes automatizados;
- Tem como objetivos obter código confiável e limpo, verificar com eficácia os erros presentes no *software* em seu desenvolvimento e validar se os requisitos do *software* são atendidos;
- O **TDD** trabalha com a ideia de que a realização dos testes comece logo nas primeiras linhas de código escritas pelo desenvolvedor e prossiga, continuamente, durante todo o seu ciclo de desenvolvimento.

Definição

- Alguns benefícios do **TDD**:
  1. Reduções significativas nas taxas de defeitos encontrados nas fases de testes unitários e de integração;
  2. Reduções significativas do esforço nas fases finais dos projetos, ao custo de um aumento moderado no esforço nas fases iniciais do desenvolvimento;
  3. O código-fonte atinge graus mais altos de qualidade técnica;
  4. Facilita a manutenção do *software*;
  5. Torna os testes documentações vivas do comportamento do *software*;
  6. Torna o projeto do *software* mais modular; etc.

Definição

- O **TDD** define duas regras, apenas, para serem seguidas, obrigatoriamente, que também determinam a ordem das tarefas da programação:
  1. Escrever um código novo apenas se o teste automatizado falhar;
  2. Eliminar a duplicação de código.
- Daí surge o conhecido "mantra" do **TDD**:

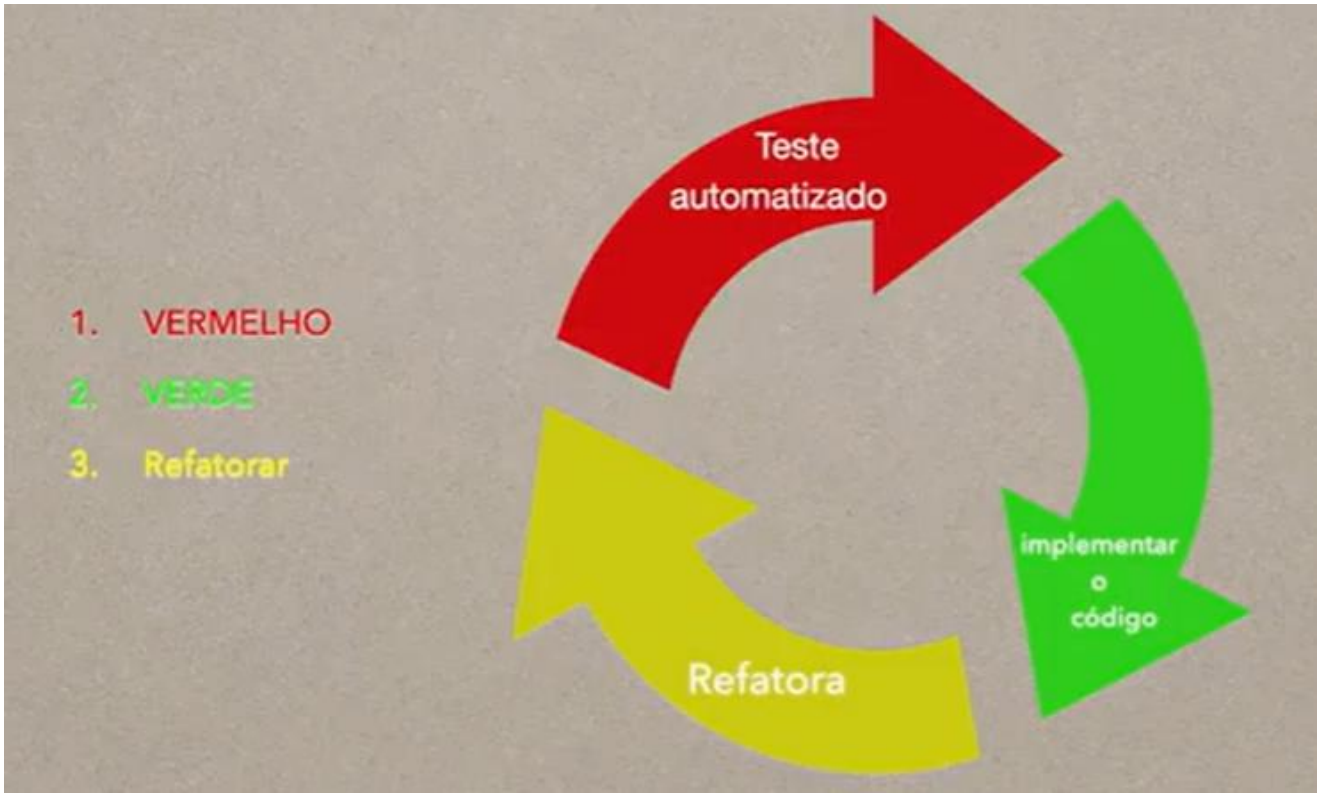
1. Vermelho;

2. Verde; e

3. Refatorar.

Definição

- Ciclo do **TDD**:



Definição

- Ciclo do desenvolvimento guiado por testes:
  1. Escrever um teste para uma determinada funcionalidade, sem escrever o código dessa funcionalidade (*test first*);
  2. Executar esse teste, sem ainda ter implementado essa funcionalidade, visando que o teste indique falha;
  3. Implementar essa funcionalidade, de forma simples, rápida e provisória (*baby Steps*), visando ser aprovada em seu teste;
  4. Executar o teste, novamente, para a funcionalidade alterada, visando que, agora, o teste indique sucesso;
  5. Refatorar o código da funcionalidade para que seja aprovada, agora com seu código confiável, limpo e definitivo;

Definição

- Ciclo do desenvolvimento guiado por testes:
  6. Executar o teste, novamente, para a funcionalidade refatorada, visando que o teste indique sucesso, novamente;
  7. Escolher uma nova funcionalidade e reiniciar o ciclo do desenvolvimento guiado por testes;
  8. A medida que se for avançando nos testes e nas funcionalidades aprovadas, todos os testes feitos anteriormente devem ser executados novamente, repetindo-se esse ciclo, aumentando, assim, a confiança no código que está sendo produzido.

Exemplo

- Ordenação de *N* números inteiros em um vetor:
  1. Definir o ambiente para executar o método de ordenação e seus respectivos testes: a classe **ExemploTDD**

```
1 public class ExemploTDD
2 {
3    public static void main(String args[])
4    {
5       OrdenaTest tdd = new OrdenaTest();
6    }
7 }
8
```

Exemplo

- Ordenação de *N* números inteiros em um vetor:
  2. Criar a classe de testes e os respectivos casos de testes para o requisito solicitado: a classe **OrdenaTest**

```
1  public class OrdenaTest
2  {  public OrdenaTest()
3     {  int proposto[]    = new int[] {10, 9};
4        int esperado[]    = new int[] {9, 10};
5        int inesperado[]  = new int[] {9};
6
7        Ordena teste = new Ordena();
8        teste.ordenaNumerosCrescentes(proposto);
9
10       System.out.println("Teste de Ordenação\n==================");
11       System.out.println("Ficou com o mesmo tamanho: " + caso1Test(proposto.length, inesperado.length));
12       System.out.println("Ordenou com sucesso......: " + caso2Test(proposto, esperado));
13    }
14
```

Exemplo

- Ordenação de *N* números inteiros em um vetor:
  3. Criar os métodos que executam os respectivos casos de testes para o requisito solicitado: a classe **OrdenaTest**

```
15    public boolean caso1Test(int tamprop, int tamesp)
16    {  boolean resp = true;
17       if(tamprop != tamesp)   resp = false;
18       return   resp;
19    }
20
21    public boolean caso2Test(int prop[], int esp[])
22    {  return   numerosIguais(prop, esp);
23    }
24
```

Exemplo

- Ordenação de *N* números inteiros em um vetor:
  4. Criar os métodos auxiliares dos casos de testes para o requisito solicitado: a classe **OrdenaTest**

```
25    public boolean numerosIguais(int nums1[], int nums2[])
26    {  boolean  resultado = true;
27       for(int i = 0, j = 0; i < nums1.length; i++, j++)
28       {  if(nums1[i] != nums2[i])
29          {  resultado = false;
30             i = nums1.length;
31          }
32       }
33       return resultado;
34    }
35 }
```

Exemplo

- Ordenação de *N* números inteiros em um vetor:

5. Iniciar o ciclo do TDD, escrevendo um código rápido para que a funcionalidade falhe nos testes já criados (VERMELHO): a classe **Ordena**

```
1 public class Ordena
2 {  public void ordenaNumerosCrescentes(int vetor[])
3    {  // Não há código, portanto, não irá ordenar o vetor
4    }
5 }
6
```

Exemplo

- Ordenação de *N* números inteiros em um vetor:

6. Executar os casos de testes anteriormente criados para testar a funcionalidade recém escrita, objetivando se obter as falhas nos testes (VERMELHO): a classe **ExemploTDD**

```
1 public class ExemploTDD
2 {
3    public static void main(String args[])
4    {
5       OrdenaTest tdd = new OrdenaTest();
6    }
7 }
8
```

```
---- exec: java ExemploTDD
Teste de Ordenação
==================
Ficou com o mesmo tamanho: false
Ordenou com sucesso.....: false

---- operation complete.
```

Exemplo

- Ordenação de *N* números inteiros em um vetor:

7. Avance o ciclo do TDD, escrevendo um código rápido para que a funcionalidade seja aprovada nos testes já criados (VERDE): a classe **OrdenaTest**

```
1  public class OrdenaTest
2  {  public OrdenaTest()
3     {  int proposto[]    = new int[] {10, 9};
4        int esperado[]    = new int[] {9, 10};
5        int inesperado[]  = new int[] {9};
6
7        Ordena teste = new Ordena();
8        teste.ordenaNumerosCrescentes(proposto);
9
10       System.out.println("Teste de Ordenação\n==================");
11       System.out.println("Ficou com o mesmo tamanho: " + caso1Test(proposto.length, esperado.length));
12       System.out.println("Ordenou com sucesso......: " + caso2Test(proposto, esperado));
13    }
14
```

Exemplo

- Ordenação de *N* números inteiros em um vetor:
  7. Reescreva um código rápido para que a funcionalidade seja aprovada nos testes já criados (VERDE): a classe **Ordena**

```
1  public class Ordena
2  {  public void ordenaNumerosCrescentes(int vetor[])
3     {  int rascunho;
4        if(vetor[0] > vetor[1])
5        {  rascunho = vetor[1];
6           vetor[1] = vetor[0];
7           vetor[0] = rascunho;
8        }
9     }
10 }
11
```

Exemplo

- Ordenação de *N* números inteiros em um vetor:

8. Executar os casos de testes anteriormente criados para testar a funcionalidade recém escrita, objetivando o sucesso nos testes (VERDE): a classe **ExemploTDD**

```
1 public class ExemploTDD
2 {
3    public static void main(String args[])
4    {
5       OrdenaTest tdd = new OrdenaTest();
6    }
7 }
8
```

```
----exec: java ExemploTDD
Teste de Ordenação
==================
Ficou com o mesmo tamanho: true
Ordenou com sucesso.....: true

----operation complete.
```

Exemplo

- Ordenação de *N* números inteiros em um vetor:
  9. Avance o ciclo do TDD, melhorando o código em pequenos avanços, ou *Baby Steps* (REFATORAR), repetindo o ciclo até chegar ao código final da funcionalidade: a classe **Ordena**

```
1  public class Ordena
2  {  public void ordenaNumerosCrescentes(int iVet[])
3     {  int   iA, iB, iT;
4        for(iA = 1; iA < iVet.length; iA++)
5        {  for(iB = iVet.length-1; iB >= iA; iB--)
6           {  if(iVet[iB-1] > iVet[iB])
7              {  iT = iVet[iB-1];
8                 iVet[iB-1] = iVet[iB];
9                 iVet[iB] = iT;
10             }
11          }
12       }
13    }
14 }
15
```

Exemplo

- Ordenação de *N* números inteiros em um vetor:

  8. Executar os casos de testes anteriormente criados para testar a funcionalidade final, objetivando o sucesso nos testes finais (VERDE): a classe **ExemploTDD**

```
1 public class ExemploTDD
2 {
3    public static void main(String args[])
4    {
5       OrdenaTest tdd = new OrdenaTest();
6    }
7 }
8
```

```
----exec: java ExemploTDD
Teste de Ordenação
==================
Ficou com o mesmo tamanho: true
Ordenou com sucesso.....: true

----operation complete.
```

Conclusão

- O TDD é um processo que muda a maneira como os desenvolvedores executam a sua rotina de trabalho, pois vão se preocupar, primeiramente, em escrever um teste automatizado para algo que ainda não foi implementado e, somente depois disso, fazer sua implementação;
- A refatoração garante que, antes de se criar um novo teste, o código que acabou de ser criado passará por uma análise, visando retirar duplicações, renomear variáveis e deixar o código o mais limpo possível.

Tópico

- *Framework*

Framework

Definição

- Um ***framework*** é uma estrutura de desenvolvimento de *software* que fornece um conjunto de ferramentas, bibliotecas, padrões e diretrizes que facilitam a criação e o desenvolvimento de aplicativos ou sistemas;
- Essas estruturas ajudam os desenvolvedores a economizar tempo, evitar retrabalho e seguir boas práticas de programação, uma vez que fornecem uma base sólida e organizada para o desenvolvimento de *software*;
- Usados em várias áreas do desenvolvimento, incluindo *Web*, aplicativos móveis, jogos, aprendizado de máquina etc., acelerando os processos de desenvolvimento e tornando-os mais eficientes e robustos.

Framework

Exemplos

1. Desenvolvimento *Web*:
   - *Ruby on Rails*: ***framework*** em **Ruby** para desenvolvimento *web* que segue o padrão *Model-View-Controller* (MVC);
   - *Django*: ***framework*** em **Python** para desenvolvimento *Web* que também segue o padrão MVC.
2. Desenvolvimento *Front-end*:
   - *React*: biblioteca ***JavaScript*** para criação de interfaces de usuário interativas;
   - *Angular*: ***framework*** *JavaScript* completo para criação de aplicativos *Web* complexos.

Framework

Exemplos

3. Desenvolvimento de Aplicativos Móveis:
   - *React Native*: ***framework*** *JavaScript* para desenvolvimento de aplicativos móveis para iOS e Android;
   - *Fluter*: ***framework*** da Google para criar aplicativos móveis para várias plataformas usando a linguagem Dart.
4. Desenvolvimento de Jogos:
   - *Unity*: framework popular para desenvolvimento de jogos 2D e 3D;
   - *Unreal Engine*: ***framework*** utilizado para criação de jogos, conhecido por sua capacidade de produzir jogos de alta qualidade e gráficos impressionantes.

Framework

Exemplos

5. Aprendizado de Máquina:
   - *TensorFlow*: biblioteca código aberto da Google para construção de modelos de aprendizado de máquina;
   - *PyTorch*: biblioteca de aprendizado profundo de código aberto muito popular entre os desenvolvedores de IA.
6. Testes de *software*:
   - *Robot Framework*: ***framework*** de automação de testes de aceitação, testes de unidade e testes de integração, conhecido por sua sintaxe legível por humanos;
   - *JUnit*: ***framework*** de teste unitário bastante popular e amplamente utilizado para aplicações em Java.

Tópico

- *JUnit*

Definição

- O ***JUnit*** é um *framework* de teste de unidade amplamente utilizado na programação Java;
- Fornece uma estrutura para escrever, organizar e executar testes automatizados do código;
- O principal objetivo do ***JUnit*** é ajudar os desenvolvedores a verificar se suas classes e métodos estão funcionando conforme o esperado, garantindo que o código seja testado de maneira consistente e confiável;

Características

1. Testes de Unidade:
   - O ***JUnit*** é usado principalmente para escrever testes de unidade, que são testes que verificam o comportamento de unidades individuais de código, como métodos ou classes, de forma isolada;
   - Ajuda a garantir que cada parte do código funcione conforme o esperado.

Características

2. Anotações:
   - O ***JUnit*** usa anotações Java para marcar métodos como testes;
   - As anotações mais comuns incluem:

     @Test, indicando que um método é um teste;

     @Before, permitindo que sejam definidas configurações e ações de limpeza antes dos testes; e

     @After, permitindo que sejam definidas configurações e ações de limpeza depois dos testes.

Características

3. Asserções:
   - O ***JUnit*** fornece métodos de asserção, como ***assertEquals()***, ***assertTrue()*** e ***assertFalse()***, que são usados nos testes para verificar se as condições especificadas são verdadeiras;
   - Se uma asserção falhar, o teste é considerado "não aprovado".

Características

4. Execução de Testes:
   - O ***JUnit*** permite a execução de testes individualmente, em grupos ou até mesmo em suítes de testes;
   - Isso facilita a organização e a execução de testes de maneira rápida e eficiente.

Características

5. *Test-Driven Development* – TDD:
   - O ***JUnit*** é frequentemente usado em conjunto com a metodologia de desenvolvimento orientado a testes (TDD), na qual os testes são escritos antes do código de produção;
   - Isso ajuda a garantir que o código seja desenvolvido para atender aos requisitos e seja testado continuamente à medida que é desenvolvido.

