



INSTITUTO FEDERAL

Mato Grosso do Sul

Test Driven Development - TDD

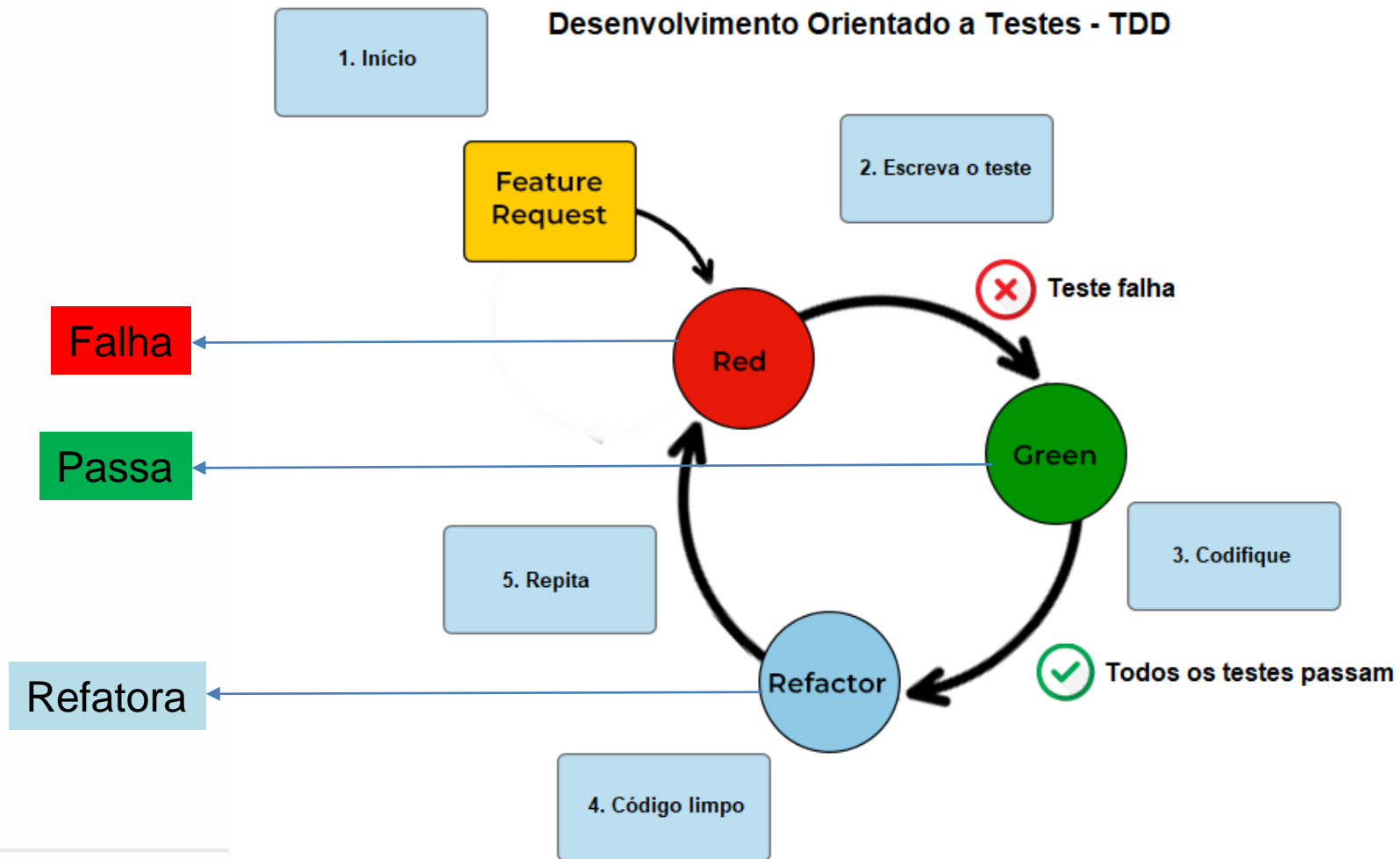
TDD – Desenvolvimento orientado a testes

