



TESTES

Material Complementar



Marianne Salomão

Cloud Engineer na IBM Brasil, desenvolvedora de software full - stack e instrutora de TI.



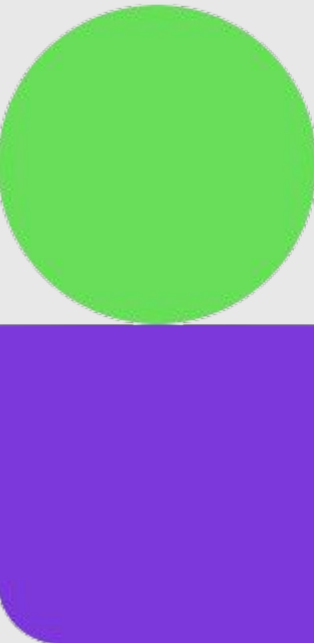
linkedin.com/in/mariannesalomao



github.com/mariannesalomao



Índice

- + [Conceitos de Testes de Software](#)
 - + [Teste Dinâmico e Estático](#)
 - + [Testes X Depuração](#)
 - + [Testes Unitários](#)
 - + [Testes de Integração](#)
 - + [Jasmine](#)
 - + [Cucumber](#)
 - + [Jest](#)
- 

Introdução

Tornou-se padrão que empresas busquem testar seus produtos em diferentes etapas do desenvolvimento até a entrega ao usuário final, devido às vantagens que os testes nos trazem, possibilitando a identificação de erros e garantindo a confiabilidade no uso do software, que deve ser um produto com qualidade de funcionamento.

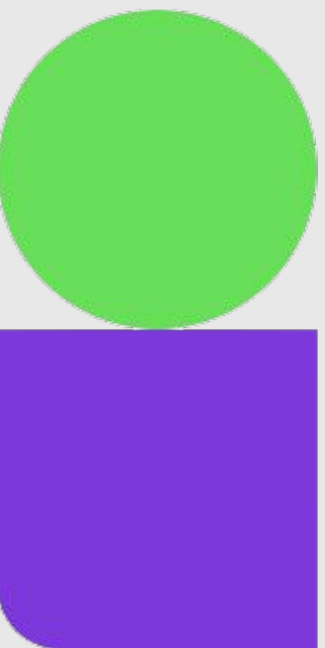


01. Conceitos de Testes

O teste de software é uma maneira de avaliar a qualidade da aplicação e reduzir o risco de falha em operação.

Testar não consiste apenas em executar testes (executar o software e verificar os resultados). Executar testes é apenas umas das atividades.

Planejamento, análise, modelagem e implementação dos testes, relatórios de progresso, resultado e avaliação da qualidade, também são partes de um processo de testes.



01. Conceitos de Testes

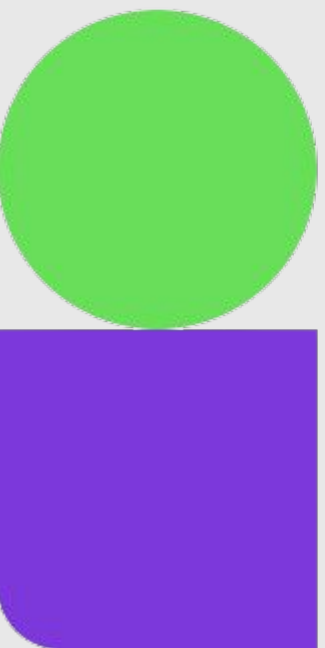
Objetivos de se testar um software:

- Avaliar os produtos de trabalho (requisitos, histórias de usuário, modelagem e código);
- Verificar se todos os requisitos especificados foram atendidos;
- Validar se o objeto de teste está completo e funciona como os usuários e outras partes interessadas esperam;
- Criar confiança no nível de qualidade do objeto de teste;
- Evitar defeitos;
- Encontrar falhas e defeitos;
- Fornecer informações suficientes às partes interessadas para permitir que elas tomem decisões, especialmente em relação ao nível de qualidade do objeto de teste;
- Reduzir o nível de risco de qualidade de software inadequada (ex.: falhas não detectadas anteriormente que ocorrem em produção);
- Cumprir com requisitos ou normas contratuais, legais ou regulamentares, e/ou verificar o cumprimento do objeto de teste com tais requisitos ou normas.

01. Conceitos de Testes

Por que o teste é necessário?

- Para reduzir o risco de falhas durante a operação. Isso se dá a partir de testes rigorosos de componentes e sistemas, além do uso de documentação adequada;
- Para contribuir com a qualidade dos componentes ou sistemas. Isso se dá a partir da detecção e correção de defeitos;
- Para atender aos requisitos contratuais/legais ou aos padrões específicos do setor a que se destinam;



01. Conceitos de Testes

Uma pessoa pode cometer **um erro (engano)**, que pode levar à introdução de um defeito no código do software ou em algum outro produto de trabalho relacionado.

Um erro que leva à introdução de um defeito em um produto de trabalho pode acionar outro erro que leva à introdução de um defeito em um outro produto de trabalho relacionado.

Exemplo: uma **pessoa** levanta um requisito de forma incompleta, ambígua, incorreta.

Caracteriza-se um erro.

O projeto de software daquele requisito é realizado de forma incorreta.

Caracteriza-se um defeito. O desenvolvedor cria um código incorreto a partir de um projeto incorreto/errado. Caracteriza-se um erro. O software gerado apresentará defeito devido ao código incorreto.

01. Conceitos de Testes

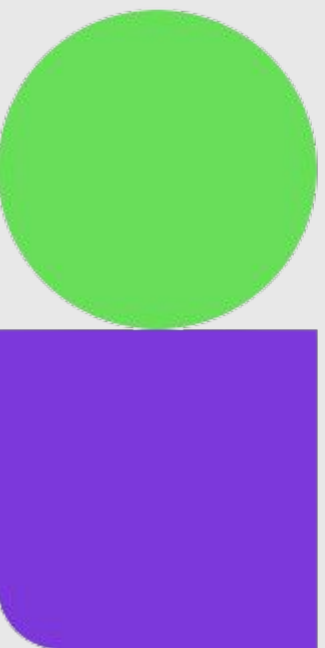
A execução de um código defeituoso, pode causar uma falha, mas não necessariamente em todas as circunstâncias.

Exemplo: bug do milênio, que ocorreu em uma circunstância específica.

Nem todos os resultados de testes inesperados são falhas.

Podem ocorrer **falsos positivos** (relatos de defeitos que não são defeitos) ou também **falsos negativos** (testes que não detectam defeitos que deveriam ser detectados).

Falsos positivos podem ocorrer devido a erros na forma como os testes foram executados ou devido a defeito nos dados, no ambiente ou em outro produto de teste ou por outros motivos.

