FACULDADE DE TECNOLOGIA DE SÃO PAULO

GIOVANNI TEIXEIRA CAMPOPIANO

IMPORTÂNCIA DE TESTES UNITÁRIOS NO DESENVOLVIMENTO WEB

SÃO PAULO

2022

FACULDADE DE TECNOLOGIA DE SÃO PAULO

GIOVANNI TEIXEIRA CAMPOPIANO

IMPORTÂNCIA DE TESTES UNITÁRIOS NO DESENVOLVIMENTO WEB

Trabalho submetido como exigência parcial
para a obtenção do Grau de Tecnólogo em
Análise e Desenvolvimento de Sistemas
Orientador: Profº Paulo Roberto Bernice

SÃO PAULO
2022

FACULDADE DE TECNOLOGIA DE SÃO PAULO

GIOVANNI TEIXEIRA CAMPOPIANO

IMPORTÂNCIA DE TESTES UNITÁRIOS NO DESENVOLVIMENTO WEB

Trabalho submetido como exigência parcial para a obtenção do Grau de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Parecer do Professor Orientador

OK

Conceito/Nota Final: 9,0 (nove)

Atesto o conteúdo contido na postagem do ambiente TEAMS pelo aluno e assinada por mim para avaliação do TCC.

Orientador: Profº Paulo Roberto Bernice

SÃO PAULO, 23 de junho de 2022.

Assinatura do Orientador

Assinatura do aluno Giovanni T. Campopiano

RESUMO

Nesta monografia, será avaliada a importância da implementação de testes unitários na camada *front-end*, de aplicações *web*. Será fornecida uma visão geral da avaliação realizada, a partir da aplicação de padrões e boas práticas, na implementação dos testes unitários.

O desenvolvimento de *software* exige foco e atenção a detalhes, e, dado certo nível complexidade das regras a serem implementadas, falhas lógicas podem passar despercebidas.

Com o objetivo de se aplicar os métodos e padrões de testes unitários, adotou-se a implementação dos testes em casos reais. Foram utilizadas referências mundiais sobre os tópicos relacionados ao tema, bem como o *framework* de desenvolvimento Angular, e o *framework* de testes Jasmine.

Considera-se que a pesquisa alcançou seus objetivos como apresentado e discutido neste artigo, e que ampliou a visão do autor sobre o uso de testes ao longo do desenvolvimento de *software*.

Palavras-chave: Aplicação web. Front-end. JavaScript. Framework. Angular. Jasmine. Testes unitários. Testes automatizados.

ABSTRACT

This work will evaluate the importance of implementing unit tests in web applications. A general vision of the evaluation will be presented, through the use of patterns and good practices, while implementing unit tests.

Developing software requires focus and attention to details, and, given certain complex rules to be implemented in the code, logic flaws might go unnoticed.

In order to make use of methodologies and test patterns used in test automation, this work sought to implement tests in real scenarios. World references related to test automation have been consulted, the web application framework Angular and the testing framework Jasmine have been used to implement the tests.

It is considered that the research achieved its goal as presented and elaborated in this work, and expanded the author's view on the use of tests in software development.

Keywords: Web application. Front-end. JavaScript. Framework. Angular. Jasmine. Unit Test. automated tests.

LISTA DE FIGURAS

LISTA DE ABREVIATURA E SIGLAS

CLI: *Command line interface*, em português, interface de linha de comando

E2E: *End-to-end*, em português, ponta a ponta

HTML: *HyperText Markup Language*, em português, linguagem de marcação de hipertexto

NPM: Node Package Manager

TDD: *Test-driven development*, em português, desenvolvimento guiado por testes.

SUMÁRIO

1. Introdução

A partir da identificação de diversas necessidades e tendências do mercado, ao longo das duas últimas décadas, metodologias ágeis vêm adotando e recomendando a utilização de práticas contempladas no Desenvolvimento Guiado por Testes, traduzido livremente do inglês, *Test-Driven Development* (TDD), sugerindo que, ao longo de todo o desenvolvimento de *software*, testes automatizados sejam escritos, constantemente, pelo desenvolvedor, antes mesmo que as funcionalidades sejam implementadas.

Nota-se que, conforme o escopo dos projetos tomam proporções cada vez maiores, com adições de novas funcionalidades, é de suma importância garantir que o código continuará a funcionar, conforme esperado, após cada alteração. Uma falha no *software* pode causar enormes prejuízos financeiros e de tempo. Em alguns casos, o prejuízo causado por uma falha no *software* é imensurável e irreversível, causando danos à imagem da empresa, e até mesmo ferindo a integridade e privacidade de pessoas. Além disso, um agravante para este fator, seria a intolerância crescente, quanto ao tempo de entrega, tornando os cronogramas cada vez mais curtos (BURNSTEIN, 2003).