- TDD - *Test Driven Development*
- Metodologia de teste de software
- Escreve os casos de teste antes do programa
- Quando os testes estiverem funcionando, você codifica o programa

TDD – Desenvolvimento orientado a testes

- Kent Beck:
 - Engenheiro de Software Americano
 - Formado pela University of Oregon
 - criador do paradigma *eXtreme Programming-XP*
 - Criador do framework **SUnit** para teste unitário em *SmallTalk*
 - *A partir dele veio i.g. JUnit*
 - *Criador do TDD em 2003*





Desenvolvimento Orientado a Testes - TDD

1. Início

Feature Request

2. Escreva o teste

✗ Teste falha

3. Codifique

✓ Todos os testes passam

5. Repita

4. Código limpo

Red

Green

Refactor

- Cada caso de teste falha inicialmente: Isto assegura que o teste realmente funciona e pode pegar um erro.
- Uma vez que isto é mostrado, a funcionalidade subjacente pode ser implementada. Isto levou ao "mantra de desenvolvimento orientado pelo teste", que é "vermelho/verde/fator/refator".

Desenvolvimento Orientado a Testes - TDD

1. Início

Feature Request

2. Escreva o teste

✗ Teste falha

Green

3. Codifique

✓ Todos os testes passam

Refactor

4. Código limpo

5. Repita

- A adição de um novo recurso começa com a escrita de um teste que passa se as especificações do recurso forem cumpridas.
- O desenvolvedor pode descobrir estas especificações perguntando sobre casos de uso e histórias de usuários.
- Um benefício chave do desenvolvimento orientado por testes é que ele faz com que o desenvolvedor se concentre nos requisitos antes de escrever o código.
- Isto está em contraste com a prática usual, onde os testes unitários são escritos somente após a escrita do código.

Desenvolvimento Orientado a Testes - TDD

1. Início

Feature Request

2. Escreva o teste

✗ Teste falha

Red

Green

3. Codifique

✓ Todos os testes passam

Refactor

5. Repita

4. Código limpo

- Realizar todos os testes.
- O novo teste deve falhar por razões esperadas
- Isto mostra que um novo código é realmente necessário para a característica desejada.
- Ele valida que o teste em potencial está funcionando corretamente.
- Ele descarta a possibilidade de que o novo teste seja defeituoso e sempre passe.

1. Início

Desenvolvimento Orientado a Testes - TDD

Feature Request

2. Escreva o teste

Red



Teste falha

Green

3. Codifique



Todos os testes passam

Refactor

5. Repita

4. Código limpo

- Escreva o código mais simples que passe no novo teste.
- O código deselegante ou rígido é aceitável, desde que passe no teste.
- O código será aperfeiçoado de qualquer forma na etapa 5.
- Nenhum código deve ser adicionado além da funcionalidade testada.

1. Início

Desenvolvimento Orientado a Testes - TDD

Feature Request

2. Escreva o teste

Red



Teste falha

Green

3. Codifique



Todos os testes passam

Refactor

4. Código limpo

5. Repita

- Todos os testes devem agora passar.
- Se algum falhar, o novo código deve ser revisado até que seja aprovado.
- Isso garante que o novo código atenda aos requisitos de teste e não quebre as características existentes.

1. Início

Desenvolvimento Orientado a Testes - TDD

Feature Request

2. Escreva o teste

❌ Teste falha

3. Codifique

✅ Todos os testes passam

5. Repita

4. Código limpo

Red

Green

Refactor

- A refatoração é necessária, utilizando testes após cada refator para garantir que a funcionalidade seja preservada
- O código é refatorado para a legibilidade e a capacidade de manutenção. Em particular, os dados de teste codificados devem ser removidos.
- A execução do conjunto de teste após cada refator ajuda a garantir que nenhuma funcionalidade existente seja quebrada.