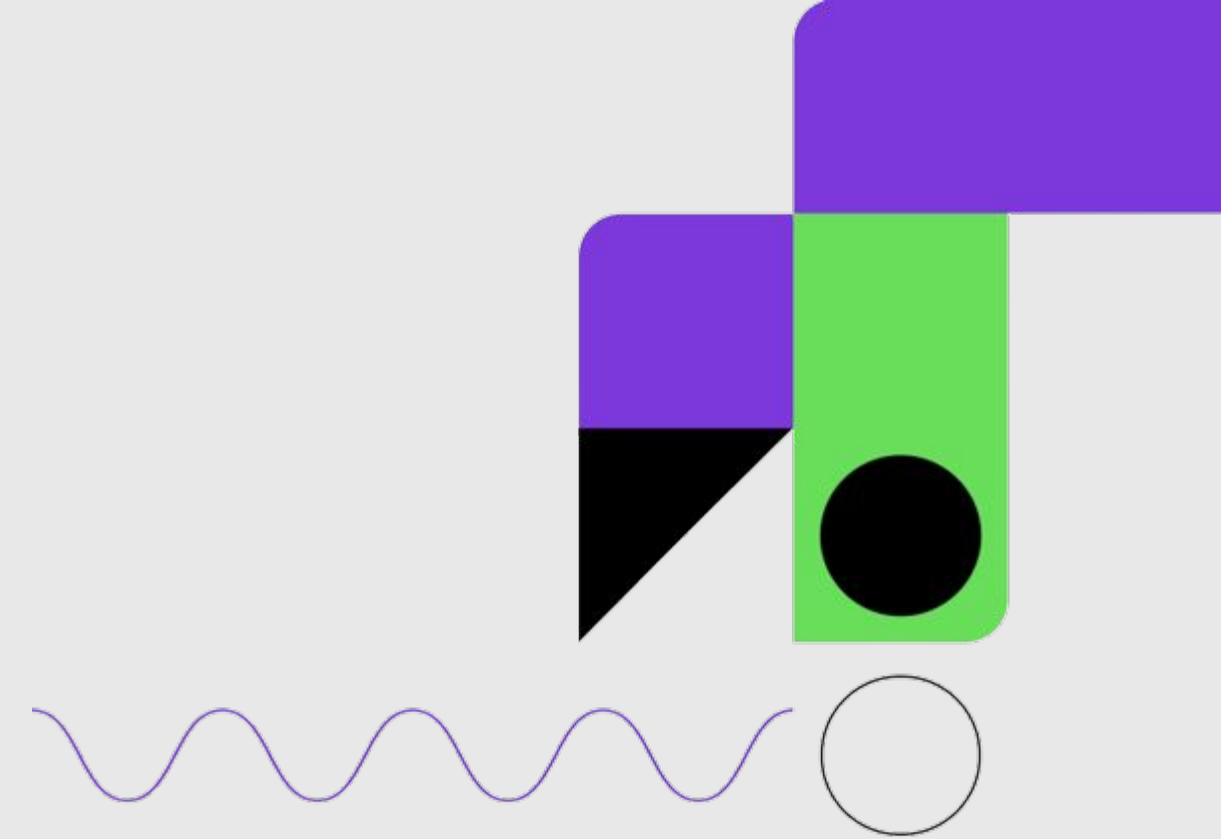


02. Testes dinâmicos e estáticos

O teste **dinâmico** refere-se à **execução de um componente ou sistema**.

Já os demais, que **não envolvem a execução do componente ou sistema**, são conhecidos como testes **estáticos**.

Para cada uma destas categorias existem inúmeros tipos de processos.





03. Teste X Depuração

As atividades não são sinônimas.

O que ocorre é que em alguns casos, os testadores são responsáveis pelo teste inicial e pelo teste de confirmação final, enquanto os desenvolvedores fazem a depuração e o teste do componente associado.

Há ainda os contextos ágeis, onde os testadores podem ser envolvidos na depuração e no teste de componente.



03. Teste X Depuração

Em resumo, há uma diferenciação para **TESTE e DEPURAÇÃO**:

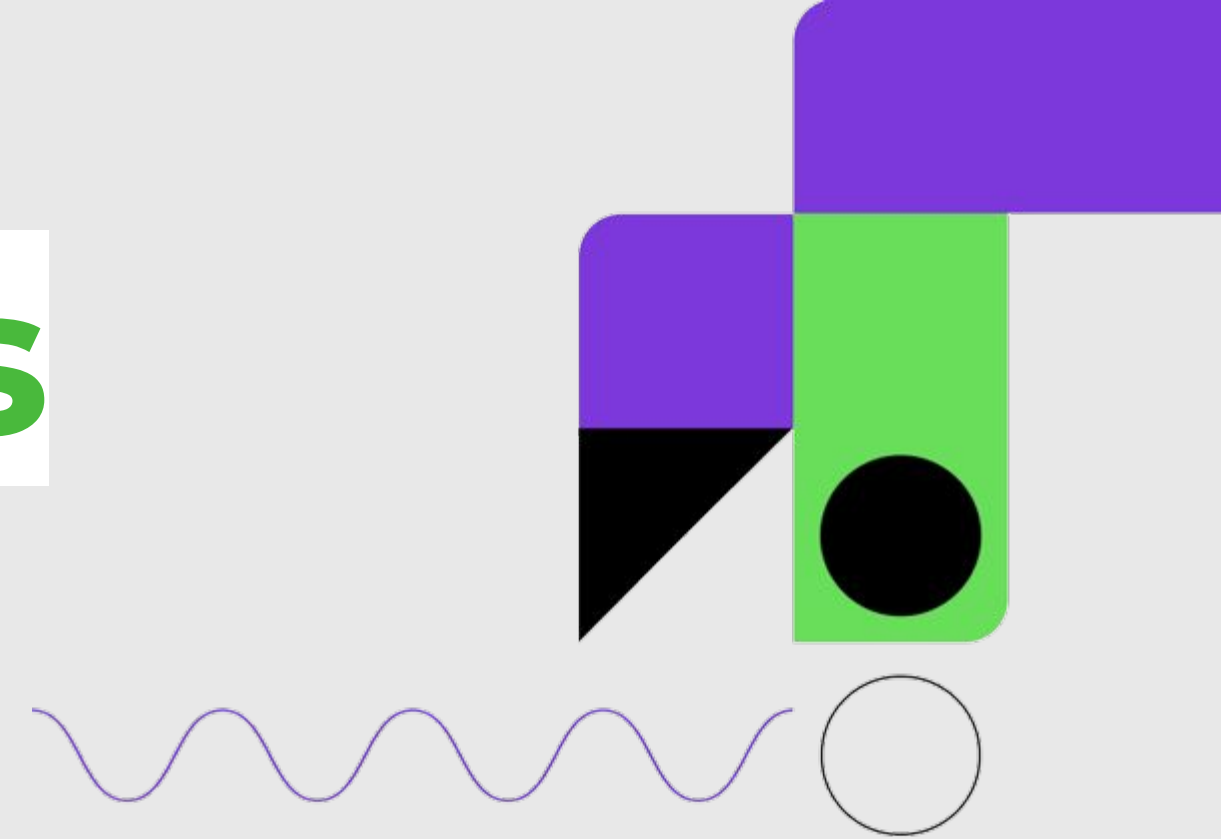
Teste → A execução dos testes pode mostrar falhas causadas por defeitos no software.

Depuração → Atividade de desenvolvimento que localiza, analisa e corrige esses defeitos.