O bom uso das práticas de TDD podem auxiliar no desenvolvimento mais eficiente e na produção de um *software* simples e de qualidade, contudo, a necessidade de ter um código robusto e de qualidade, e curto espaço de tempo para se desenvolver e entregar o *software*, vem impactando toda a forma de se pensar em engenharia de *software*, além de impactar, também, os desenvolvedores envolvidos na elaboração dos projetos (CRISPIN; GREGORY, 2009).

Neste cenário, as equipes devem encontrar nas metodologias, técnicas e ferramentas, a maneira mais produtiva de produzir *softwares* com qualidade (CRISPIN; GREGORY, 2009).

Embora a testagem de *software* venha se tornando uma atividade cada vez mais requerida atualmente, em geral, ela não é realizada de forma meticulosa, por motivos já elencados, como limitações temporais, qualificação técnica, recursos, e a própria complexidade dos sistemas (CRISPIN; GREGORY, 2009). Com isso, a automação de teste tem sido vista como uma das principais medidas para melhorar a eficiência dessa atividade, além de aliviar sua carga de trabalho (FEWSTER; GRAHAM, 1999).

1.1 Objetivo

Esta análise tem por objetivo avaliar os benefícios dos testes automatizados, especificamente pela abordagem de testes unitários, no desenvolvimento de aplicações *web*.

2. Teste de Software

Compreender o verdadeiro significado de teste de *software* é de suma importância para a otimização de esforços e recursos, e sua má interpretação pode impactar diretamente no objetivo e na eficiência dos testes. Por exemplo, na definição “O processo de testagem, demonstra que não existem erros no *software*”, é estabelecido o objetivo de não encontrar erros no *software*, podendo enviesar, o responsável pelo teste, a buscar esse objetivo, ou seja, deixando-o com forte tendência a selecionar dados de teste que possuem uma probabilidade baixa de levar o programa ao fracasso, o que impacta diretamente na eficiência dos testes. (MYERS, 2004).

Ao testar um *software*, deseja-se agregar valor, elevar a qualidade, confiabilidade, manutenibilidade, performance e robustez geral do código, através da identificação e remoção de erros. Logo, a ideia de se testar um *software* para demonstrar que o mesmo funciona, está equivocada. Deve-se partir do princípio que o *software* é passível de conter erros, e que, o teste, deve ser focado em encontrar o maior número de erros possíveis (MYERS, 2004).

Dessa forma, podemos definir que o teste de *software* é um processo controlado, que verifica a qualidade e integridade do *software*, por meio da identificação de erros, cuja incidência é assumida, desde o início.

O teste demonstra a presença de defeitos, reduzindo a probabilidade dos mesmos permanecerem no *software*, mas, vale ressaltar que, caso não sejam encontrados, não é provado que o *software* está livre de defeitos.

Em geral, é impraticável encontrar todos os erros de um *software*. Testar todos os cenários, ou as possíveis combinações de entradas, não é viável, portanto, sugere-se considerar prioridades e riscos do projeto, com intuito de canalizar os esforços de teste.

Um *software* pode ser testado, de forma automatizada, basicamente, por três abordagens: testes de ponta a ponta, traduzido livremente do inglês, end-to-end (E2E), testes de integração e testes unitários.

No teste ponta a ponta, o objetivo é testar a aplicação como um todo, de uma vez, enquanto que os testes de integração visam testar como cada uma das unidades se integra. Os testes unitários, como o próprio nome sugere, dizem respeito à testagem automatizada de uma única unidade, um único componente, e suas respectivas funcionalidades.

2.1. Testes Unitários

O teste unitário é uma técnica essencial para o aperfeiçoamento, na codificação de um sistema, pois proporciona *feedback* em tempo real, no momento em que o código é escrito, além disso, garante cobertura e altíssima detecção de erros ao utilizar técnicas para teste de unidade (HUNT; THOMAS, 2003).

O profissional que desenvolve e executa os testes unitários, é responsável por garantir a exatidão e eficiência do código, por meio de simulações que devem abranger as mais variadas condições e cenários, além das entradas e saídas esperadas.

O teste unitário é desenvolvido com o intuito de facilitar as atividades do desenvolvimento de um sistema, garantindo uma base robusta nos demais processos de testes de integração e testes ponta a ponta, o que possibilita projetos de alta qualidade, e também proporciona uma redução considerável no tempo necessário para se depurar um trecho de código. Segundo Hunt e Thomas (2003), o teste unitário é feito por programadores, para programadores.

3. Metodologia

Esta seção tem como objetivo elaborar sobre as tecnologias e padrões utilizados na avaliação dos benefícios providos pelos testes unitários, no desenvolvimento da camada de *front-end*, de aplicações *web*, além de fornecer um breve guia de configuração.

3.1. Tecnologias

Para a plataforma de desenvolvimento, foi escolhido o Visual Studio Code: desenvolvido pela Microsoft, é comumente conhecido como VS Code, e é uma das plataformas de desenvolvimento mais utilizadas por desenvolvedores de aplicações *web*, por fornecer diversas ferramentas e customizações, que auxiliam no processo de desenvolvimento de código.