Exemplos de refatoração:

- movendo o código para onde ele mais logicamente pertence
- remoção de código duplicado
- fazer autodocumentação de nomes
- métodos de divisão em partes menores
- reordenando as hierarquias sucessórias.

Desenvolvimento Orientado a Testes - TDD

1. Início

Feature Request

2. Escreva o teste

❌ Teste falha

3. Codifique

✅ Todos os testes passam

4. Código limpo

5. Repita

Red

Green

Refactor

- O ciclo acima é repetido para cada nova peça de funcionalidade.
- Os testes devem ser pequenos e incrementais, e os *commits* devem ser feitos com frequência.
- Dessa forma, se o novo código falhar em alguns testes, o programador pode simplesmente desfazer ou reverter em vez de depurar excessivamente.
- Ao utilizar bibliotecas externas, é importante não escrever testes que sejam tão pequenos a ponto de testar efetivamente apenas a própria biblioteca, a menos que haja alguma razão para acreditar que a biblioteca esteja com buggu ou não tenha recursos suficientes para atender a todas as necessidades do software em desenvolvimento.

Estilos do TDD

- Ao se concentrar na escrita apenas do código necessário para passar nos testes, os projetos podem muitas vezes ser mais limpos e claros do que se consegue por outros métodos.
 - "keep it simple, stupid" ([KISS](#))
 - "[You aren't gonna need it](#)" (YAGNI)
 - Kent Beck sugere: "[Fake it till you make it](#)".

Benefícios do TDD

- Ele ajuda a garantir que a aplicação seja escrita para teste, pois os desenvolvedores devem considerar como testar a aplicação desde o início, em vez de adicioná-la mais tarde.
- Ele também garante que os testes para cada característica sejam escritos.
- Além disso, escrever os testes primeiro leva a uma compreensão mais profunda e antecipada dos requisitos do produto, assegura a eficácia do código de teste e mantém um foco contínuo na qualidade do software.

Benefícios do TDD

- Ao escrever o código do primeiro recurso, há uma tendência dos desenvolvedores e organizações de empurrar o desenvolvedor para o próximo recurso (Feature), mesmo negligenciando completamente os testes.
- O primeiro teste TDD pode nem mesmo compilar no início, porque as classes e métodos que ele requer podem ainda não existir. No entanto, esse primeiro teste funciona como o início de uma especificação executável.

IMPORTANCE OF TEST-DRIVEN DEVELOPMENT



● Improve code quality

Melhorar a qualidade do código fonte



● Boost system design

Impulsionar o projeto de todo o sistema



● Increase developer productivity

Aumentar produtividade da equipe



● Reduce project costs over time

Reduzir custos ao longo do tempo



● Obtain assistance for bug prevention

Redução da manutenção

Passo a passo do TDD

- Pense no que você quer fazer.
- Pense em como testá-lo.
- Escreva um pequeno teste. Pense sobre a API desejada.
- Escreva apenas o código suficiente para reprovar no teste.
- Execute e veja o teste falhar. (O test-runner, se você estiver usando algo como JUnit, mostra a "Barra Vermelha"). Agora você sabe que seu teste vai ser executado.
- Escreva apenas o código suficiente para passar no teste (e passe em todos os testes anteriores).
- Execute e veja todos os testes passarem. (O test-runner, se você estiver usando a JUnit, etc., mostra a "Barra Verde"). Se não passar, você fez algo errado, conserte-o agora, pois tem que ser algo que você acabou de escrever.

Passo a passo do TDD

- Se você tiver alguma lógica duplicada, ou código inexpressivo, refatorar para remover a duplicação e aumentar a expressividade - isto inclui a redução do acoplamento e o aumento da coesão.
- Faça os testes novamente, você ainda deve ter a Barra Verde. Se você conseguir a Barra Vermelha, então você cometeu um erro em sua refatoração. Conserte-a agora e execute novamente.
- Repita os passos acima até não encontrar mais testes que conduzam à escrita de novo código.

