

## **Curso: Testes Automatizados**

### **Aula: 3 – Test Driven Development (Prática)**

Olá! Seja bem-vindo!

Nesse curso, você conheceu estratégias de automação de testes e foi apresentado ao conceito de uma das técnicas de desenvolvimento de software, o TDD (do inglês, Test Driven Development), está lembrado? Então, o objetivo dessa aula é demonstrar como você pode colocar o TDD em prática. Preparado? Vamos, lá!

A partir da metodologia do projeto, do seu contexto e particularidades, é possível definir qual a ferramenta e/ou framework que melhor se adequa à sua aplicação. Vale ressaltar que esta definição não é única, ela pode variar de acordo com o escopo e a complexidade do negócio.

Agora, você vai construir um exemplo, ok? Para isso, você vai utilizar o JUnit, um framework de teste unitário, escrito na linguagem Java. O JUnit possui a flexibilidade de executar de forma independente todos os casos de teste, exibindo seus respectivos resultados, sendo eles aprovado ou reprovado. Sua Interface de Desenvolvimento Eclipse, já possui um plugin para o JUnit, que facilita a implementação.

Para começar a construir o seu modelo, tenha em mente que a alteração do cenário possa mudar de acordo com as regras de negócio de cada sistema e o ciclo do TDD permanecerá o mesmo. Você precisa desenvolver primeiro o teste, para então, escrever o código e assim refatorá-los, em seguida. Tudo certo até aqui?

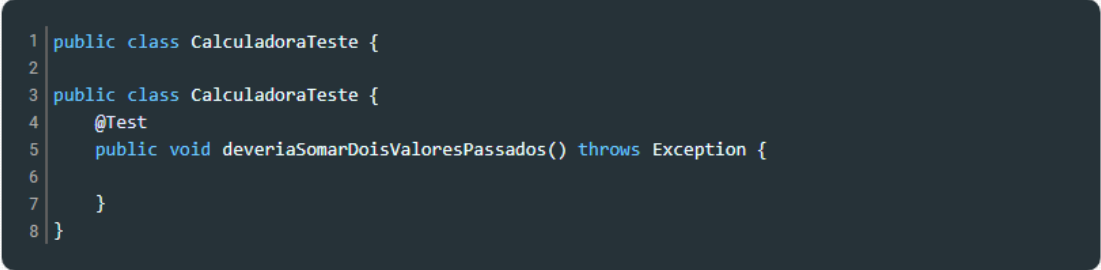
Agora, como um exemplo inicial, imagine que você precisa criar uma classe “Calculadora”, que tem as funções de somar, subtrair, multiplicar e dividir. Como você poderia desenvolvê-la aplicando o TDD?

A primeira dica é: você enquanto desenvolvedor, não deve pensar em como ele implementará a calculadora, mas em como ele irá desenvolver um teste que valide cada funcionalidade da calculadora. Para isto, você precisa criar uma classe de teste, importar as bibliotecas do JUnit e indicar com a anotação “@Test” cada um

dos seus métodos de teste, que por sua vez, só serão executados se possuírem essa anotação. As anotações indicam se um método é de teste ou não, se um método deve ser executado antes da classe e/ou depois da classe, se o teste deve ou não ser ignorado e se a classe em questão é uma suíte de teste, ou seja, se a partir desta classe é disparada a execução das demais classes de teste. Até aqui, tudo certo?

Outro ponto importante, é que o nome do método deve indicar exatamente o que ele executará, assim você facilita a compreensão e o código fica mais simples. Essas dicas que você acabou de aprender estão representadas na seguinte imagem: Perceba que a classe de teste foi criada, obedecendo ao padrão de ter um nome específico relacionado à funcionalidade em questão, acrescido da palavra “Teste”, por isso a mesma chama-se “CalculadoraTeste”. Já o método recebe o nome da função que ele deve desempenhar, ou seja, este deve somar dois números que serão inseridos pelo usuário.

```
public class CalculadoraTeste {  
public class CalculadoraTeste {  
    @Test  
    public void deveriaSomarDoisValoresPassados() throws Exception {  
    }  
}
```

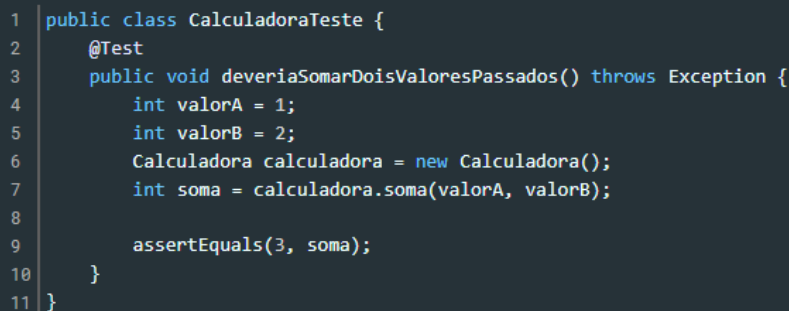
A screenshot of a code editor with a dark background and light-colored text. The code is the same as the one in the previous block, but with line numbers 1 through 8 on the left side.

```
1 public class CalculadoraTeste {  
2  
3 public class CalculadoraTeste {  
4     @Test  
5     public void deveriaSomarDoisValoresPassados() throws Exception {  
6  
7     }  
8 }
```