Dentre os diversos *frameworks* disponíveis no mercado, foram escolhidas as seguintes tecnologias:

1. Angular: *framework* de aplicações *web*, desenvolvido e mantido pela empresa Google, e utiliza a licença MIT, uma licença permissiva, muito utilizada em *software* livre. Amplamente utilizado, tornou-se rapidamente um *framework* de peso no mercado, desde o lançamento do Angular 2, em 2016.
2. Jasmine: *framework* de testes automatizados para JavaScript, desenvolvido pela empresa Pivotal Labs, também faz uso da licença MIT, e é instalado como *framework* de teste padrão, junto ao Angular.

A escolha do Angular se deu pelo fato do *framework* proporcionar, intuitivamente, a modularização e componentização do código, no momento do desenvolvimento, onde, cada novo componente criado, acompanha seu respectivo componente de teste.

3.2. Configuração

À seguir, um breve guia de configuração para criar um novo projeto, em Angular:

1. Instalar uma plataforma de desenvolvimento, como Visual Studio Code, ou similar;
2. Instalação da versão mais atualizada do Node.JS, e um gerenciador de pacotes, para JavaScript. Ao instalar o Node.JS, é instalado, por padrão, o Node Package Manager (NPM);
3. Em um terminal de comando, executar o comando *npm install -g @angular/cli* para instalar a interface de linha de comando (CLI) do Angular;
4. Ainda em um terminal de comando, executar o comando *ng new novo-projeto* para criar um novo projeto.

3.3. Padrão de teste "AAA"

O padrão de teste “AAA”, ou *triple A*, é um padrão de estrutura de testes, e sugere que o desenvolvedor divida seu método de testes, em três sessões distintas: *Arrange*, *Act* e *Assert*.

Cada uma das sessões é responsável por um segmento da testagem, e serve como um guia para facilitar a criação de novos testes, assim como auxiliar na leitura de testes desenvolvidos previamente.

1. Arrange: etapa de preparo, onde montamos o cenário do nosso teste. Inicializamos variáveis, provedores, e tudo que for necessário para simular um cenário para nosso teste. Nota-se que, quanto mais específicos forem os cenários e dados montados nesta etapa, mais o teste fica vulnerável a erros, ou resultados falso-positivos. Portanto, o ideal é fazer uso de casos gerais durante a etapa de preparo;
2. Act: etapa de ação, onde realizamos a ação que define nosso teste. No caso de uma função, essa fase consiste em chamar a mesma. No caso de um botão, a ação será clicá-lo;
3. Assert: última etapa, fazemos afirmações sobre o comportamento esperado do código, e é onde realizamos a validação de resultados e ações, após a execução do código definido no teste. Nesta etapa, verificamos se as funções necessárias foram chamadas, se as variáveis que deveriam ter seus valores alterados, foram alteradas com o valor esperado, e demais ações;

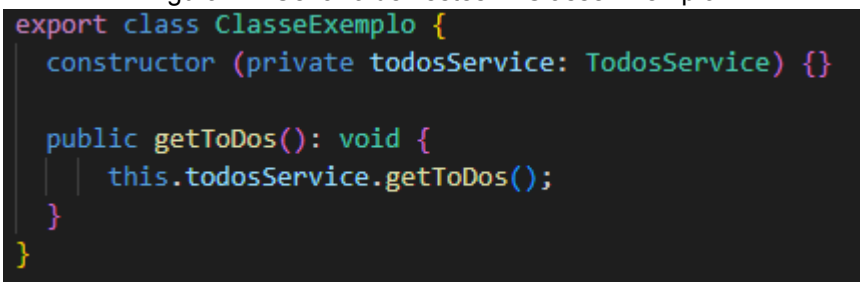
4. Apresentação

Esta seção apresenta exemplos de aplicação de testes unitários.

4.1. Primeiro teste: Validando funções

Neste primeiro cenário, será utilizada uma aplicação de listas de tarefas, que contém a seguinte classe, com apenas uma função, para entender como aplicar o padrão de teste “AAA” para escrever um teste.

Figura 1 – Cenário de Testes 1: Classe Exemplo



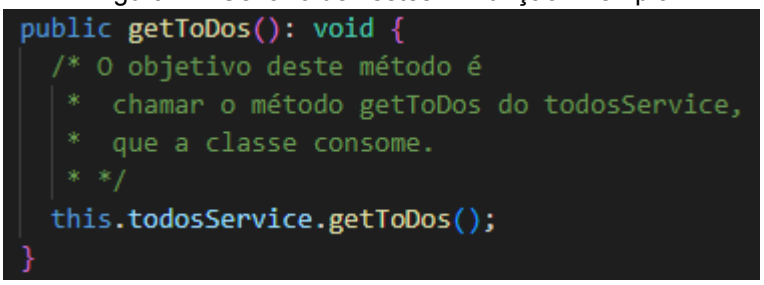
```
export class ClasseExemplo {
  constructor (private todosService: TodosService) {}

  public getToDos(): void {
      this.todosService.getToDos();
  }
}
```

Fonte: Autoria Própria

Primeiramente, define-se o que deve ser testado. Para isso, identifica-se o que é esperado que a unidade de código faça: por se tratar de uma classe com apenas um método, conclui-se que, para qualquer outro código externo que utilize esta classe, espera-se que ela possua um método getToDos(), e que este método não retorne nada.