Alguns Frameworks

- Junit para Java
- PHPUnit para PHP
- CppUnit para C++
- Jasmine para JavaScript

TDD vs BDD

- TDD - Test Driven Development
- BDD - Behavior Driven Development
 - Derivado do TDD

BDD – Desenvolvimento orientado a comportamento

- BDD é técnica de desenvolvimento ágil que visa integrar regras de negócios com linguagem de programação, focando o comportamento do software.
- Além disso, pode-se dizer também, que BDD é a evolução do TDD. Isto porque, os testes ainda orientam o desenvolvimento, ou seja, primeiro se escreve o teste e depois o código.
- A linguagem de negócio usada em BDD é extraída das histórias ou especificações fornecidas pelo cliente durante o levantamento dos requisitos.
- Possibilita uma comunicação eficiente entre as equipes de desenvolvimento e testes.

O que é o framework Jasmine?

- Framework para criação de testes em Javascript.
- Utiliza o conceito BDD.
- BDD permite criação de testes intuitivos e de fácil compreensão.
- É rápido e não precisa de dependência externa.
- Teste diretamente no navegador ou por linha de comando no terminal.
- Pode ser usado independente ou incorporado a um projeto NodeJS, Ruby, Python, etc.

Criando suítes e testes

- distribuição standalone: acessa o site, faz download e já programa com os testes.
- 1º Acessar o github do jasmine-standalone
 - <https://github.com/jasmine/jasmine/releases>
- 2º Criar uma pasta de sua preferência e descompactar o arquivo zipado



Search or jump to...



Pull requests

Issues

Codespaces

Marketplace

Explore



jasmine / jasmine Public

Watch 448

Fork 2.3k

Star 15.5k

<> Code

Issues

29

Pull requests

1

Actions

Projects

Wiki

Security

Insights

Releases

Tags

Find a release

Oct 29, 2022



sgravrock



v4.5.0



481f1e7

Compare

4.5.0

Latest

Please see the [release notes](#).

Assets

3



jasmine-standalone-4.5.0.zip

99.2 KB

Oct 29, 2022



Source code (zip)

Oct 29, 2022



Source code (tar.gz)