04. Testes Unitários

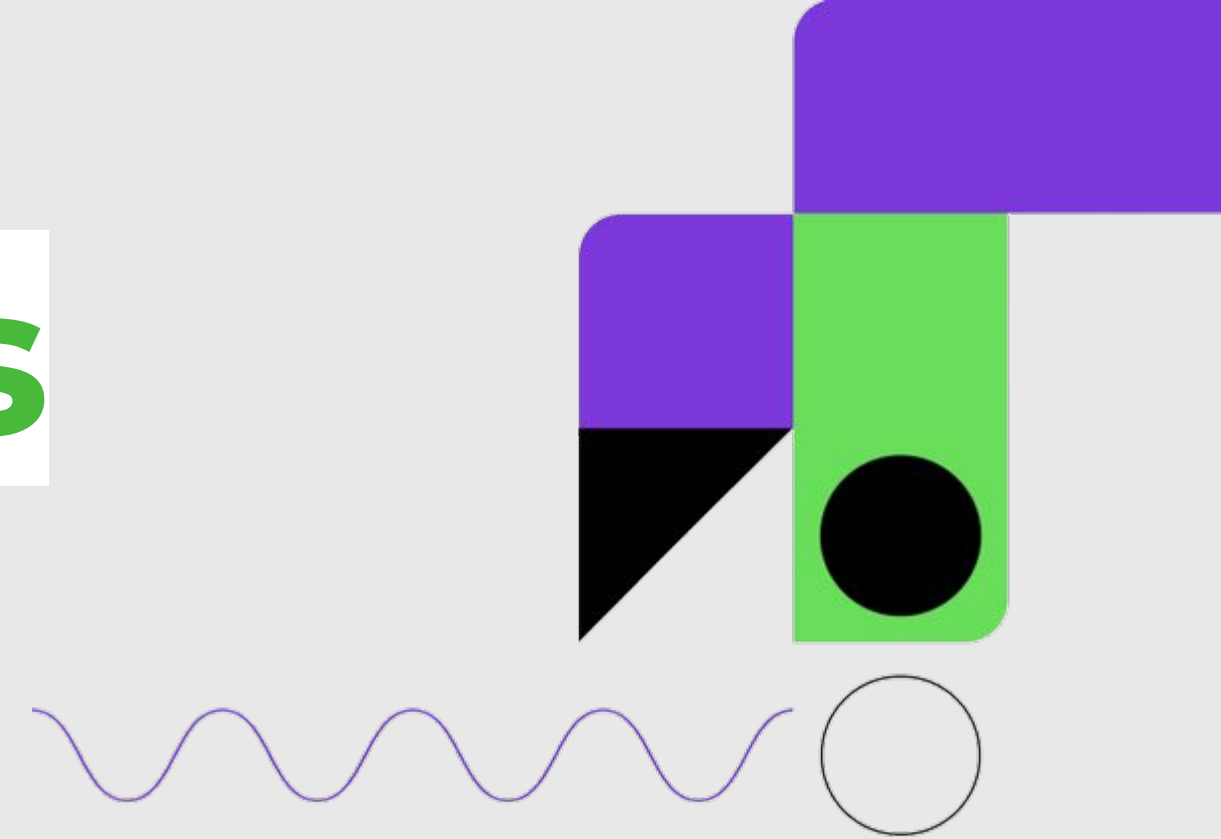
Esses testes são feitos em um **nível muito baixo** (próximo ao código fonte) do projeto, por isso, geralmente quem os realiza são os programadores envolvidos no projeto.

Geralmente são realizados de forma isolada do restante do sistema, visto que tem por objetivo assegurar a qualidade das unidades de forma individual e não o sistema como um todo. Podemos entender como “**unidade**” as menores partes do nosso sistema, ou seja, métodos e funções das classes ou pacotes utilizados no projeto.



04. Testes Unitários

Esses testes têm como objetivo verificar as menores unidades isoladamente, garantindo que a lógica de cada uma delas está correta e que funciona conforme o esperado. Geralmente têm um baixo custo para automatização e podem ser executados rapidamente, inclusive por um servidor de integração contínua.

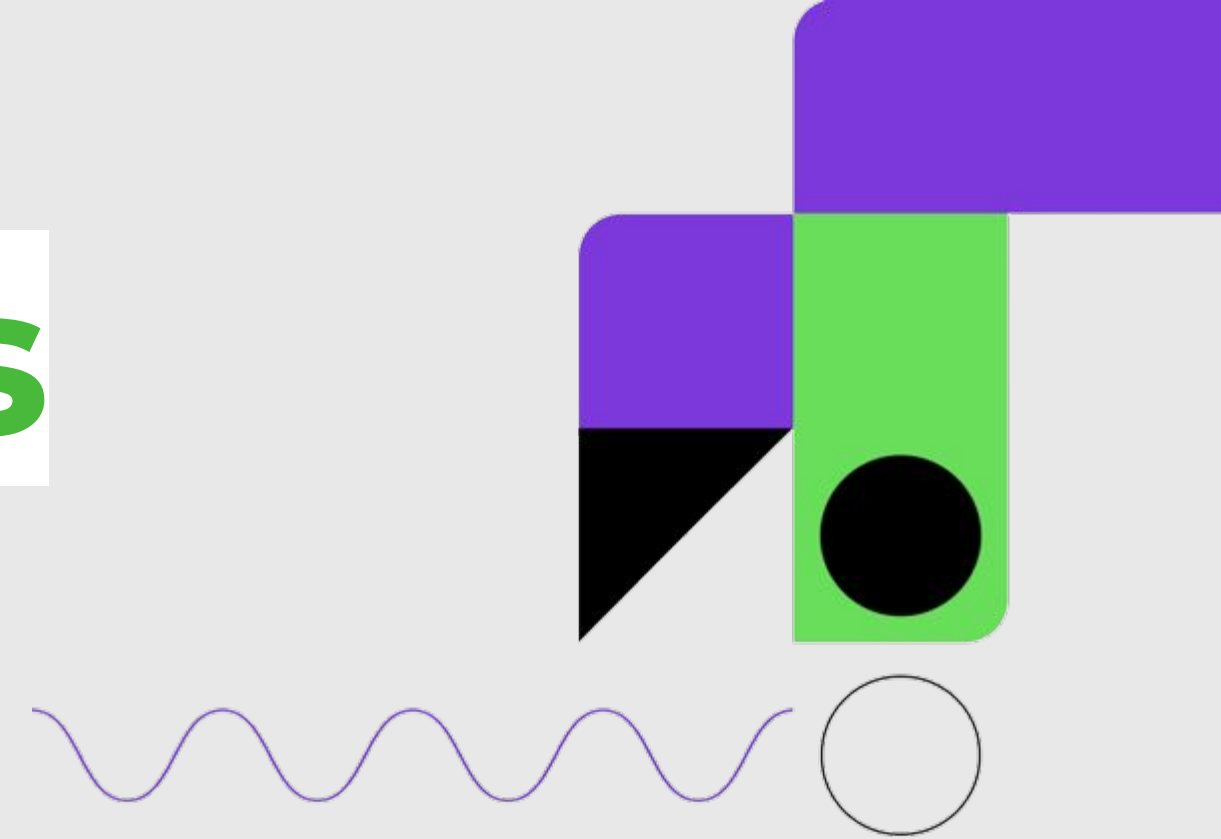


04. Testes Unitários

Para um desenvolvedor, a prática é a seguinte:

Se você está projetando seu software, e "diz" que uma classe dos seus códigos é uma **unidade**, então testar essa classe é justamente fazer o **teste unitário**.