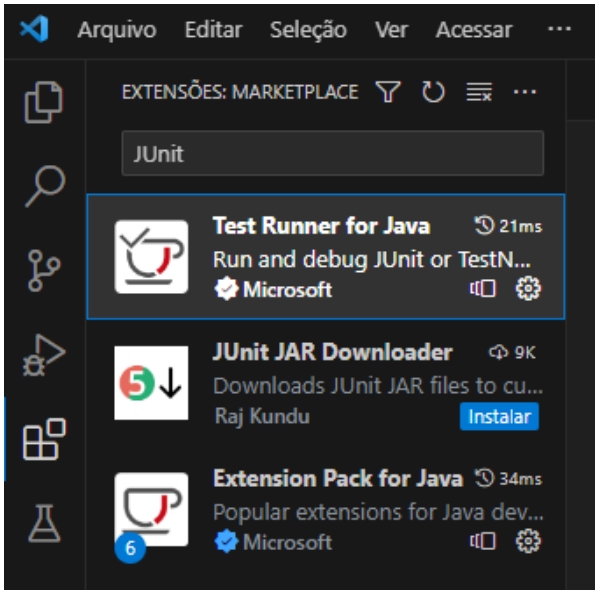
Características

6. Integração com IDEs:
   - Muitas IDEs (Ambientes de Desenvolvimento Integrado), como Eclipse e IntelliJ IDEA e Visual Studio Code (VSC) oferecem suporte integrado ao ***JUnit***, o que facilita a criação, execução e análise de resultados de testes diretamente na interface de desenvolvimento.

JUnit

Instalação

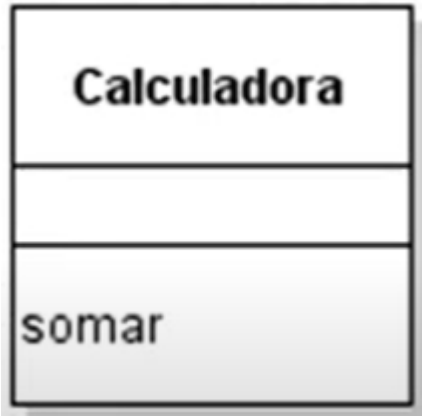
- A título de exemplo, neste material, a IDE escolhida para ser instalado o ***JUnit*** será o *Visual Studio Code,* ou *VsCode*;
- Para outras IDEs, deverá ser localizado o respectivo tutorial, com suas instruções, para a devida instalação do *framework*;



- Estando com o VsCode em execução, com o JDK e o pacote de extensões para Java instalados, na aba EXTENSÕES, no canto esquerdo da tela, instale o pacote *Test Runner for Java*, que contém o suporte para o *framework* ***JUnit 5***.

Exemplo

- Será utilizado o mesmo exemplo apresentado na documentação do ***JUnit***, com as classes **Calculadora** e **CalculadoraTest**, a seguir:

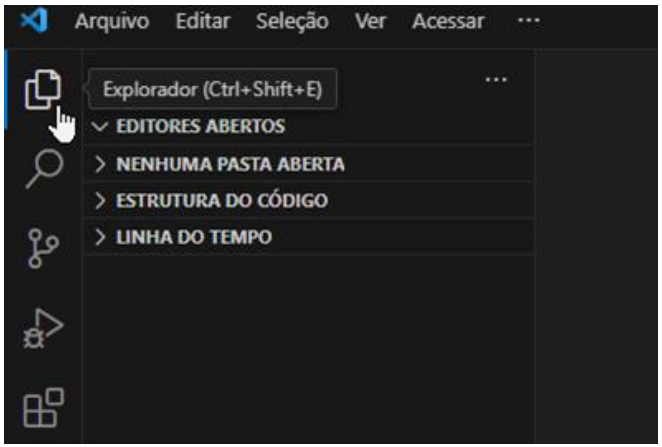


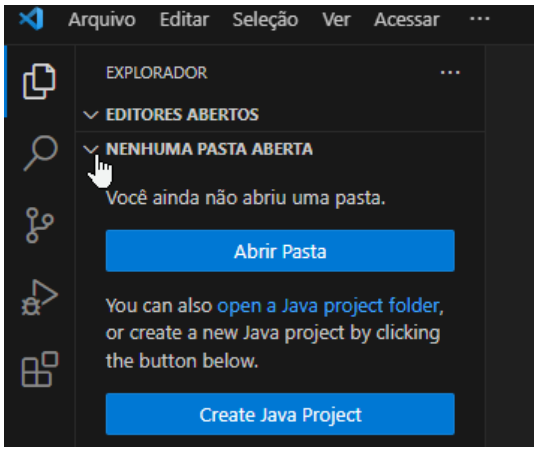
Exemplo

- Será utilizado o mesmo exemplo apresentado na documentação do ***JUnit***, com as classes **Calculadora** e **CalculadoraTest**, a seguir:

Exemplo

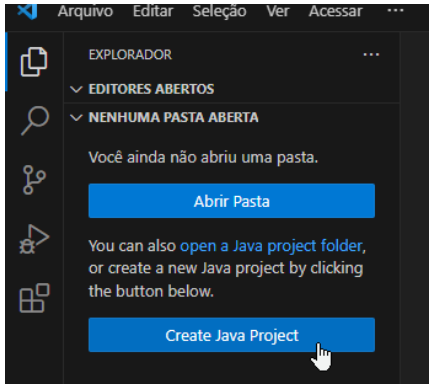
- No ***JUnit***, no canto superior esquerdo, acionar o ícone do **Explorador**, conforme apresentado na figura esquerda abaixo;
- A seguir, selecionar a opção **NENHUMA PASTA ABERTA**, conforme apresentada na figura direita abaixo;

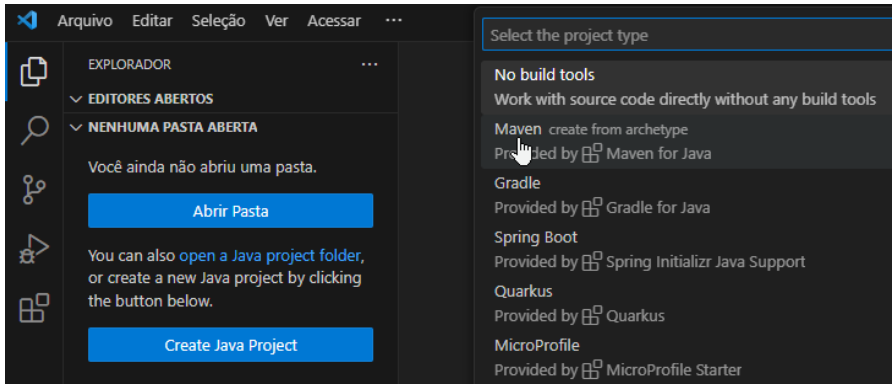


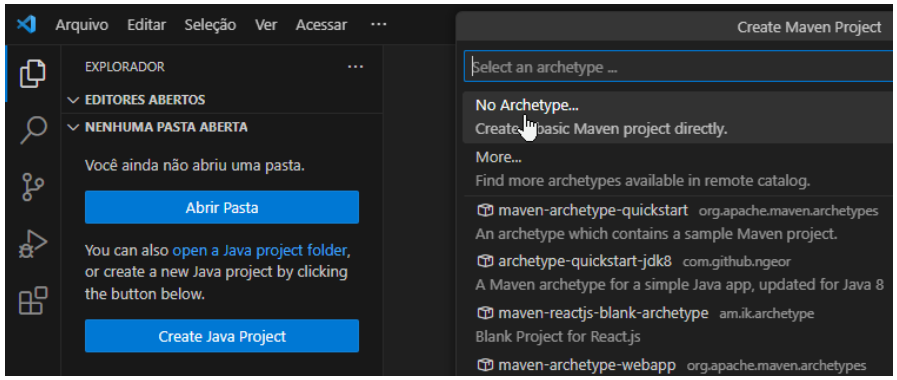


Exemplo

- Em seguida, selecionar a opção **Create Java Project**, seguida da opção **Maven** e **No Archetype...**, dadas pelas figuras abaixo:

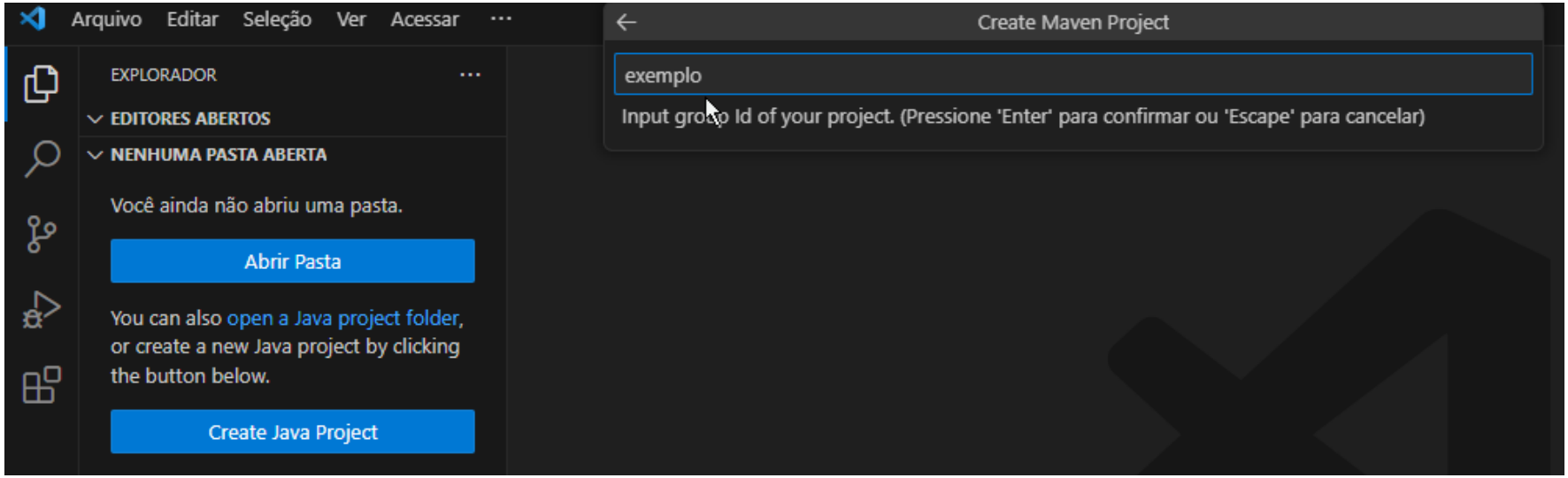






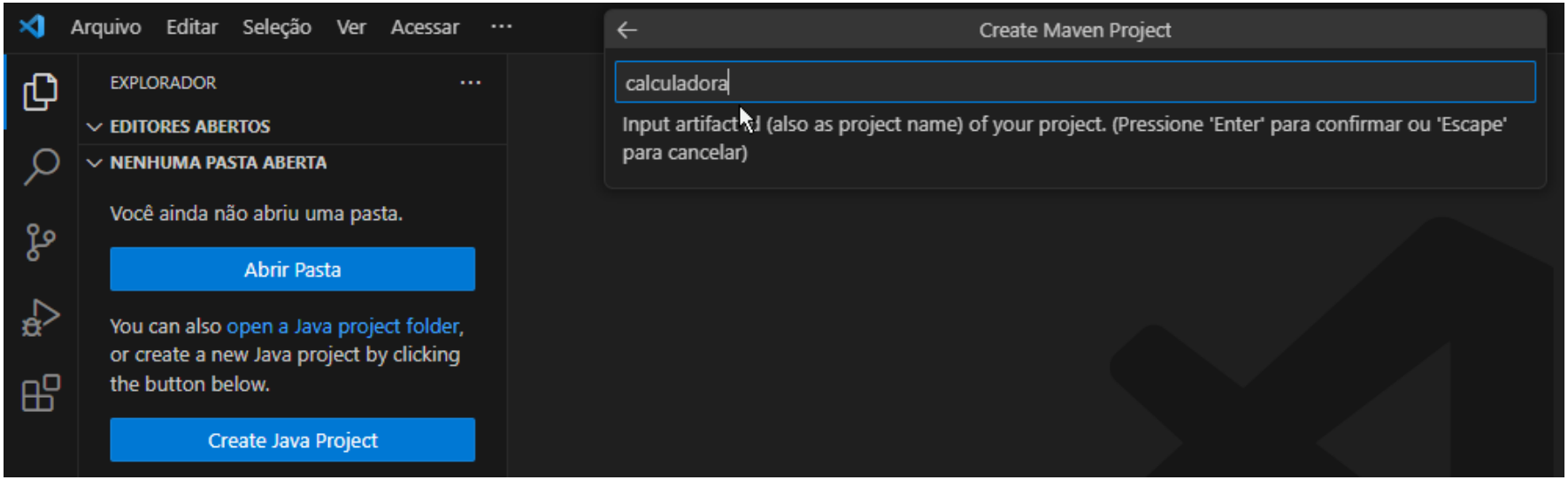
Exemplo

- Daí, digite o id do projeto **exemplo**, seguido de **<ENTER>,** conforme mostra a figura abaixo:



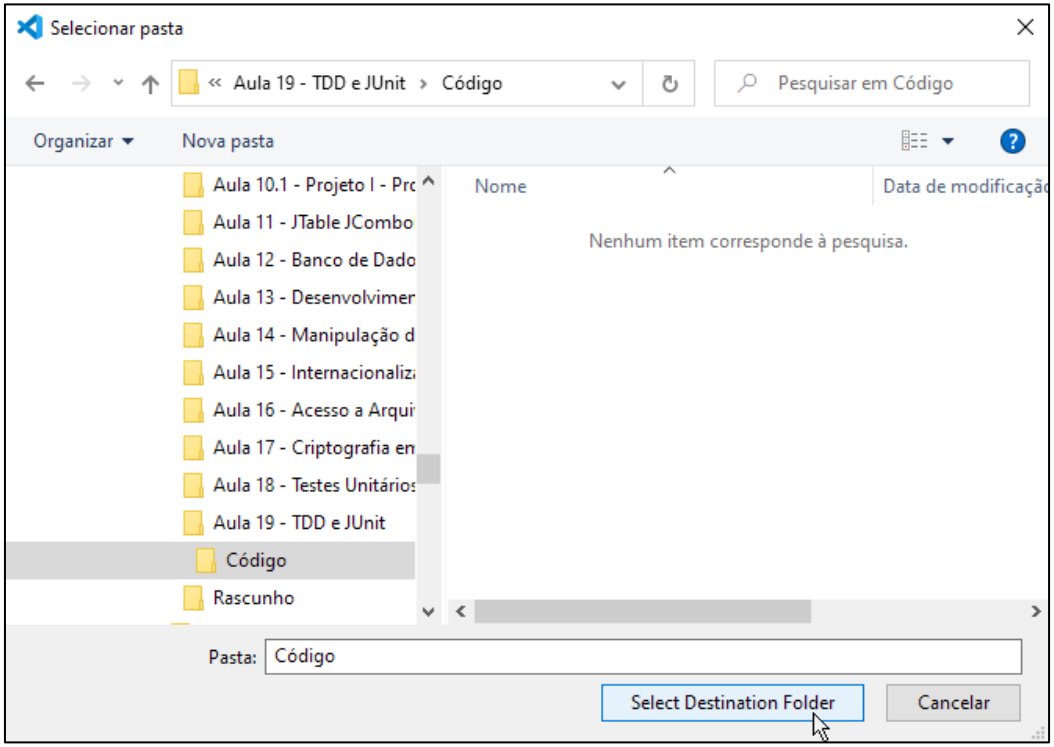
Exemplo

- Então, digite o nome do projeto **calculadora**, seguido de **<ENTER>,** conforme mostra a figura abaixo:



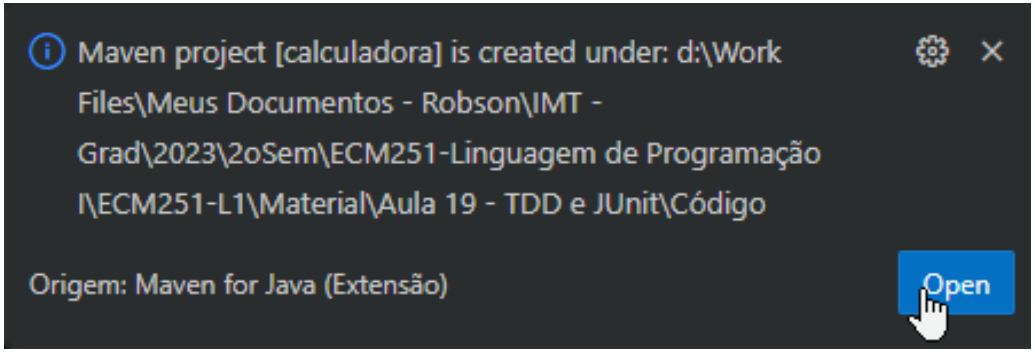
Exemplo

- Agora, escolha o **local da pasta destino**, e acione o botão Select Destination Folder, conforme mostra a figura abaixo:



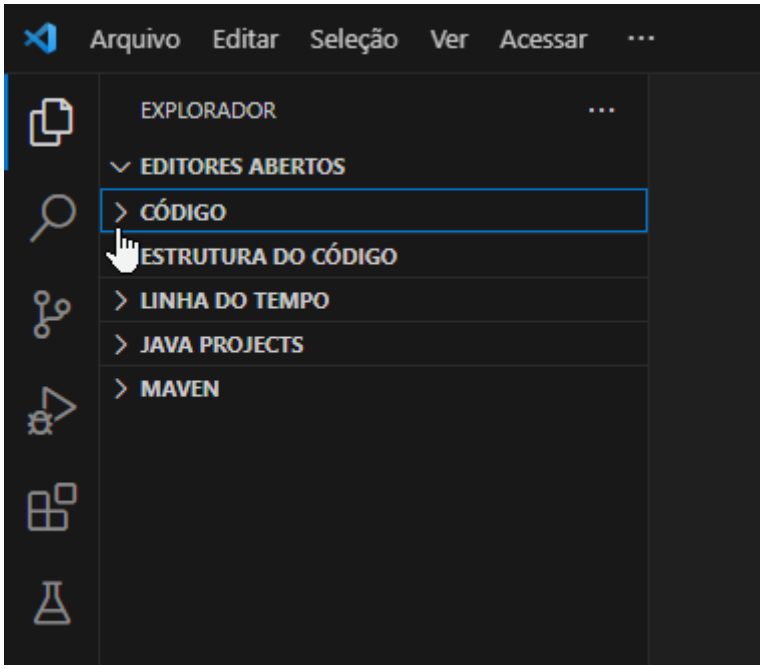
Exemplo

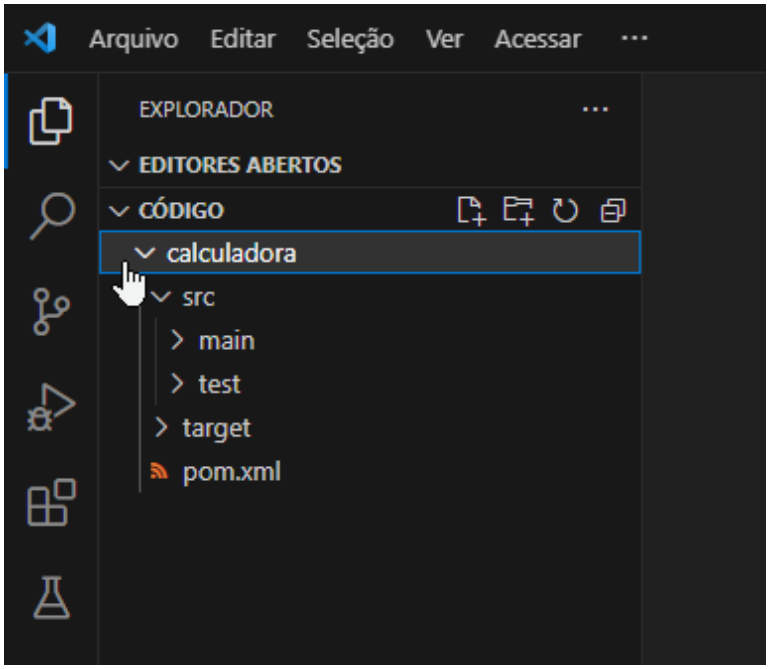
- Acione o botão **Open**, na janela que aparecerá no canto inferior direto, para abrir o projeto**,** conforme mostra a figura abaixo:



Exemplo

- No canto superior esquerdo irá aparecer o nome da pasta destino do projeto, além da sua estrutura interna de pastas, conforme mostram as figuras abaixo:

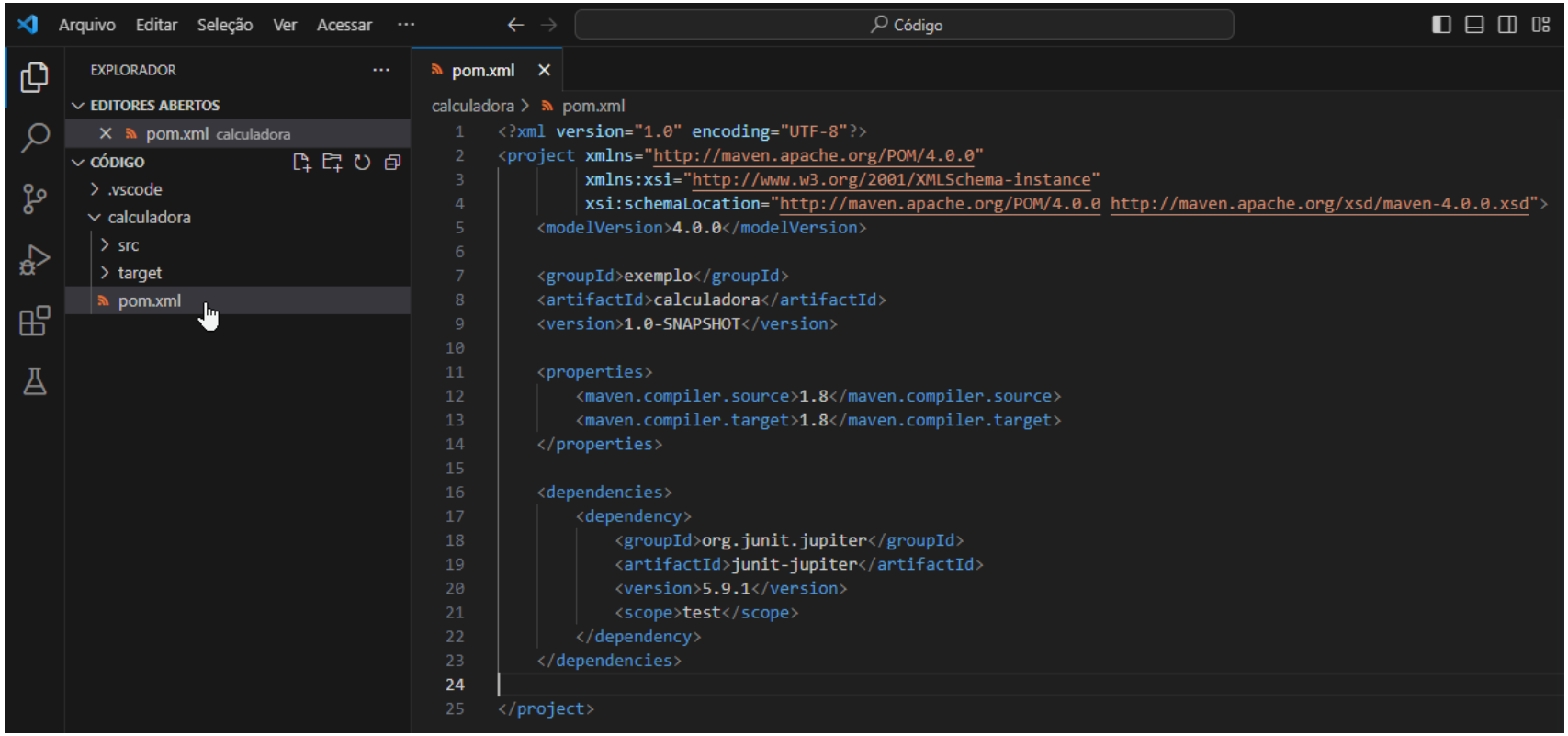




JUnit

Exemplo

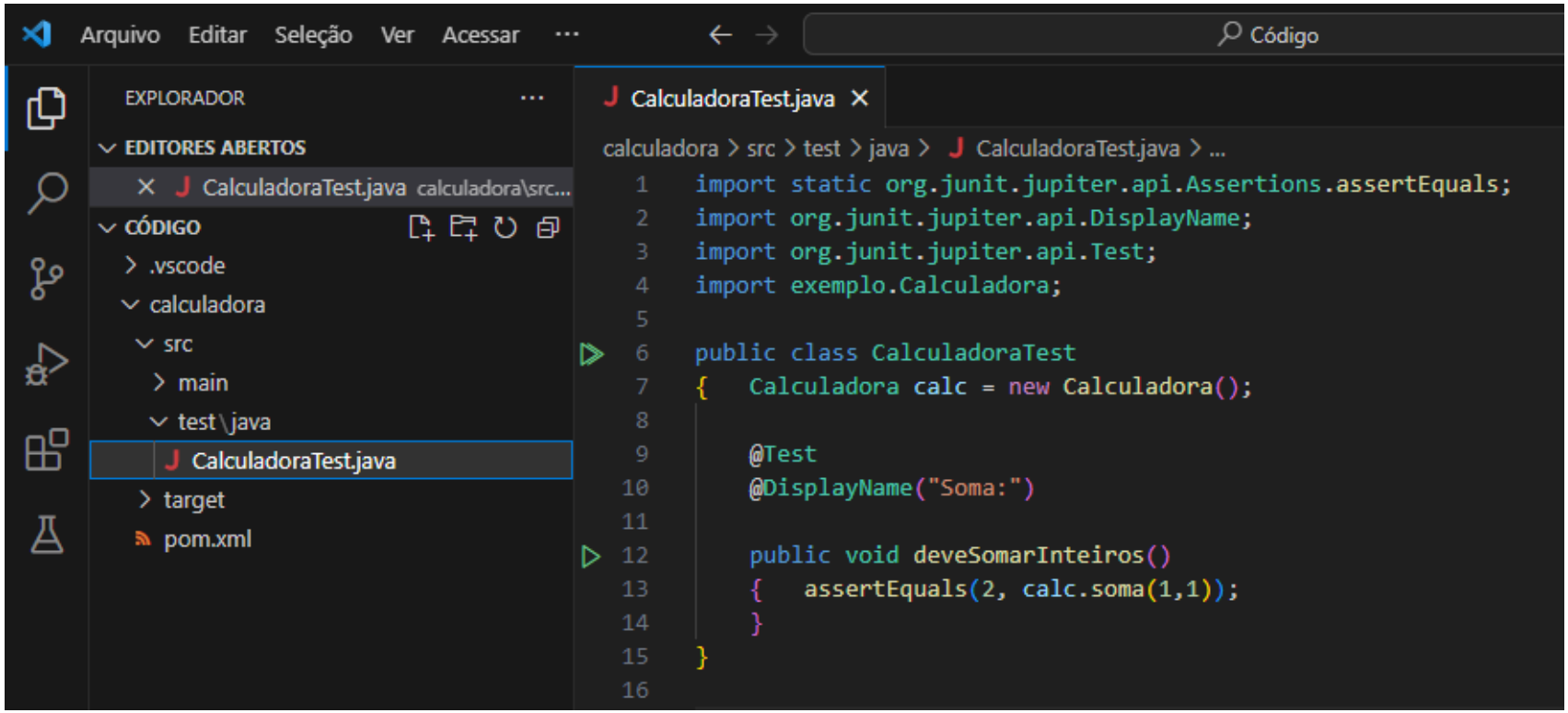
- Abrir o arquivo **pow.xml** e incluir as linhas 16 à 23, conforme mostra a figura abaixo:



JUnit

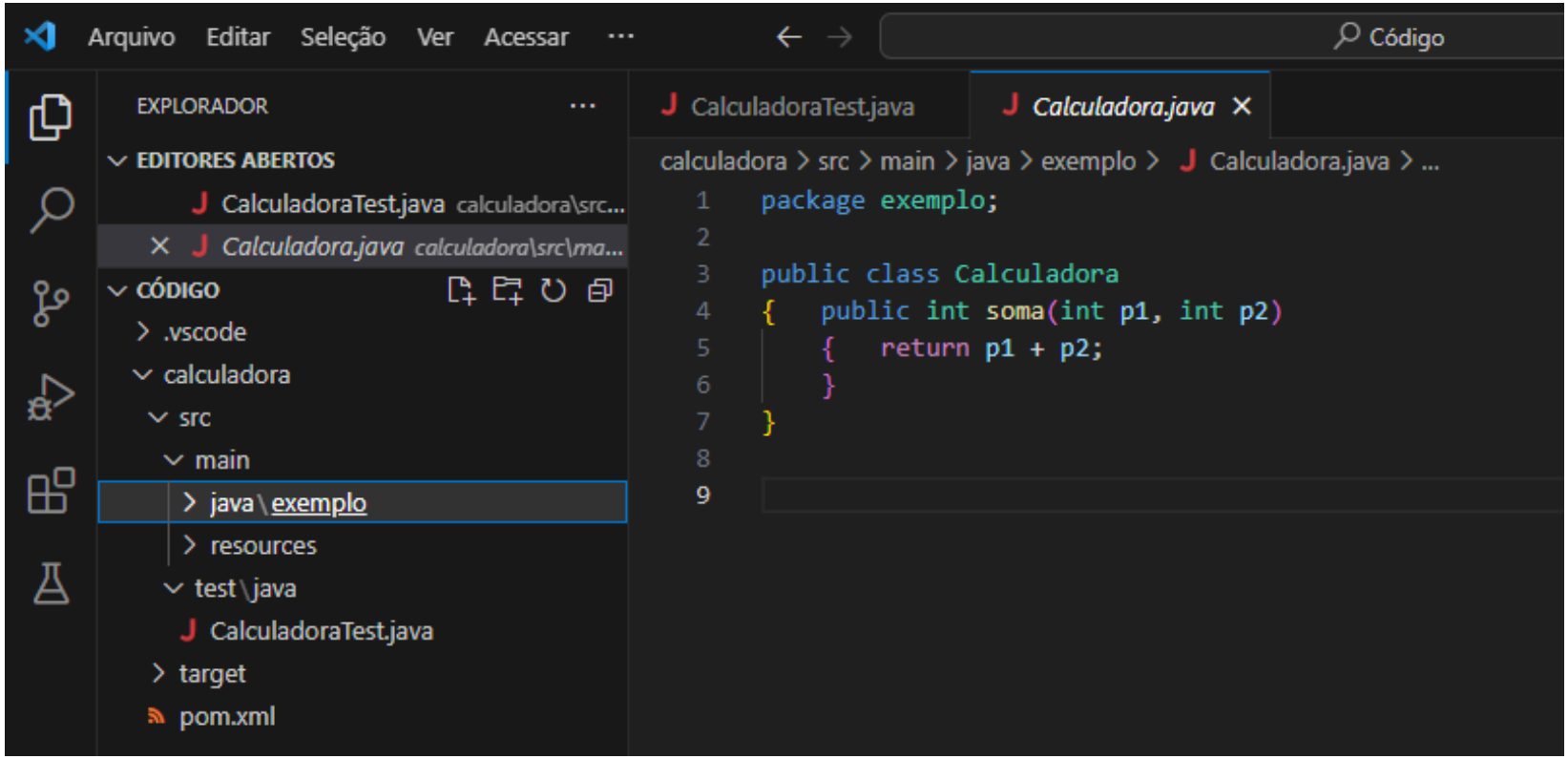
Exemplo

- Criar a classe **CalculadoraTest.java**, conforme mostra a figura abaixo:



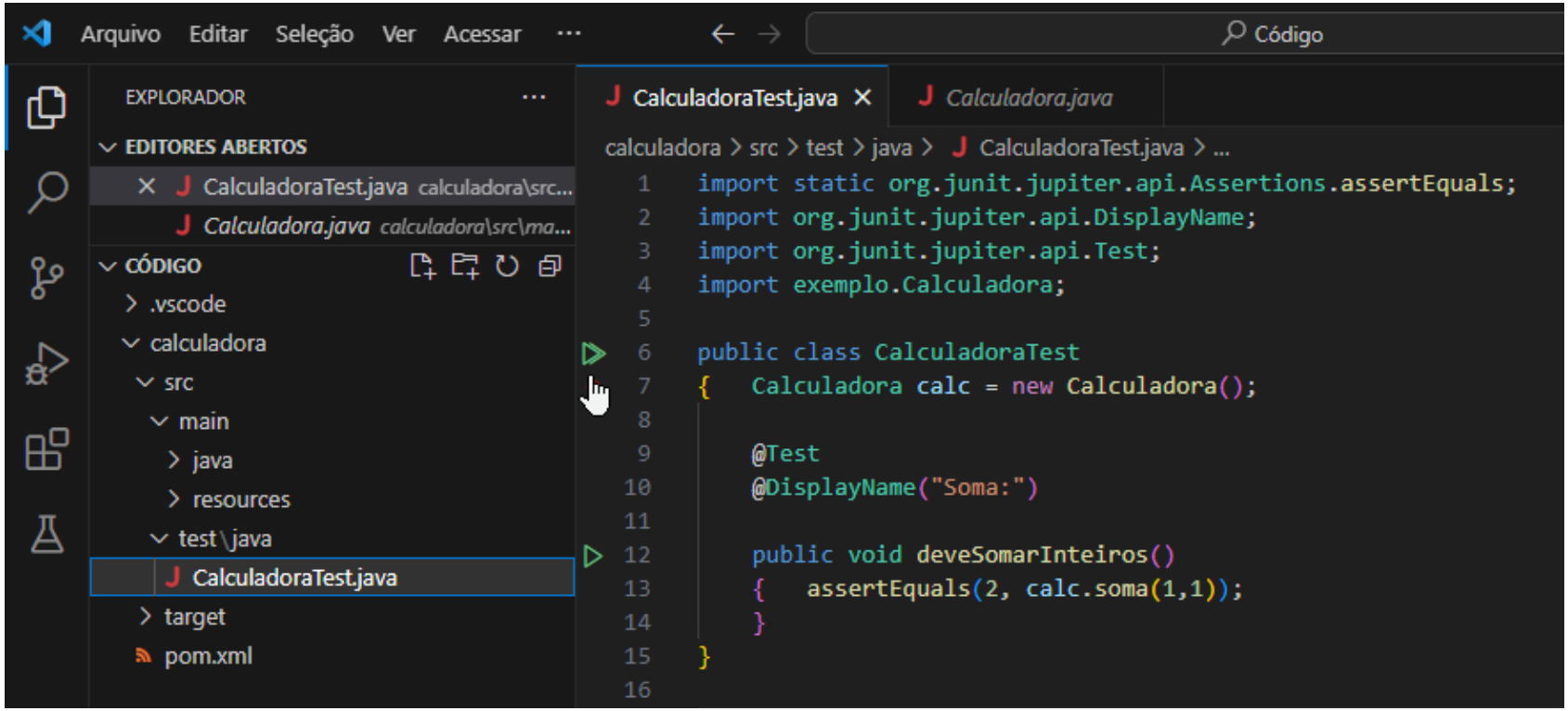
Exemplo

- Criar a classe **Calculadora.java**, conforme mostra a figura abaixo:



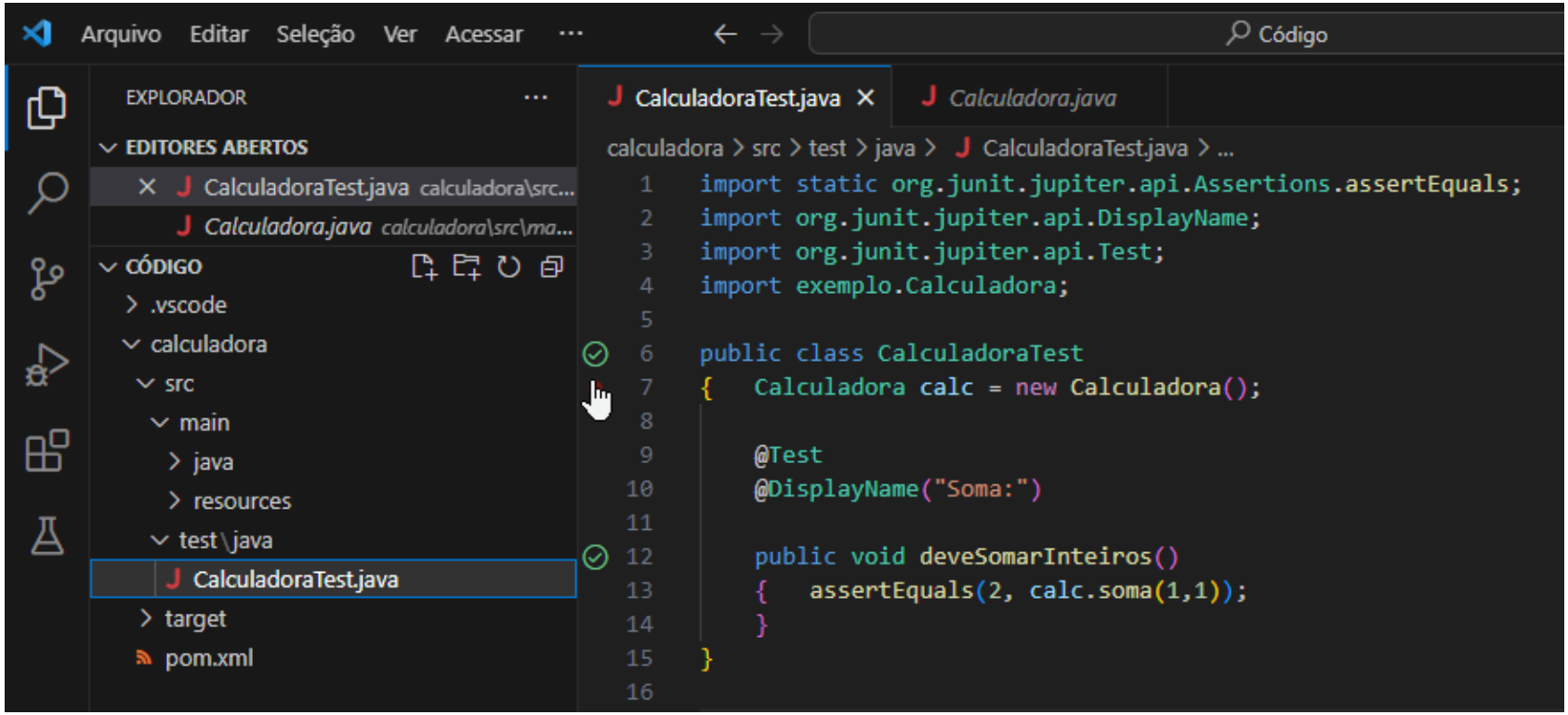
Exemplo

- Na classe **CalculadoraTest.java**, clicar sobre os triângulos verdes (*play*) para realizar os testes, conforme mostra a figura abaixo:



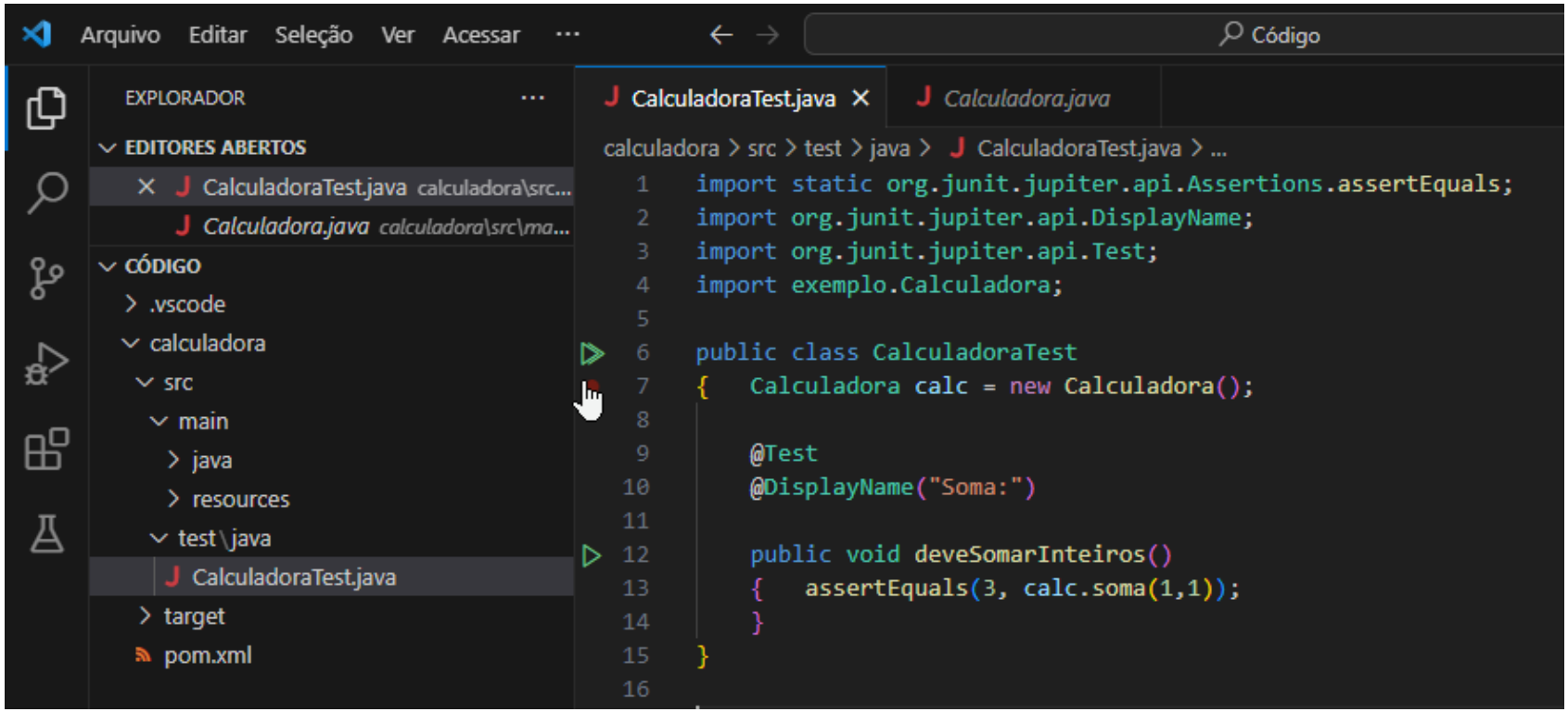
Exemplo

- Na classe **CalculadoraTest.java**, verifique que os testes obtiveram sucesso, conforme mostra a figura abaixo:



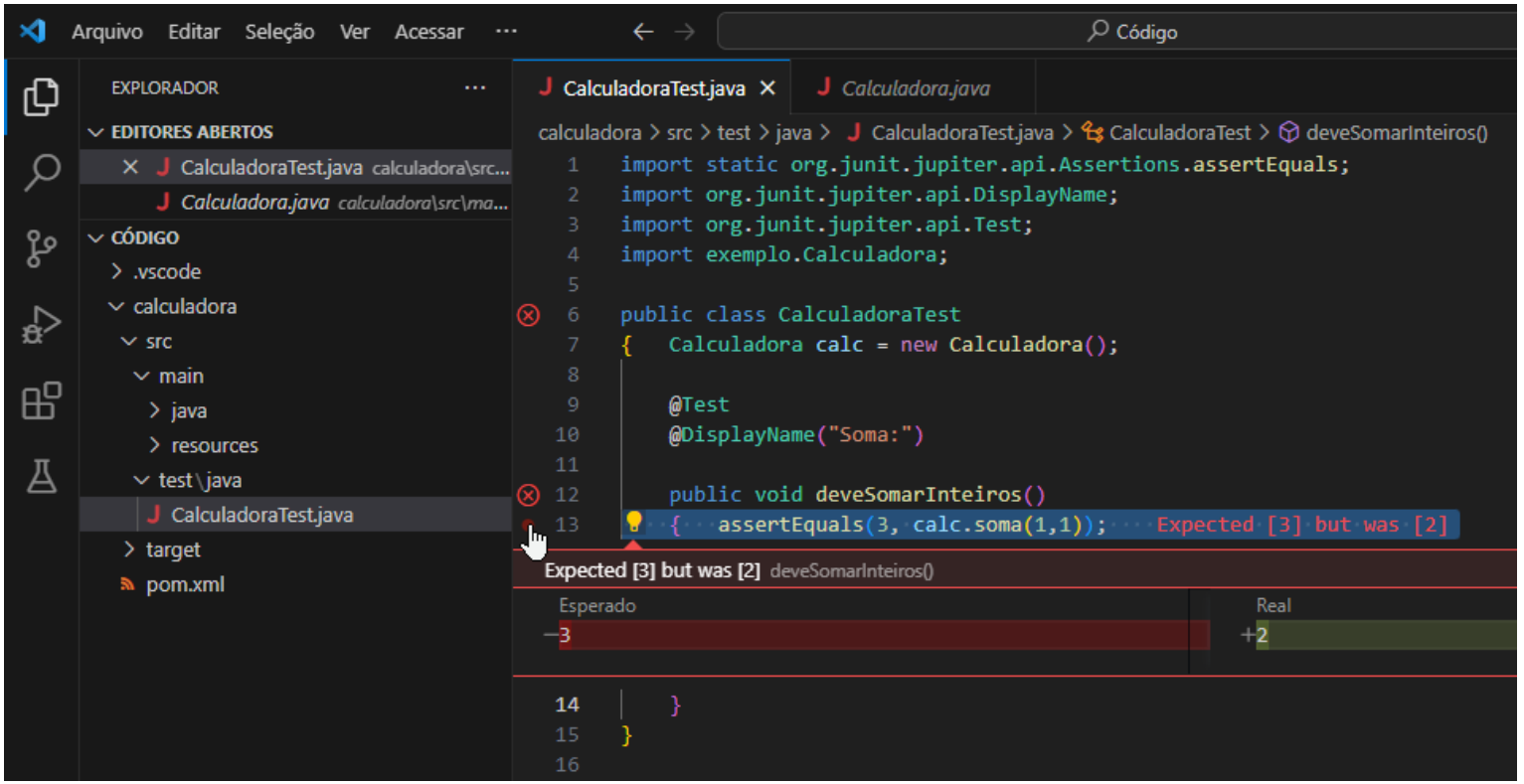
Exemplo

- Na classe **CalculadoraTest.java**, altere o método **assertEquals**, alterando o valor de 2 para 3, verificando o resultado a seguir:



Exemplo

- Na classe **CalculadoraTest.java**, verifique que os testes não obtiveram sucesso, conforme mostra a figura abaixo:



Conclusões

- O ***JUnit*** tem evoluído ao longo dos anos, e existem diferentes versões do *framework*, como o **JUnit 4** e o **JUnit 5**, cada um com melhorias e recursos adicionais;
- O ***JUnit*** é uma ferramenta essencial para garantir a qualidade do código Java e é amplamente adotado na comunidade Java de desenvolvimento.

Exercício 1

- Utilizando TDD, refazer o Exemplo, da página 19 deste material (Aula 19), utilizando o *framework* **JUnit** e alterando a ordenação dos números para decrescente.

Exercício 2

- Utilizando TDD, refazer o Exemplo 2, da página 28 do material da Aula 18 da semana passada, utilizando o *framework* **JUnit**.

Exercício 3

- Utilizando TDD, refazer o Exercício 1, da página 49 do material da Aula 18 da semana passada, utilizando o *framework* **JUnit**.

Exercício 4

- Utilizando TDD, refazer o Exercício 2, da página 50 do material da Aula 18 da semana passada, utilizando o *framework* **JUnit**.

Exercício 5

- Utilizando TDD, refazer o Exercício 3 – Desafio, da página 51 do material da Aula 18 da semana passada, utilizando o *framework* **JUnit**.

Bibliografia Básica

- MILETTO, Evandro M.; BERTAGNOLLI, Silvia de Castro. Desenvolvimento de software II: introdução ao desenvolvimento web com HTML, CSS, javascript e PHP (Tekne). Porto Alegre: Bookman, 2014. E-book. Referência Minha Biblioteca: https://integrada.minhabiblioteca.com.br/#/books/9788582601969
- WINDER, Russel; GRAHAM, Roberts. Desenvolvendo Software em Java, 3ª edição. Rio de Janeiro: LTC, 2009. E-book. Referência Minha Biblioteca: https://integrada.minhabiblioteca.com.br/#/books/978-85-216-1994-9
- DEITEL, Paul; DEITEL, Harvey. Java: how to program early objects. Hoboken, N. J: Pearson, c2018. 1234 p. ISBN 9780134743356.

Continua...

Bibliografia Básica (*continuação*)

- HORSTMANN, Cay S; CORNELL, Gary. Core Java. SCHAFRANSKI, Carlos (Trad.), FURMANKIEWICZ, Edson (Trad.). 8. ed. São Paulo: Pearson, 2010. v. 1. 383 p. ISBN 9788576053576.
- LIANG, Y. Daniel. Introduction to Java: programming and data structures comprehensive version. 11. ed. New York: Pearson, c2015. 1210 p. ISBN 9780134670942.
- TURINI, Rodrigo. Desbravando Java e orientação a objetos: um guia para o iniciante da linguagem. São Paulo: Casa do Código, [2017]. 222 p. (Caelum).

Bibliografia Complementar

- HORSTMANN, Cay. Conceitos de Computação com Java. Porto Alegre: Bookman, 2009. E-book. Referência Minha Biblioteca: https://integrada.minhabiblioteca.com.br/#/books/9788577804078
- MACHADO, Rodrigo P.; FRANCO, Márcia H. I.; BERTAGNOLLI, Silvia de Castro. Desenvolvimento de software III: programação de sistemas web orientada a objetos em java (Tekne). Porto Alegre: Bookman, 2016. E-book. Referência Minha Biblioteca: https://integrada.minhabiblioteca.com.br/#/books/9788582603710
- BARRY, Paul. Use a cabeça! Python. Rio de Janeiro: Alta Books, 2012. 458 p.
  ISBN 9788576087434.

Continua...

Bibliografia Complementar (continuação)

- LECHETA, Ricardo R. Web Services RESTful: aprenda a criar Web Services RESTfulem Java na nuvem do Google. São Paulo: Novatec, c2015. 431 p.
  ISBN 9788575224540.
- SILVA, Maurício Samy. JQuery: a biblioteca do programador. 3. ed. rev. e ampl. São Paulo: Novatec, 2014. 544 p.
  ISBN 9788575223871.
- SUMMERFIELD, Mark. Programação em Python 3: uma introdução completa à linguagem Phython. Rio de Janeiro: Alta Books, 2012. 506 p.
  ISBN 9788576083849.

Continua...

Bibliografia Complementar (continuação)

- YING, Bai. Practical database programming with Java. New Jersey: John Wiley & Sons, c2011. 918 p.
- ZAKAS, Nicholas C. The principles of object-oriented JavaScript. San Francisco, CA: No Starch Press, c2014. 97 p. ISBN 9781593275402.
- CALVETTI, Robson. Programação Orientada a Objetos com Java. Material de aula, São Paulo, 2020.

FIM

Engenharia da Computação – 3ª série

TDD e JUnit
(L1/2 – L2/2)

2023

Horário

Terça-feira: 2 aulas/semana

- L1/2 (09h30min-11h10min): *Prof. Calvetti*;
- L2/2 (11h20min-13h00min): *Prof. Calvetti*;

Exercícios

- Terminar, entregar e apresentar ao professor para avaliação, os exercícios propostos na aula de teoria, deste material.

Bibliografia (apoio)

- LOPES, ANITA. GARCIA, GUTO. Introdução à Programação: 500 algoritmos resolvidos. Rio de Janeiro: Elsevier, 2002.
- DEITEL, P. DEITEL, H. Java: como programar. 8 Ed. São Paulo: Prentice-Hall (Pearson), 2010;
- BARNES, David J.; KÖLLING, Michael. Programação orientada a objetos com Java: uma introdução prática usando o BlueJ . 4. ed. São Paulo: Pearson Prentice Hall, 2009.

Aula 19 – L1/2 e L2/2

FIM