Oct 29, 2022

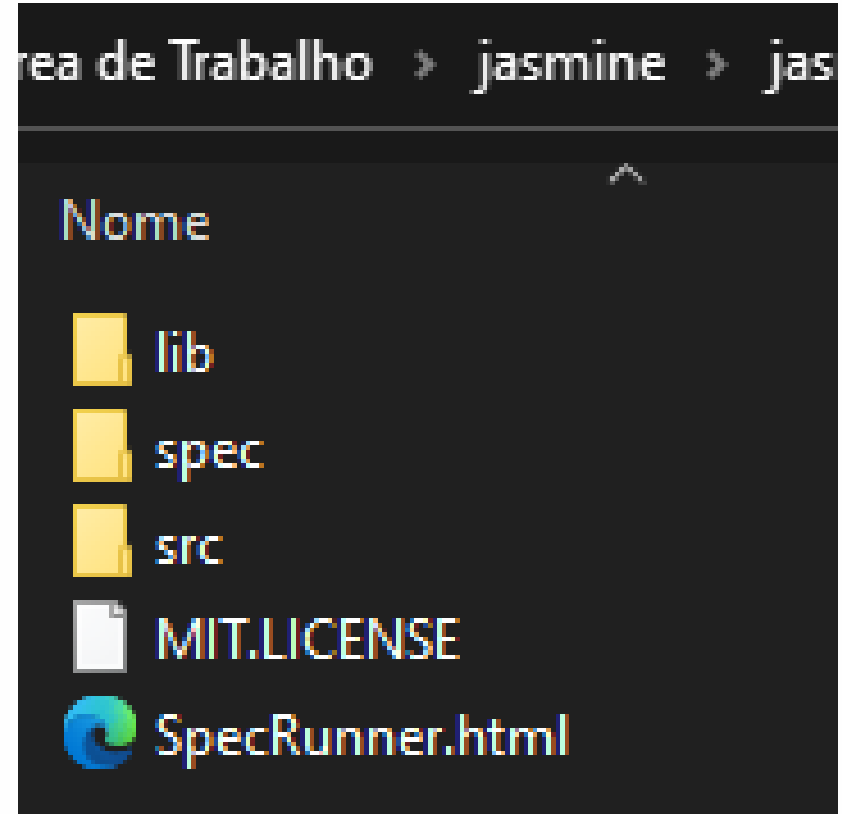


6

6 people reacted

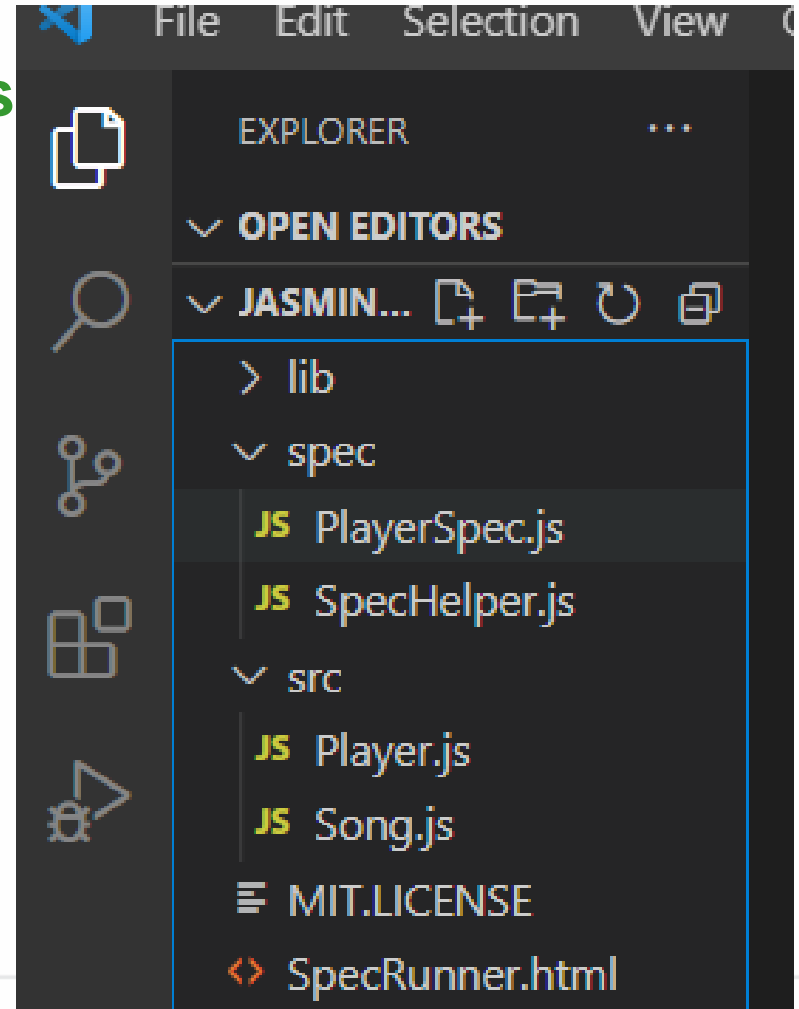
Criando suítes e testes

- 3º Possui 3 diretórios:
- **lib** (bibliotecas do próprio jasmine)
- **spec** (duas classes de testes por padrão)
- **Src** (código fonte da aplicação)



Criando suítes e testes

- Para cada objeto criado em src você deverá criar um objeto referente a Especificação
- Ex. Player e PlayerSpec



O que é o framework Jasmine?

- SpecRunner.html
 - responsável por executar os testes no navegador (tudo pronto e configurado) e entregar o relatório do resultado dos testes.
 - exibir e executar os testes.