Testar cada classe do seu projeto é fazer o teste unitário do projeto. É algo que os requisitos da sua empresa, ou o próprio desenvolvedor define.



05. Testes de Integração

Os testes unitários buscam verificar se elementos individuais (unidades) do sistema estão corretos, mas isso **não nos garante que a interação entre essas unidades ocorrerá da forma que planejamos**. É nesse momento que utilizamos os **testes de integração**.

Geralmente eles são mais complexos para serem desenvolvidos e mais lentos ao ser executados, pois ao contrário dos testes unitários, nosso objetivo não será testar a lógica nas menores unidades do sistema, mas sim as funcionalidades inteiras, o conjunto funcionando em simultâneo e entregando o resultado esperado.

Por isso, o ideal é realizar testes de integração após a realização dos testes unitários, garantindo que as unidades estão corretas individualmente e também que funcionam em conjunto.



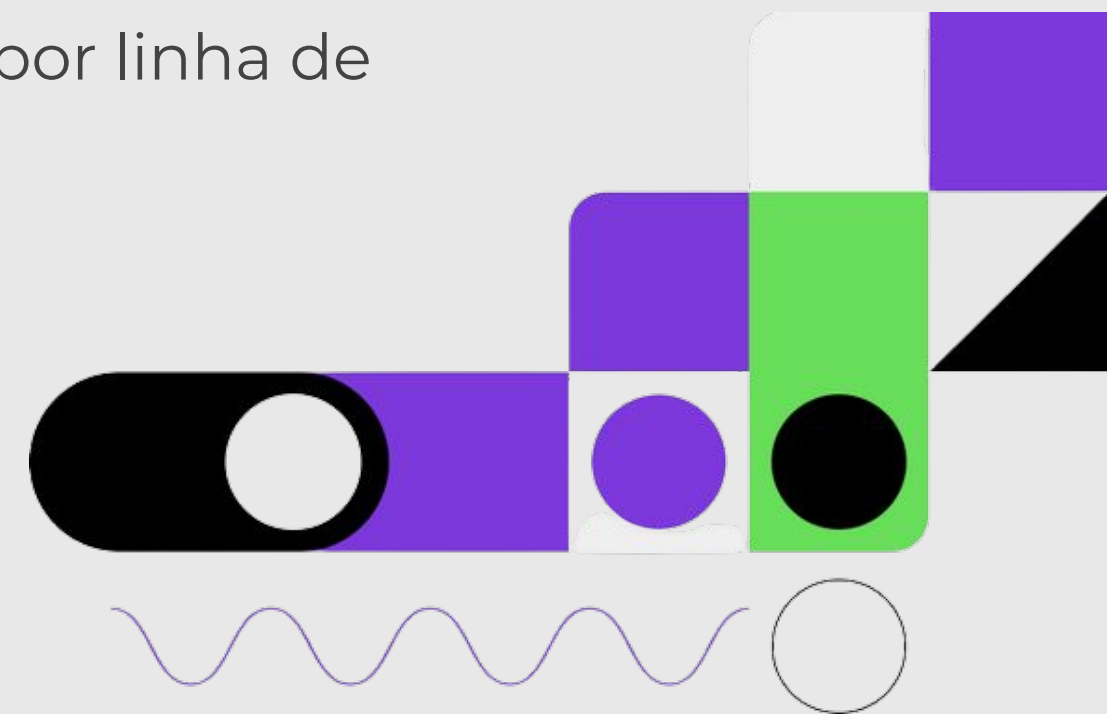
06. Jasmine

O Jasmine é um **framework BDD**, ou seja, para desenvolvimento guiado por testes. Ele é utilizado para criação de testes em JavaScript.

Ele utiliza os conceitos do BDD (Behavior Driven Development), que seriam testes guiados por comportamento, que permite a criação de testes intuitivos e de fácil compreensão.

Suas vantagens incluem ser rápido, e não possuir dependências externas, incluindo por padrão tudo o que é necessário para testar uma aplicação.

Possui a capacidade de executar os testes diretamente no navegador, ou por linha de comando no terminal, além de ser de fácil instalação e configuração.



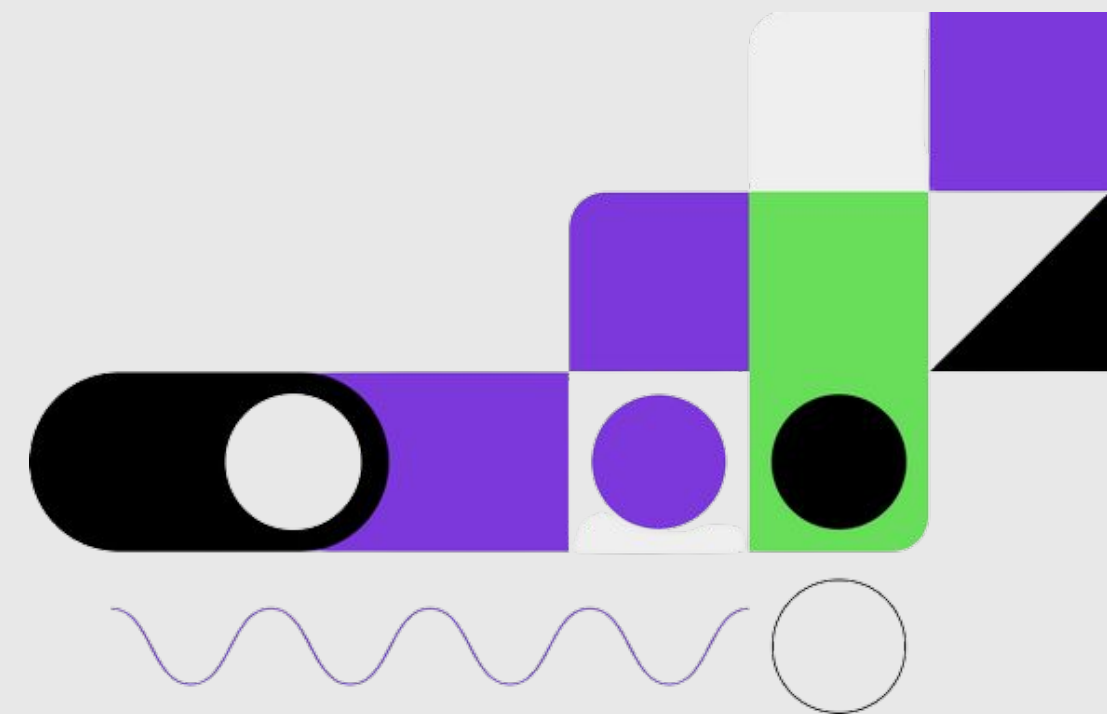
06. Jasmine

Vamos iniciar um projeto com testes em Jasmine. Instale o Jasmine com o comando abaixo:

```
npm install -g jasmine
```

Agora para iniciar os testes precisaremos da função **describe()** que recebe como parâmetros uma string que é o título para o conjunto de testes e uma função **function()** que é o bloco de código que implementa os testes:

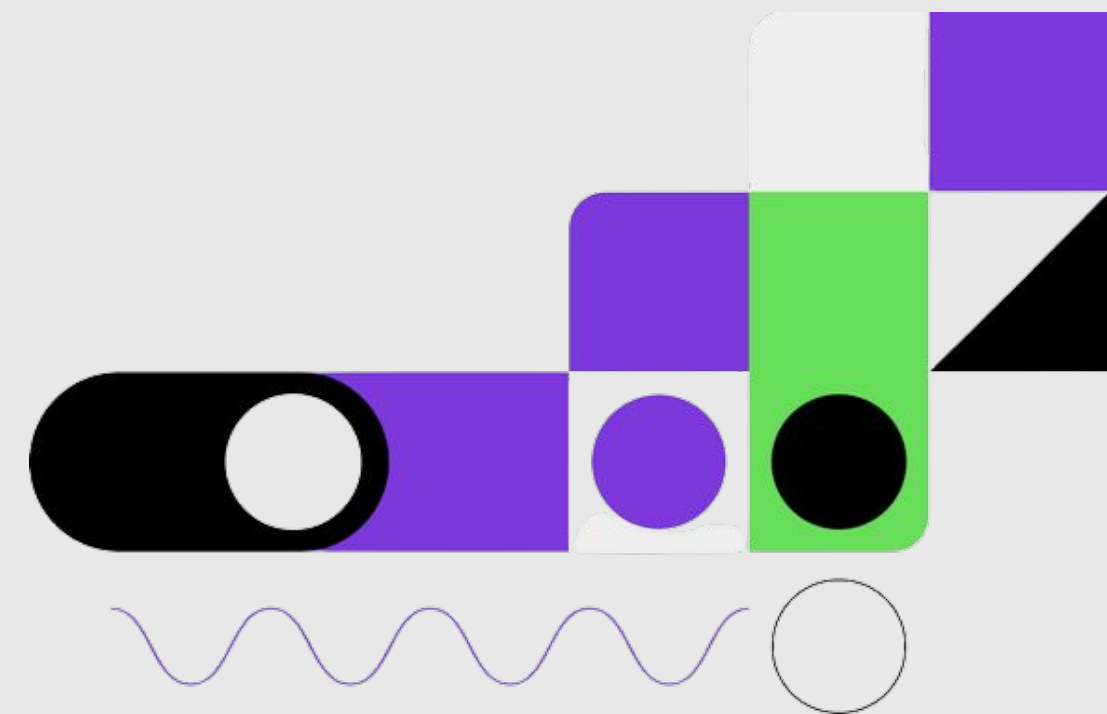
```
describe("Testes", function() {  
  
  })
```



06. Jasmine

Dentro da **function()** teremos a função **it()** que também recebe como parâmetros uma string e uma função `function()` que servem, respectivamente, para dar um título ao teste específico, e implementar o teste específico.

```
describe("Testes", function() {  
  it("Teste1", function() {  
  
  })  
})
```

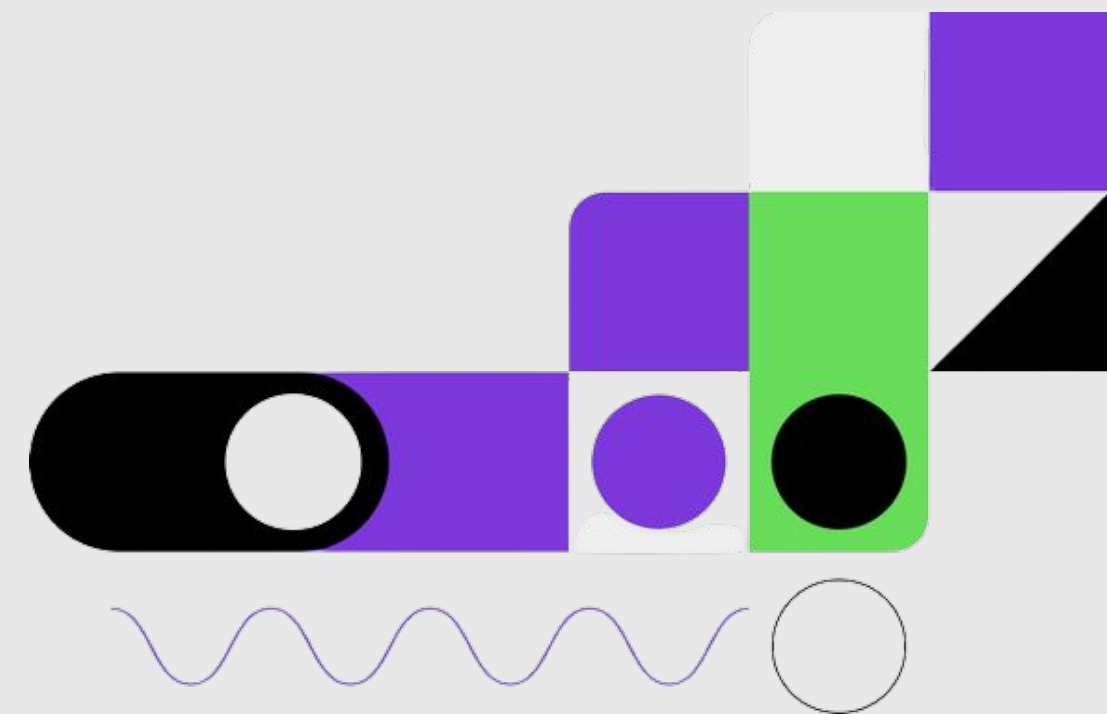


06. Jasmine

Dentro de **it()** teremos os resultados esperados daquele determinado teste, para defini-los utilizaremos a função **expect()** que recebe como parâmetros uma afirmação que pode ser verdadeira ou falsa.

Aplicada à essa afirmação temos uma condição, uma expectativa a qual essa afirmação precisa atender, por exemplo: **toBe()(ser algo)**, nesse caso, para exemplificar será **true**.

```
describe("Testes", function() {  
  it("Teste1", function() {  
    expect(true).toBe(true)  
  })  
})
```