Em seguida, deve-se identificar o que é esperado que o método getToDos() faça.

Figura 2 – Cenário de Testes 1: Função Exemplo

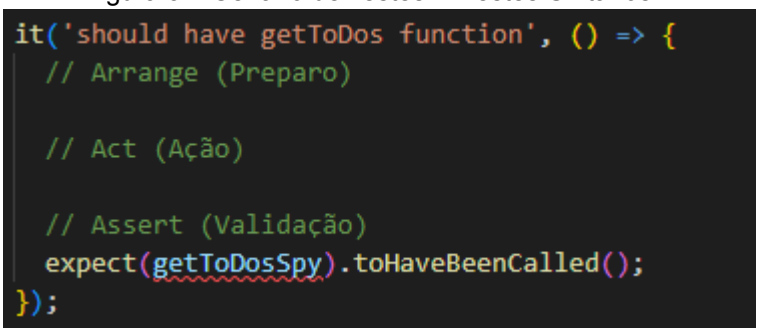


```
public getToDos(): void {
  /* O objetivo deste método é
   *  chamar o método getToDos do todosService,
   *  que a classe consome.
   * */
  this.todosService.getToDos();
}
```

Fonte: Autoria Própria

Identificando o objetivo do método, inicia-se o desenvolvimento do teste unitário: sabendo-se o que é preciso validar, escreve-se o que é esperado que o método realize.

Figura 3 – Cenário de Testes 1: Testes Unitários 1

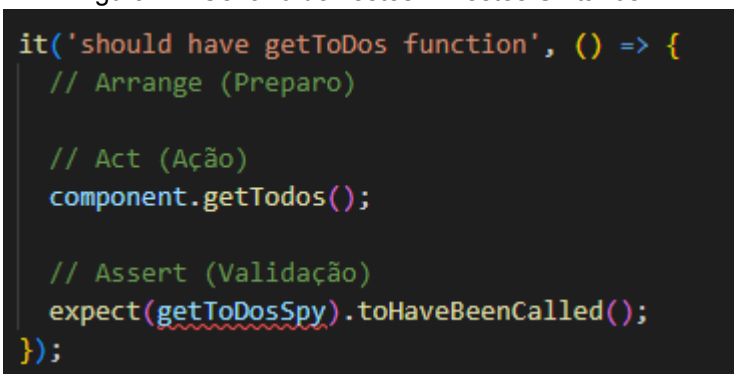


```
it('should have getToDos function', () => {
  // Arrange (Preparo)

  // Act (Ação)

  // Assert (Validação)
  expect(getToDosSpy).toHaveBeenCalled();
});
```

Fonte: Autoria Própria

O segundo passo será analisar qual é a ação tomada: para testar a função getToDos(), a ação será chamá-la.

Figura 4 – Cenário de Testes 1: Testes Unitários 2



```
it('should have getToDos function', () => {
  // Arrange (Preparo)

  // Act (Ação)
  component.getTodos();

  // Assert (Validação)
  expect(getToDosSpy).toHaveBeenCalled();
});
```

Fonte: Autoria Própria

Por fim, deve-se verificar o que necessita ser preparado, para que o teste consiga ser utilizado: dentro deste exemplo, é preciso instanciar o serviço TodosService, consumido pela ClasseExemplo, e inicializar um espião.

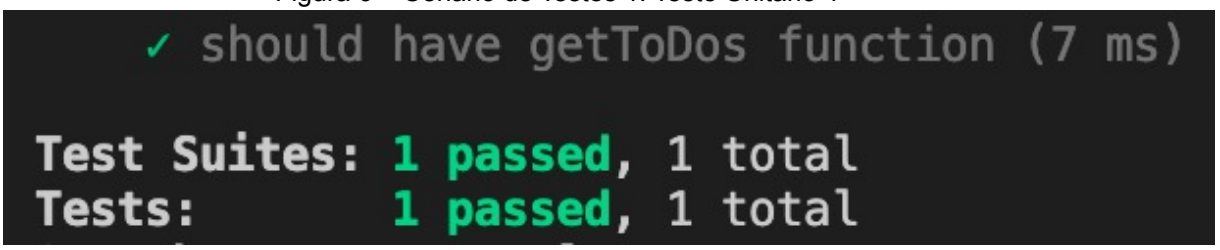
Figura 5 – Cenário de Testes 1: Testes Unitários 3

```
it('should have getToDos function', () => {
  // Arrange (Preparo)
  /**
   * Pegando a instância do serviço que nosso módulo
   * de teste (TestBed) está utilizando.
   * */
  const todosService = TestBed.inject(TodosService);
  /**
   * O framework de testes Jasmine fornece o método spyOn(),
   * que permite o acesso dinâmico à determinadas funções.
   * No caso, estaremos espionando o método getToDos(),
   * do nosso serviço, para verificarmos se o mesmo foi chamado.
   * */
  const getToDosSpy = spyOn(todosService, 'getToDos');

  // Act (Ação)
  component.getTodos();

  // Assert (Validação)
  expect(getToDosSpy).toHaveBeenCalled();
});
```