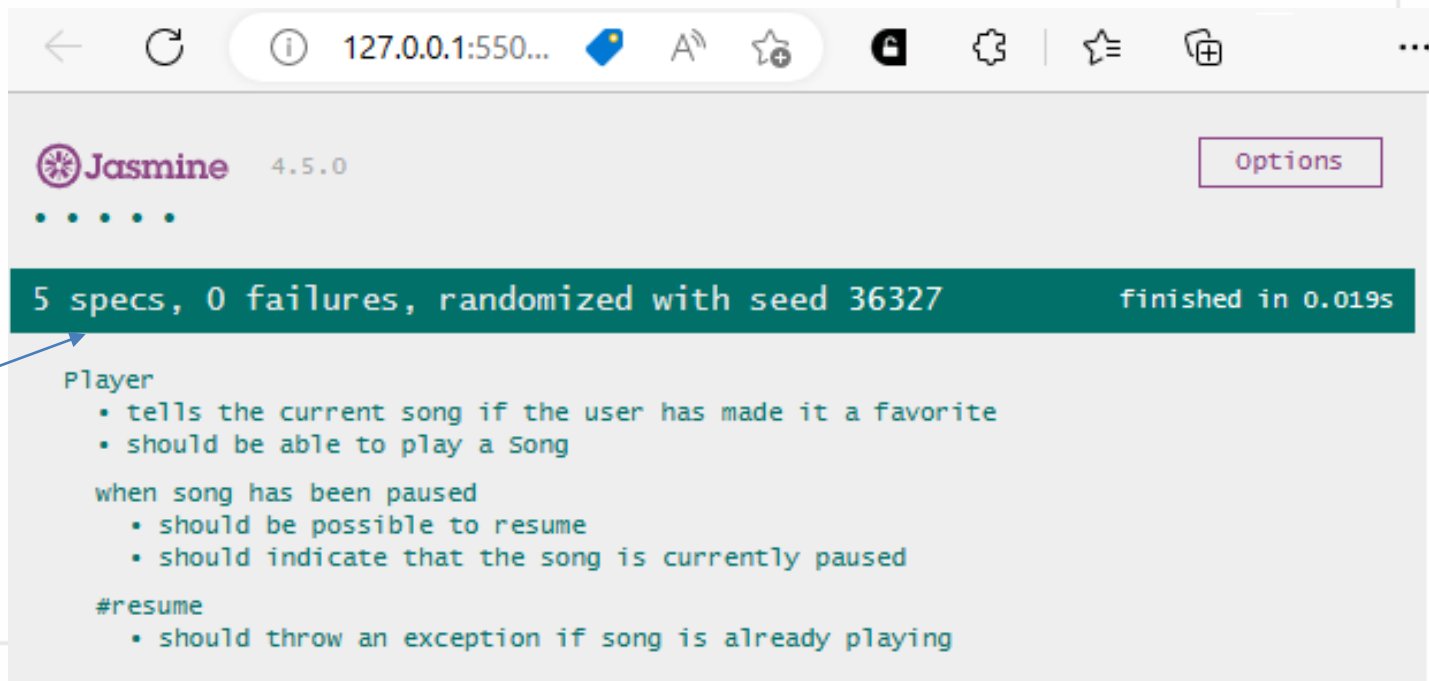
O que é o framework Jasmine?

- SpecRunner.html
 - Possui duas instruções:

```
16      <!-- include source files here... -->
17      <script src="src/Player.js"></script>
18      <script src="src/Song.js"></script>
19
20      <!-- include spec files here... -->
21      <script src="spec/SpecHelper.js"></script>
22      <script src="spec/PlayerSpec.js"></script>
23
```

Criando suítes e testes

- Abrir o arquivo no navegador:



Criando suítes e testes

- Mão na massa!
 - no diretório do jasmine
 - ~~1-apagar arquivos do spec e do src~~
 - 2-inserir/criar novos arquivos
 - 3-ajustar os nomes no SpecRunner.js
 - 4-abrir/testar **SpecRunner.html**

Criando suítes e testes

- Suítes de testes, servem para definir o escopo do que está sendo testado
 - uma aplicação é composta de diversas suítes
 - exemplo: Cadastro de clientes, Operações matemáticas, etc.
 - Tem que ser isoladas em blocos mínimos testáveis.
 - Specs ou Expectations (Expectativas)