06. Jasmine

Exemplo de teste:

```
let testes = require('./testes.js')

/* Importa o arquivo com as funções e atribui
à variável testes. */

describe('testes', function() {

    // Testando a function reverse_string(str)

    it('returns reverse string', function() {

        let str = 'Jasmine'

        expect(testes.reverse_string(str)).toEqual('enimsaJ')

    })

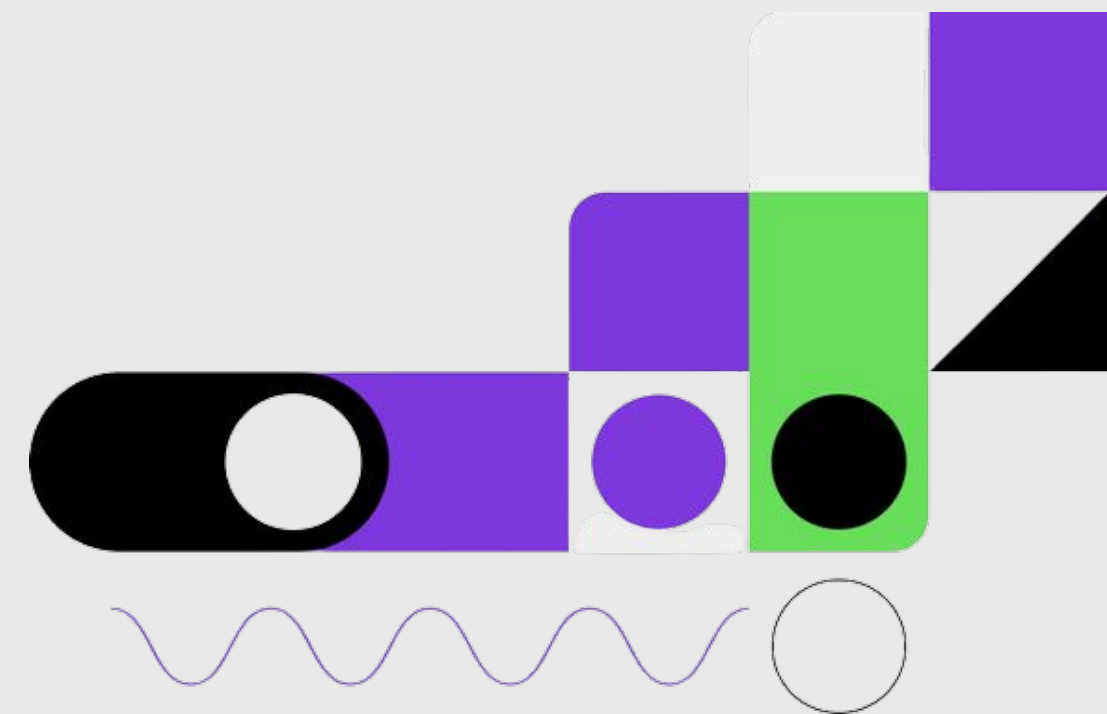
    it('verify if a string contain a substring', function() {

        let str = 'Jasmine'

        expect(testes.reverse_string(str)).toContain('enim')

    })

})
```





07. Cucumber



07. Cucumber

Essa ferramenta ajuda a aplicar o Behavior-Driven Development, **BDD**.

Para começarmos a entender como funciona o Cucumber, iremos iniciar outro projeto.



07. Cucumber

Instale o cucumber pelo comando abaixo:

```
npm install cucumber --save-dev
```

Na próxima página terá um exemplo de código com Cucumber



07. Cucumber

```
const assert = require('assert');

const { Given, When, Then } = require('cucumber');

Given('Hoje é segunda', function () {

  // Aqui vai uma expressão

  return 'Pendente';

});

When('Eu perguntei se hoje é sexta', function () {

  // Aqui vai uma expressão

  return 'Pendente';

});

Then('Eu devo dizer {string}', function (string) {

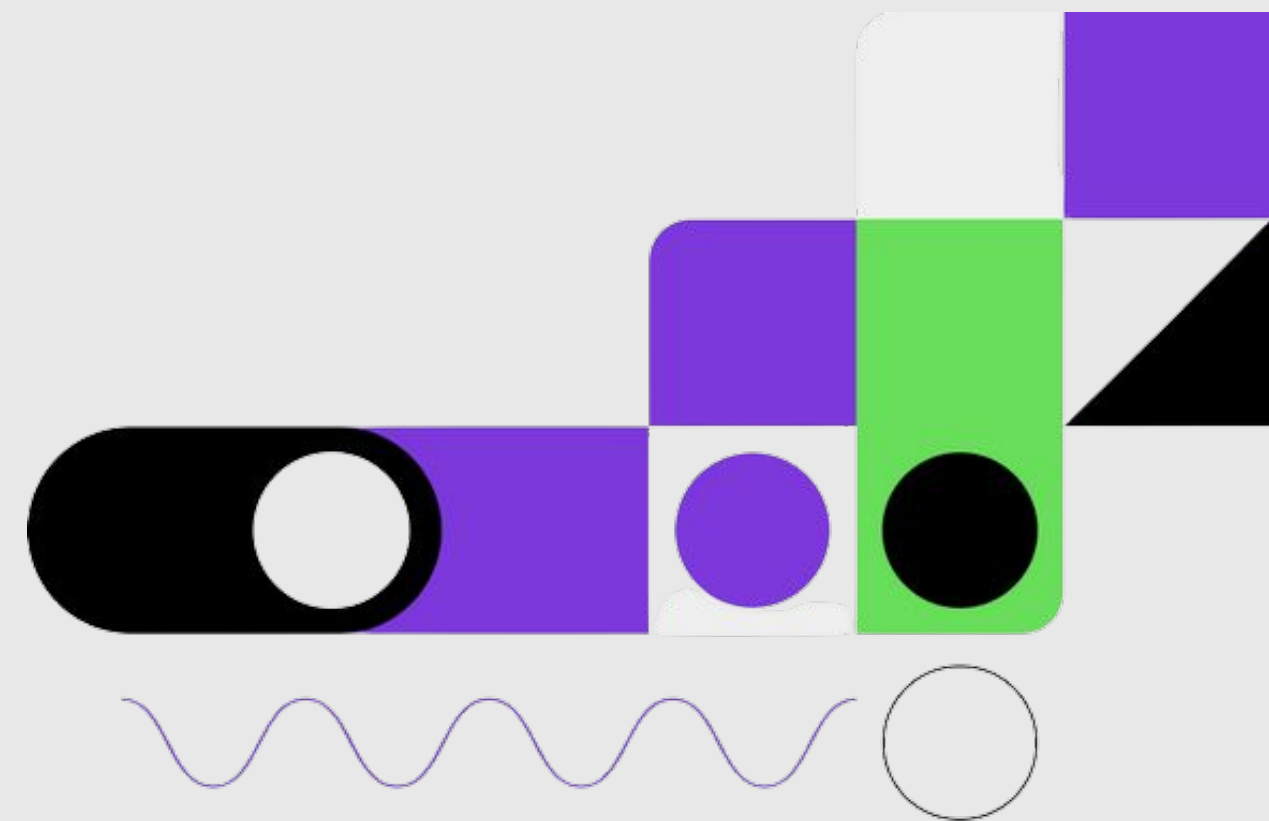
  // Aqui vai uma expressão

  return 'Pendente';

});
```