Fonte: Autoria Própria

Executando o teste, verifica-se que, ao chamar o método getToDos() do componente ClasseExemplo, o resultado esperado de chamar a função getToDos() da camada de serviço TodosService, é atingido.

Figura 6 – Cenário de Testes 1: Teste Unitário 4



Fonte: Autoria Própria

4.2. Segundo teste: Validando comportamentos

Além de realizar a testagem lógica do *software*, o desenvolvedor *front-end* deve se atentar ao comportamento esperado da página, conforme o usuário final interage com os elementos HTML contidos na mesma. É preciso garantir não só que o *software* funcione, mas também que todos os elementos HTML sejam renderizados e tenham seus comportamentos validados, de acordo.

Neste exemplo, será realizada a testagem de um botão utilizado para paginação, que emite um evento ao ser clicado.

Figura 7 – Cenário de Testes 2: Elemento Botão

Fonte: Autoria Própria

Figura 8 – Cenário de Testes 2: Elemento Botão HTML

```
<!-- HTML -->
<button
  (click)="getToDosNext(page)"
  class="c-todos-list__navigation-next"
>
  Próxima
</button>
```

Fonte: Autoria Própria

Ao ser clicado, a função getToDosNext() é chamada, e a variável numérica “page”, é passada como parâmetro. A função getToDosNext() se encontra dentro da classe de exemplo.

Figura 9 – Cenário de Testes 2: Função Exemplo

```
public getToDosNext(page: number): void {
  this.todosService.getToDosNext(page).subscribe((response) => {
    this.todos = response.results;
  });
}
```

Fonte: Autoria Própria

Para testar o comportamento do botão, pouco importa o que a função chamada faz, pois a mesma deverá ser testada de forma análoga ao exemplo anterior, separadamente. Neste caso, deve-se atentar apenas ao que é esperado sobre comportamento do botão: chamar a função getToDosNext(), passando a variável numérica “page” como parâmetro, e o elemento deve ser renderizado com o texto “Próxima”.

Inicia-se o desenvolvimento do teste, escrevendo o comportamento esperado.

Figura 10 – Cenário de Testes 2: Testes Unitários 1

```
it('should have "Próxima" button', () => {
  // Arrange (Preparo)

  // Act (Ação)

  // Assert (Validação)
  expect(buttonElement.nativeElement.textContent).toBe('Próxima');
  expect(getNextSpy).toHaveBeenCalled();
});
```

Fonte: Autoria Própria

Escreve-se a ação de clique no botão.

Figura 11 – Cenário de Testes 2: Testes Unitários 2

```
it('should have "Próxima" button', () => {
  // Arrange (Preparo)

  // Act (Ação)
  buttonElement.nativeElement.click();

  // Assert (Validação)
  expect(buttonElement.nativeElement.textContent).toBe('Próxima');
  expect(getNextSpy).toHaveBeenCalled();
});
```

Fonte: Autoria Própria

Finalmente, verifica-se o que é necessário para se executar o teste: cria-se um espião para a função que deve ser chamada, e aloca-se, na variável buttonElement, o elemento do botão que está sendo testado, por meio de sua classe.

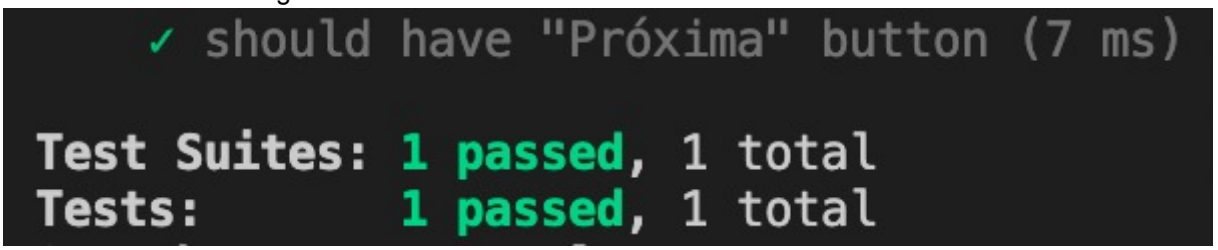
Figura 12 – Cenário de Testes 2: Testes Unitários 3

```
it('should have "Próxima" button', () => {
  // Arrange (Preparo)
  const getNextSpy = spyOn(component, 'getToDosNext');
  const buttonElement = fixture.debugElement.query(
    By.css('c-todos-list__navigation-next')
  );

  // Act (Ação)
  buttonElement.nativeElement.click();

  // Assert (Validação)
  expect(buttonElement.nativeElement.textContent).toBe('Próxima');
  expect(getNextSpy).toHaveBeenCalled();
});
```