Criando suítes e testes

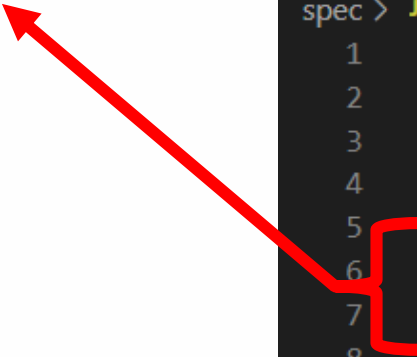
- No Jasmine a suíte é uma função global JavaScript chamada “**describe**”, que possui dois parâmetros que são sua **descrição** e **teste (function)**

```
describe("Hello World", function() {  
  
});
```

Teste = spec

Criando suítes e testes

- **it**
 - Uma suíte contém uma ou mais expectativas que testam o estado do código.
 - Deve começar com a palavra **deve** ou **deveria** (should), fazendo referência ao que o teste deve fazer:



```
spec > JS olaSpec.js > ...
1  /*
2  * Este é um arquivo que chamará nosso arquivo javaSc
3  * Cada bloco "Describe" é equivalente a um caso de t
4  */
5  describe("Hello World", function() {
6      it("should Return Hello world",function() {
7          expect(helloworld()).toEqual('Hello World');
8      });
9  });
```


Criando suítes e testes

- **expect**
 - Uma expectativa em Jasmim é uma asserção que é verdadeira ou falsa.
 - Verdade = Passar
 - Falso = Falhar

Expectativa ou
verificação ou
validação

```
spec > JS olaSpec.js > ...  
1  /*  
2  * Este é um arquivo que chamará nosso arquivo javaSc  
3  * Cada bloco "Describe" é equivalente a um caso de t  
4  */  
5  describe("Hello World", function() {  
6      it("should Return Hello world",function() {  
7          expect(helloworld()).toEqual('Hello World');  
8      });  
9  });
```

Criando suítes e testes

- **expect**
 - Deve ser utilizado em conjunto com uma comparação (Matcher) que conterá o valor a ser comparado.
 - **toEqual()**: comparador de igualdade para strings

Matcher: São funções que retornam um comparador booleano para ser verificado por uma expectation (verificação)

```
spec > JS olaSpec.js > ...  
1  /*  
2  * Este é um arquivo que chamará nosso arquivo javaSc  
3  * Cada bloco "Describe" é equivalente a um caso de t  
4  */  
5  describe("Hello World", function() {  
6      it("should Return Hello world",function() {  
7          expect(helloworld()).toEqual('Hello World');  
8      });  
9  });
```

Criando suítes e testes

- **Matcher:**
 - toBe
 - toEqual
 - toBeDefined
 - toBeUndefined
 - toBeNull
 - toBeTruthy
 - toBeFalsy
 - toContain
 - toBeLessThan
 - toBeGreaterThan
 - toThrow
 - toThrowError
 - Etc.
 - É possível criar um Matcher

Criando suítes e testes

```
src > JS ola.js > ...
```

```
1  var olamundo = function() {  
2      return 'Olá Mundo!';  
3  };
```

```
spec > JS olaSpec.js > ...
```

```
1  /*  
2  * Este é um arquivo que chamará nosso arquivo javascript que precisa ser testado.  
3  * Cada bloco "Describe" é equivalente a um caso de teste.  
4  */  
5  describe("Hello World", function() {  
6      it("should Return Hello world",function() {  
7          expect(helloworld()).toEqual('Hello World');  
8      });  
9  });
```

```
<script src="src/ola.js"></script>  
<script src="spec/olaSpec.js"></script>
```

Exercício Resolvido

- Somar dois números

```
spec > JS somarSpec.js > ...  
1  ✓ describe("Operação de Adição",function(){  
2  |       it("deve garantir que 1+1=2", function(){  
3  |           expect(1+1).toBe(2);  
4  |       });  
5  |   });
```