08. Jest

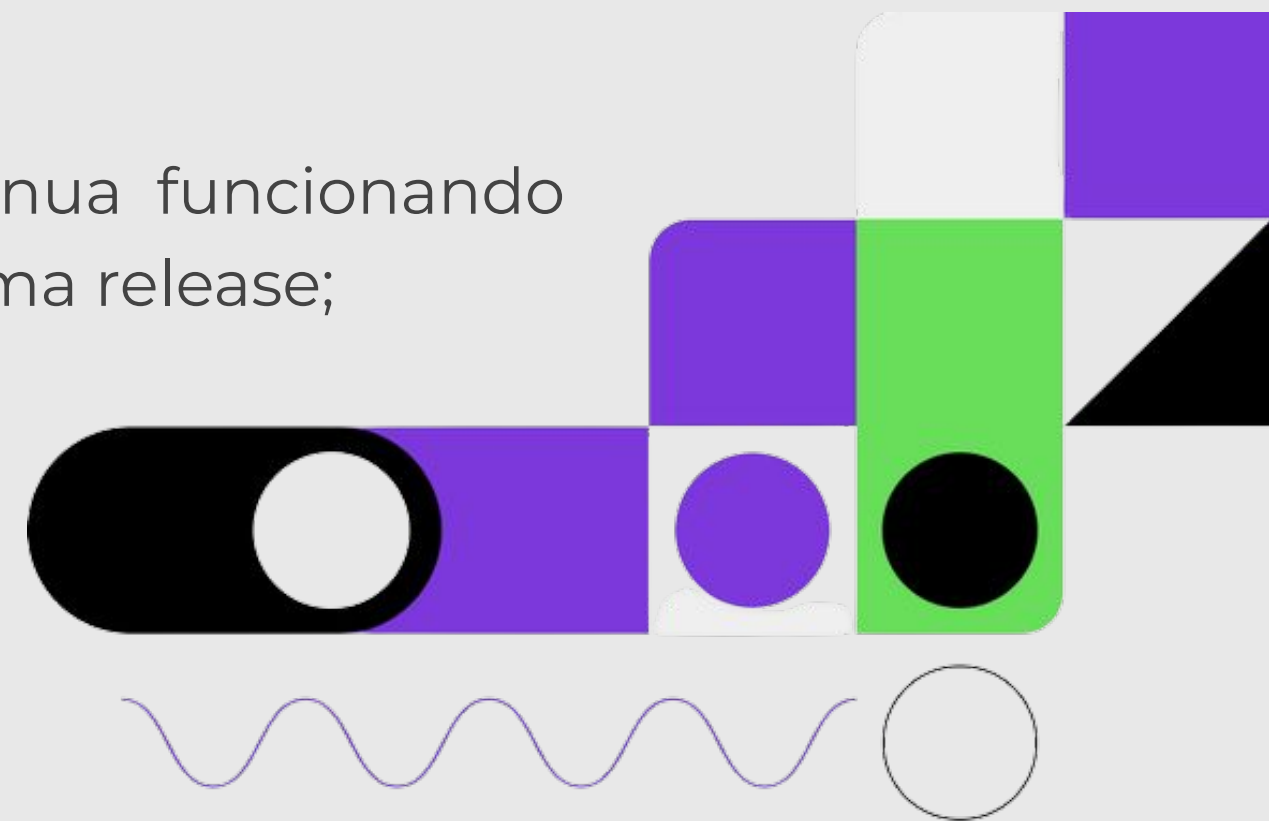
○ **Test Driven Development/TDD** ou Desenvolvimento Orientado a Testes é uma técnica criada por Kent Beck, um famoso programador e autor de livros que atualmente trabalha no Facebook.



08. Jest

A ideia do TDD é que você deve **escrever primeiro os testes** da sua aplicação, para depois implementar o código que fará com que eles funcionem. Isso pode soar um tanto estranho mas é uma ideia ousada que possui vários benefícios, tais como:

- diminuição do número de bugs, uma vez que não existem features sem testes;
- foco nas features que importam para o projeto, pois escrevemos os testes com os requisitos em mãos;
- permite testes de regressão, para ver se um sistema continua funcionando mesmo após várias mudanças e/ou muito tempo desde a última release;
- aumento da confiança do time no código programado;



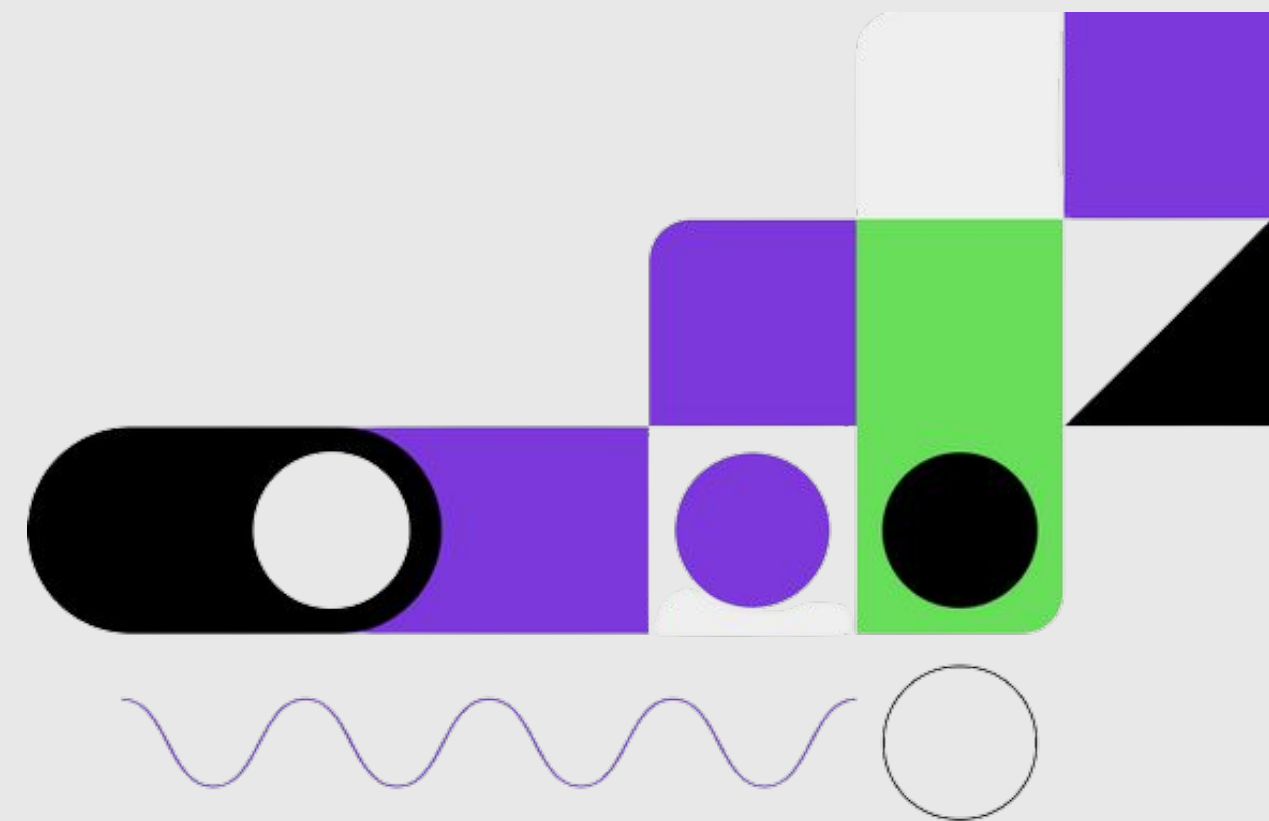
08. Jest

Um exemplo prático de aplicação de testes de unidade com TDD seria a criação de um método que aplica um desconto, em R\$, ao valor de um produto, também em R\$.