Fonte: Autoria Própria

Nota-se que, por mais que os testes escritos estejam válidos, ainda não foi realizada uma cobertura completa do cenário: o teste ainda não contempla o que é esperado do parâmetro “page”, passado na chamada da função getToDosNext().

Figura 13 – Cenário de Testes 2: Testes Unitários 4



Fonte: Autoria Própria

Dessa forma, o teste está incompleto, e é considerado um falso positivo. Exemplifica-se facilmente a falha, ao passar como parâmetro para testar a chamada da função, por exemplo, uma variável do tipo *string*, ao invés de *number*.

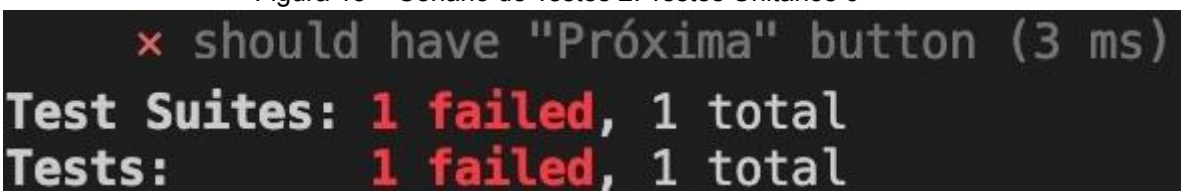
Figura 14 – Cenário de Testes 2: Testes Unitários 5

```
// Assert (Validação)
expect(buttonElement.nativeElement.textContent).toBe('Próxima');

/*O parâmetro page === 'teste' é uma string
*  porém, a função requer um parâmetro numérico
*/
expect(getNextSpy).toHaveBeenCalledWith('teste');
```

Fonte: Autoria Própria

Executando o teste, o mesmo retorna uma falha.

Figura 15 – Cenário de Testes 2: Testes Unitários 6



Fonte: Autoria Própria

Criando-se uma variável “expectedPage”, para representar o parâmetro “page”, garantimos que o teste está completo.

Figura 16 – Cenário de Testes2: Testes Unitários 7

```
it('should have "Próxima" button', () => {
  // Arrange (Preparo)
  const getNextSpy = spyOn(component, 'getToDosNext');
  const buttonElement = fixture.debugElement.query(
    By.css('c-todos-list__navigation-next')
  );
  /** Uma das vantagens de se generalizar os valores
   *  utilizados nos testes, é garantir que o cenário
   *  irá funcionar para qualquer valor.
   */
  const mockTotalPages = Math.random() * 100;
  /** A variavel expectedPage não fica restrita
   *  a um valor fixo
   */
  const expectedPage = Math.floor(Math.random() * mockTotalPages) + 1;

  // Act (Ação)
  buttonElement.nativeElement.click();

  // Assert (Validação)
  expect(buttonElement.nativeElement.textContent).toBe('Próxima');
  expect(getNextSpy).toHaveBeenCalledWith(expectedPage);
});
```

Fonte: Autoria Própria

Executando novamente o teste, obtém-se o resultado esperado.

Figura 17 – Cenário de Testes 2: Testes Unitários 8

```
    ✓ should have "Próxima" button (7 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
```

Fonte: Autoria Própria

4.3. Terceiro teste: Falsos positivos

Neste último exemplo, será elaborado mais um caso de testes passível de gerar resultados falsos positivos, como o problema encontrado no teste anterior, explorando estruturas condicionais.

Para realizar esta análise, foi escolhido um cenário matemático, para facilitar a compreensão. A classe exemplo a seguir, pertence a uma aplicação de plotagem

de gráficos, e possui um método que checa o sinal da função descontínua $f(x) = \frac{1}{x}$, recebendo como parâmetro um valor numérico $x \in R$, e retorna um booleano.