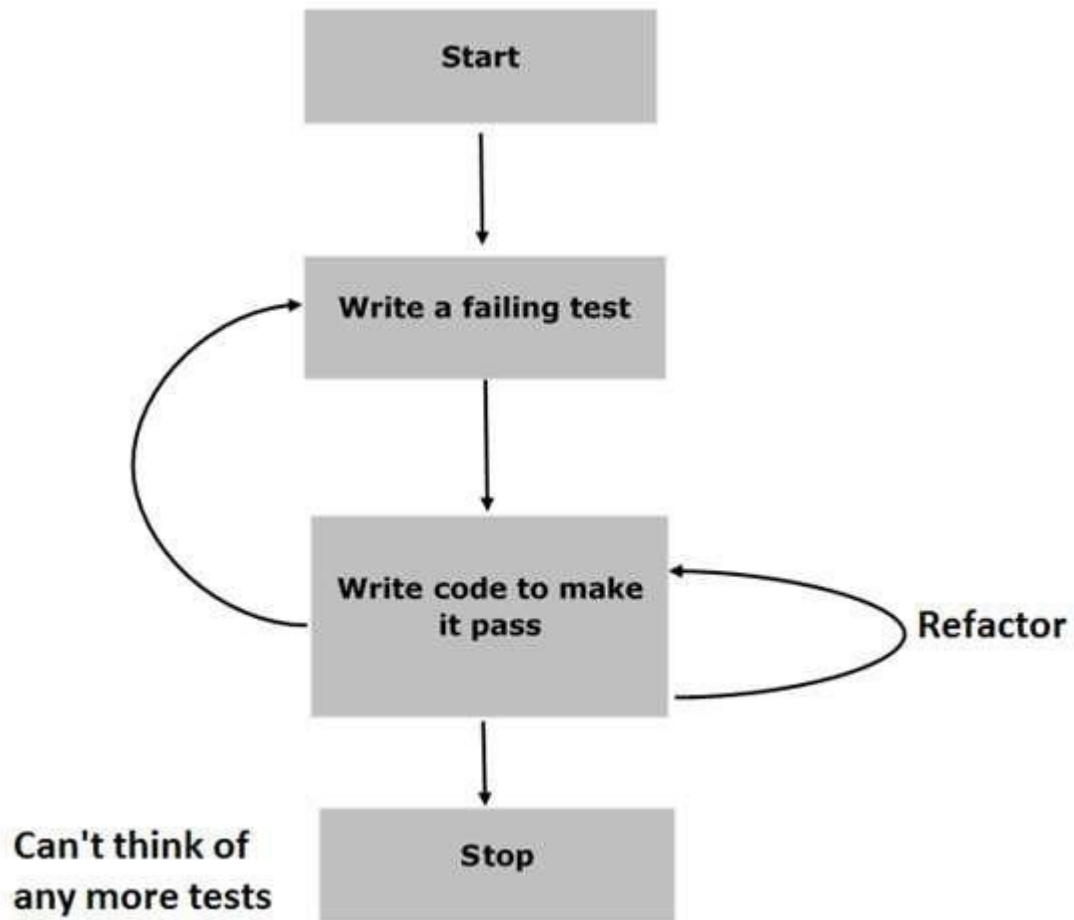
4.5.0

•

1 spec, 0 failures, randomized with seed 83445

Operação de Adição

- deve garantir que 1+1=2



Referências

- TDD: https://en.wikipedia.org/wiki/Test-driven_development
- Test Case: https://en.wikipedia.org/wiki/Test_case
- What is TDD:
<https://www.spiceworks.com/tech/devops/articles/what-is-tdd/>
- TDD: <http://wiki.c2.com/?TestDrivenDevelopment>
- Afinal o que é TDD: <https://www.treinaweb.com.br/blog/afinal-o-que-e-tdd>
- BDD: <https://www.devmedia.com.br/desenvolvimento-orientado-por-comportamento-bdd/21127>