Você está lembrado do que aprendeu sobre testes em uma abordagem de ciclo, com três pontos básicos? Ótimo, pois essa é a lógica. Se você executar este teste, ele falhará, pois ainda não existe nada implementado. A partir daí, você precisa adicionar o código de teste que validará o cenário em questão. Esse

exemplo que você acabou de conferir, está representado na imagem a seguir: Agora, acrescentamos um código ao método, que insere dois valores e cria o objeto “Calculadora” para que a soma seja efetuada. Logo em seguida, temos a validação, por meio da biblioteca de asserções, que verificará se a soma está correta.

```
public class CalculadoraTeste {  
    @Test  
    public void deveriaSomarDoisValoresPassados() throws Exception {  
        int valorA = 1;  
        int valorB = 2;  
        Calculadora calculadora = new Calculadora();  
        int soma = calculadora.soma(valorA, valorB);  
  
        assertEquals(3, soma);  
    }  
}
```

A screenshot of a code editor with a dark background and light-colored text. The code is the same as the one in the previous block, but with line numbers 1 through 11 on the left side.

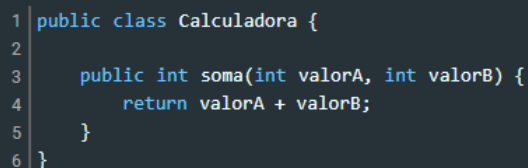
```
1 public class CalculadoraTeste {  
2     @Test  
3     public void deveriaSomarDoisValoresPassados() throws Exception {  
4         int valorA = 1;  
5         int valorB = 2;  
6         Calculadora calculadora = new Calculadora();  
7         int soma = calculadora.soma(valorA, valorB);  
8  
9         assertEquals(3, soma);  
10    }  
11 }
```

Perceba que no contexto deste exemplo, temos a adição de um código que valida a soma de dois números. Se você o executar novamente, entretanto, o mesmo ainda vai falhar, visto que o código da funcionalidade de soma ainda não está pronto, ou seja, a classe “**Calculadora**” ainda não foi criada, pois desenvolvemos primeiramente o teste, com o intuito de refletir sobre os fluxos de validações.

Agora, você precisa criar o código que faça o teste passar, pois o objeto “Calculadora” foi criado e instanciado na classe de teste, mas a classe “Calculadora”, que é a implementação da funcionalidade em si, ainda não está criada, lembra? Na

imagem a seguir, temos a representação desta classe, que caracteriza o que foi especificado no início: a soma de dois valores. Feito isso, o teste, enfim, passará.

```
public class Calculadora {  
  
    public int soma(int valorA, int valor B) {  
        return valorA + valorB;  
    }  
}
```

A code snippet displayed in a dark-themed editor with line numbers 1 through 6 on the left. The code is the same Java class definition as shown in the previous block.

```
1 public class Calculadora {  
2  
3     public int soma(int valorA, int valorB) {  
4         return valorA + valorB;  
5     }  
6 }
```

Como este exemplo é bem trivial, não existe aqui alguma refatoração interessante, porém o processo de refatorar o código é sempre necessário quando o sistema começa a crescer ou quando as funcionalidades forem modificadas, de maneira a garantir uma melhor padronização e manutenibilidade do código desenvolvido.

Nesta aula, você aprendeu uma aplicação prática de TDD. E para uma melhor fixação do conteúdo, é fundamental que você coloque em prática o que aprendeu na aula. Implementando, por exemplo, os testes para as funções de subtração, multiplicação e divisão.

Fique à vontade para explorar as possibilidades desta abordagem e proceder da melhor forma que se adeque às suas necessidades. Existem vários frameworks de teste para execução de TDD, dependendo das tecnologias que são usadas no projeto. No decorrer deste curso, você aprenderá diversos modelos.

Por hoje é só!

Bons estudos!

**Referências:**

BECK, Kent. Test-Driven Development by Example. Addison-Wesley Professional, 2002;

ASTELS, David. Test-Driven Development – A practical Guide. Prentice Hall PTR, 2003;

KERIEVSKY, J. “Refactoring to Patterns”. Addison-Wesley, 2004.