Figura 18 – Cenário de Testes 3: Classe exemplo

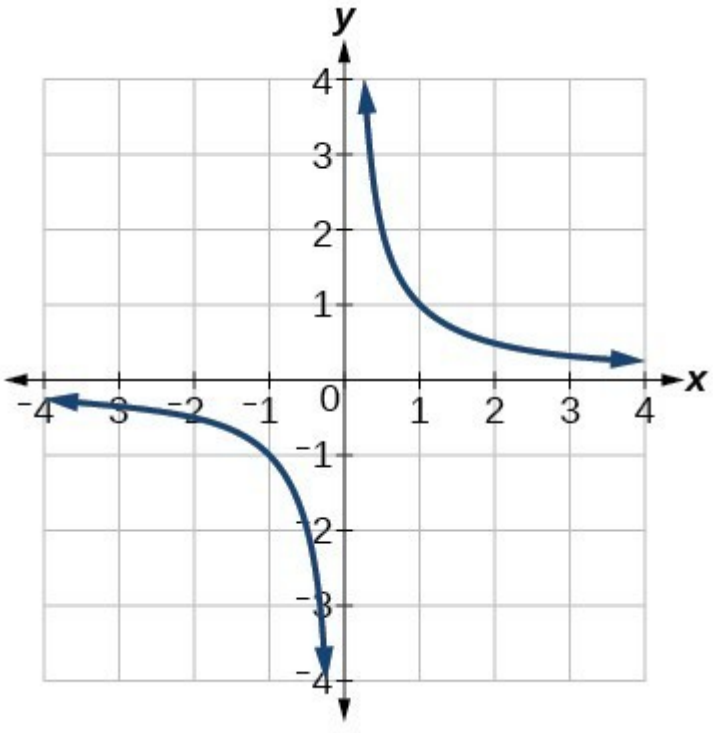
```
export class ClasseExemplo2 {
  public checkSign(x: number): boolean {
    let positive: boolean;
    // retorna undefined no ponto em que a função é descontínua
    if (x === 0) {
      return undefined;
    }
    if (1 / x > 0) {
      positive = true;
    }
    return positive;
  }
}
```

Fonte: Autoria Própria

Ao testar a função checkSign(), deve-se notar que:

1. $f(x) = \frac{1}{x}$ é descontínua e não possui valor real quando $x = 0$;
2. $f(x) = \frac{1}{x}$ é positiva quando $x > 0$;
3. $f(x) = \frac{1}{x}$ é negativa quando $x < 0$.

Figura 19 – Cenário de Testes 3: Gráfico de Função $f(x) = \frac{1}{x}$



Fonte: Autoria Própria

De maneira geral, o preparo do teste é realizado de forma simples, seguindo o mesmo padrão de testes “AAA”, utilizado nos cenários anteriores: define-se o que é esperado do método, realiza-se a chamada do método, e, por fim, inicializa-se as variáveis necessárias para a execução do mesmo.

Figura 20 – Cenário de Testes 2: Testes Unitários 1

```
it('should have checkSign function', () => {
  // Arrange (Preparo)
  const mockValue = Math.random();
  const expectedReturn = true;

  // Act (Ação)
  const checkSignReturn = component.checkSign(mockValue);

  // Assert (Validação)
  expect(checkSignReturn).toEqual(expectedReturn);
});
```

Fonte: Autoria Própria

Executando o teste, temos um falso positivo, pois a validação foi feita exclusivamente para o cenário onde o parâmetro x recebe um valor positivo, e o método checkSign() é esperado retornar o valor *true*.

Figura 21 – Cenário de Testes 2: Testes Unitários 2

```
    ✓ should have checkSign function (7 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
```

Fonte: Autoria Própria

No caso de $x = 0$, ou $x < 0$, o teste não faz validação alguma. Portanto, deve-se desenvolver um teste para cada um dos cenários.

Figura 22 – Cenário de Testes 2: Testes Unitários 3

```
it('should have checkSign function, x > 0 ', () => {
  // Arrange (Preparo)
  const mockValue = Math.random();
  const expectedReturn = true;

  // Act (Ação)
  const checkSignReturn = component.checkSign(mockValue);

  // Assert (Validação)
  expect(checkSignReturn).toEqual(expectedReturn);
});

it('should have checkSign function, x < 0', () => {
  // Arrange (Preparo)
  const mockValue = Math.random() * -1;
  const expectedReturn = false;

  // Act (Ação)
  const checkSignReturn = component.checkSign(mockValue);

  // Assert (Validação)
  expect(checkSignReturn).toEqual(expectedReturn);
});

it('should have checkSign function, x === 0 ', () => {
  // Arrange (Preparo)
  const mockValue = 0;
  const expectedReturn = undefined;

  // Act (Ação)
  const checkSignReturn = component.checkSign(mockValue);

  // Assert (Validação)
  expect(checkSignReturn).toEqual(expectedReturn);
});
```

Fonte: Autoria Própria

Executando os testes, verificamos que todos os cenários foram validados.

Figura 23 – Cenário de Testes 2: Testes Unitários 4

```
    ✓ should have checkSign function, x > 0 (8 ms)
    ✓ should have checkSign function, x < 0 (1 ms)
    ✓ should have checkSign function, x === 0 (1 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
```

Fonte: Autoria Própria

CONCLUSÃO

Tendo em vista a relevância da qualidade de *software* no cenário atual, este trabalho buscou avaliar e contribuir com a área de teste de *software*, analisando as atividades dentro desse escopo.