Vamos começar escrevendo a função de teste dessa funcionalidade em um arquivo index.js, comparando o retorno da função **aplicarDesconto** com o valor esperado, o que chamamos de asserção ou assert (em Inglês):

```
function aplicarDescontoTest() {  
  return aplicarDesconto(10, 2) === 8;  
}
```

```
console.log('A aplicação de desconto está funcionando? ');  
console.log(aplicarDescontoTest());
```

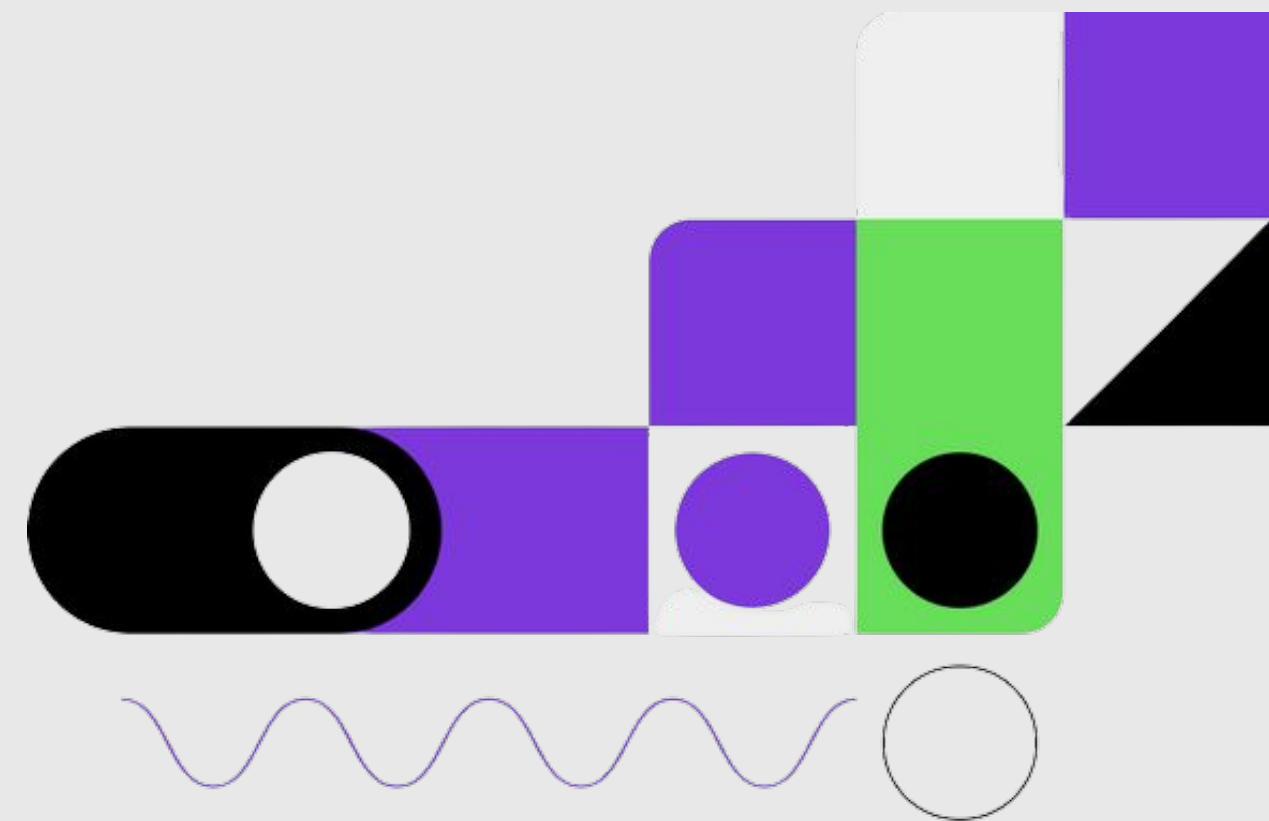


08. Jest

Se aplicarmos um desconto de R\$2 sobre um produto de R\$10, o valor esperado como retorno é R\$8, certo? Mas o que acontece se executarmos este bloco de código com o comando “node index” no terminal?

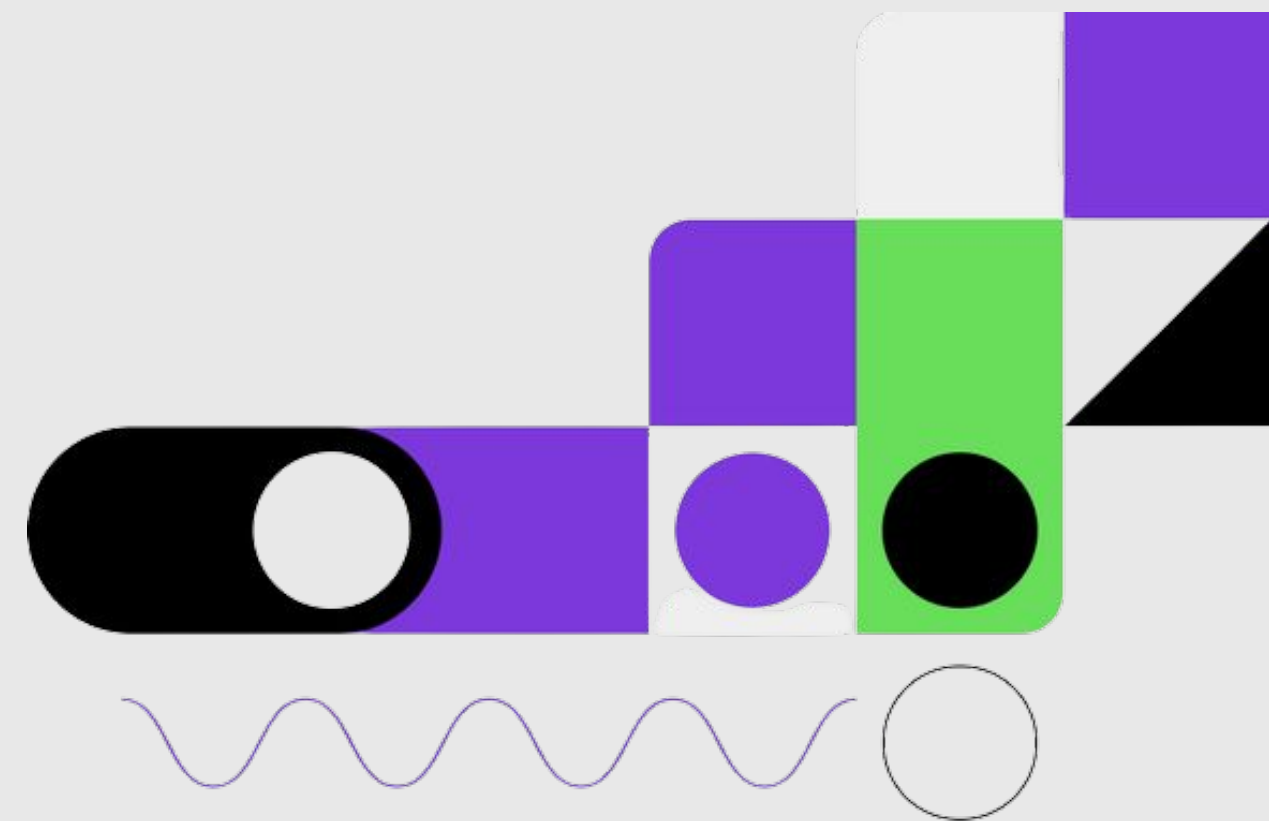
Dará um erro porque a função **aplicarDesconto** ainda não existe. Ou seja, sabemos o resultado esperado, mas não programamos a função ainda. Vamos fazê-lo agora, com uma simples subtração, no mesmo arquivo:

```
function aplicarDesconto(valor, desconto) {  
  return valor - desconto;  
}
```



08. Jest

Agora se rodarmos o arquivo index.js no terminal novamente ele deve indicar que está funcionando, pois ao testar nossa função passando 10 e 2, ela retornará 8 e a asserção será verdadeira.





Fechamento

Vimos diversos assuntos até aqui, ainda há muito a ser abordado. Esse material pode servir como consulta e guia de estudos.

Até a próxima pessoal!