Esta monografia contribui por meio de pesquisas e estudos a respeito de teste de *software*, levantando seu papel dentro do desenvolvimento de *software*, destacando a correspondência entre testes e desenvolvimento de *software*, e a importante união que deve ocorrer entre essas duas disciplinas.

Em seguida, destacou-se a fase de testes unitários, e como as atividades nessa fase de testes, elevam a qualidade do *software*. Retratou-se também cenários de teste, demonstrando técnicas e padrões de teste, e como o pensamento crítico deve estar presente, para opor quaisquer conformidades, e garantir que o teste escrito de fato garantirá robustez ao código.

Neste contexto, introduziu-se exemplos onde os testes podem induzir ao erro, enfatizando que o desenvolvedor deve se dedicar a refletir sobre o porquê testar, e avaliar o código que está testando, os possíveis cenários e condições que necessitam ser testadas, escrevendo sempre expressões gerais, a fim de evitar cenários ideais, que produzem resultados enganosos.

A partir da metodologia contemplada no padrão de testes “AAA”, verificou-se a praticidade de se estruturar a escrita dos testes, e, consequentemente, a facilidade de se compreender os testes escritos.

Dessa forma, conclui-se a relevância dos testes unitários para garantir qualidade, confiabilidade, manutenibilidade, performance e robustez geral de aplicações *web*.

REFERÊNCIAS BIBLIOGRÁFICAS

ANGULAR. GETTING STARTED. Disponível em: <https://angular.io/guide/setup-local>. Acesso em: 30 mai. 2022.

ANGULAR. DEVELOPER GUIDES. Disponível em: <https://angular.io/guide/testing>. Acesso em: 30 mai. 2022.

BEHAVIOR-DRIVEN DEVELOPMENT. Disponível em: <https://cucumber.io/docs/bdd/>. Acesso em: 02 mai. 2022.

BURNSTEIN, I. Practical Software Testing: a process-oriented approach. Nova Iorque: Springer-Verlag, 2003.

CRISPIN, L.; GREGORY, J. Agile Testing: a practical guide for testers and agile teams. Boston: Pearson Education, 2009.

ELLIMS, M; BRIDGES, J; INCE, D. C. The economics of unit testing. Empirical Software Engineering, v. 11, n. 1, p. 5-31, 2006.

FEWSTER, M.; GRAHAM, D. Software Test Automation: effective use of test execution tools. Boston: ACM Press, 1999.

HUNT, A.; THOMAS, D. Pragmatic Unit Testing: In Java with JUnit. Estados Unidos: The Pragmatic Bookshelf, 2003.

JASMINE. JASMINE 4.1. Disponível em: <https://jasmine.github.io/pages/getting_started.html>. Acesso em: 25 mar. 2022.

JESTJS. JEST 27.2. Disponível em: <https://jestjs.io>. Acesso em: 3 mar. 2022.

KHORIKOV V. Unit Testing Principles, Practices, and Patterns. Simon and Schuster, 2020.

MYERS, G. J. The Art of Software Testing. 2.ed. Nova Jersey: John Wiley & Sons, 2004.

NUNES, A. Teste Unitário E O Padrão AAA (Arrange, Act, Assert). Disponível em: <https://medium.com/@alamonunes/teste-unit%C3%A1rio-e-o-padr%C3%A3o-aaa-arrange-act-assert-cb81d587368a>. Acesso em: 30 mai. 2022.

O que são testes automatizados?. Filho da nuvem. Youtube. 10 fev. 2020. 7min55s. Disponível em: <https://www.youtube.com/watch?v=-TDgy8lhq-k>. Acesso em: 30 mai. 2022.

Qual a diferença entre TESTE UNITÁRIO e de integração? (Pirâmide de testes). Filho da nuvem. Youtube. 2 mar. 2020. 13min45s. Disponível em: <https://www.youtube.com/watch?v=IFzl5yWjbb4>. Acesso em: 30 mai. 2022.

ROCHA, F. G. TDD: fundamentos do desenvolvimento orientado a teste. 2013. Disponível em: <https://www.devmedia.com.br/tdd-fundamentos-do-desenvolvimento-orientado-a-testes/28151>. Acesso em: 02 mai. 2022.

SILVA, G. BDD na prática: entenda o que é e como funciona. Disponível em: <https://coodesh.com/blog/candidates/metodologias/bdd-na-pratica-entenda-o-que-e-e-como-funciona/>. Acesso em: 02 mai. 2022.

SPIRLANDELI, C.; ROLAND, C. E. F. A utilização de testes automatizados no desenvolvimento de software. 2019.

HUSSAIN, A. Angular Unit Testing. Disponível em: <https://codecraft.tv/courses/angular/unit-testing/overview/>. Acesso em: 13 mai. 2022

SUN, Y. Unit Testing. In: Practical Application Development with AppRun. Apress, Berkeley, CA, 2019. p. 247-264.