UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA - CCET
DEPARTAMENTO DE COMPUTAÇÃO - DC

Marcelina Maye Abaga Maye

Um estudo sobre testes automatizados em aplicações React.js

SÃO CARLOS - SP
2025

Marcelina Maye Abaga Maye

Um estudo sobre testes automatizados em aplicações React.js

Trabalho de conclusão de curso apresentada ao Departamento de Computação da Universidade Federal de São Carlos, para obtenção do título de bacharel em Ciência da Computação.

Orientador: Prof. Dr. André Takeshi Endo

SÃO CARLOS - SP
2025

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia - CCET
Departamento de Computação

Comissão avaliadora

Membros da comissão examinadora que avaliou e aprovou a Defesa do Trabalho de Conclusão de Curso da candidata Marcelina Maye Abaga Maye, realizada em 20/02/2025

Prof. Dr. André Takeshi Endo
Instituição: Universidade Federal de São Carlos

Prof. Dr. Auri M. R. Vincenzi
Instituição: Universidade Federal de São Carlos

Prof. Dr. Daniel Lucrédio
Instituição: Universidade Federal de São Carlos

Dedicatória

Dedico este trabalho primeiramente a Deus, fonte de toda inspiração, ao meu pai Atanasio e a minha mãe Marcelina, pilares do meu ser e apoio incondicional.

AGRADECIMENTO

Agradeço primeiramente a Deus, sem o qual nada disso seria possível. Agradeço aos meus pais, Atanásio e Marcelina, por serem sempre este ombro amigo e a válvula de escape quando tudo parece não ter solução.

Agradeço a mim mesma por nunca desistir, mesmo quando sobravam motivos e por encarar cada dia com otimismo.

Agradeço ao Prof. Dr. André Takeshi Endo por aceitar ser meu orientador e por me oferecer todo suporte necessário.

Agradeço ao Departamento de Computação por todos os ensinamentos transmitidos.

Suba o primeiro degrau com fé, não é necessário que você veja toda a escada. Apenas de o primeiro passo. - Martin Luther King

RESUMO

Garantir a qualidade, o desempenho, a redução de tempo e evitar o retrabalho em correção de trechos de código manualmente são algumas das exigências do desenvolvimento de aplicações web modernas. Por isso, implementar testes automatizados é a melhor opção em termos de eficiência, redução de custos do projeto e melhora da produtividade. Dada a ampla utilização do *framework* React.js, uma biblioteca JavaScript para construir interface de usuário, torna-se importante obter evidências sobre como essas aplicações incluem testes automatizados. Por isso, este estudo centra-se na exploração de recursos de automação de testes em aplicativos web que utilizam React.js para desenvolvimento do *Front-End*. Para isto, foram estudadas a biblioteca React.js e as ferramentas de automação de testes que esta utiliza.

Primeiramente, foram selecionados 20 projetos *open source* React.js que implementem testes automatizados para análise segundo os critérios estabelecidos. Em seguida, realizou-se a clonagem dos projetos do GitHub e foram executados os testes. Também foi construída uma tabela que recolhe os seguintes dados: número de estrelas, *commits*, data do último *commit*, *frameworks* de teste, linguagem de programação, número de teste, LoC de teste, LoC de produção, estrutura de teste e relatório de cobertura. Com fundamento nestes dados obtidos, foram respondidas as questões de pesquisa definidas.

Por fim, os resultados apontam que dentre os *frameworks* de teste destacam, React Testing Library e Jest. A linguagem de programação mais utilizada foi o JavaScript e também foram discutidas as LoC de testes em relação as LoC de produção e o número de testes em relação ao LoC de produção. Os resultados mostram que num total de 20 projetos, 5 apresentaram maior proporção de LoC de teste em relação ao teste de produção e 7 apresentaram maior quantidade de teste em relação ao LoC de produção.

Palavras-chave: Testes automatizados. React.js. Jest. React Testing Library.

ABSTRACT

Ensuring quality, performance, reducing time and avoiding rework in manually correcting code snippets are some of the requirements of developing modern web applications. Therefore, implementing automated tests is the best option in terms of efficiency, reducing project costs and improving productivity. Given the widespread use of the React.js framework, a JavaScript library for building user interfaces, it is important to obtain evidence on how these applications include automated tests. Therefore, this study focuses on exploring test automation resources in web applications that use React.js for Front-End development. To this end, the React.js library and the test automation tools it uses were studied.

First, 20 open source React.js projects that implement automated tests were selected for analysis according to the established criteria. Then, the projects were cloned from GitHub and the tests were run. A table was also constructed that collects the following data: number of stars per commit, date of the last commit, test frameworks, programming language, number of tests, test LoC, production LoC, test structure, and coverage report number of stars. Based on this data obtained, the defined research questions were answered.

Finally, the results indicate that among the test frameworks, React Testing Library and Jest stand out. The most used programming language was JavaScript, and the test LoC in relation to the production LoC and the number of tests in relation to the production LoC were also discussed. The results show that out of a total of 20 projects, 5 had a higher proportion of test LoC in relation to the production LoC and 7 had a higher number of tests in relation to the production LoC.

Keyword: Automatic testing. React.js. Jest. React testing library.

Lista de Figuras

1 Exemplo de componente React.js . . . 14
2 Exemplo de teste unitário . . . 15
3 Exemplo de teste de integração . . . 16
4 Comando utilizado para instalar o gerenciador de pacotes de Node.js . . . 20
5 Comando yarn . . . 20
6 Comando utilizado para executar os testes . . . 21
7 Testes sendo executados . . . 21
8 Relatório de cobertura . . . 22
9 Frequência dos *frameworks* de teste . . . 25
10 Gráfico de estrutura dos testes . . . 27
11 Gráfico de proporção de Loc de teste em relação ao Loc de produção . . . 28
12 Gráfico de quantidade de testes em relação a Loc de produção . . . 29
13 Gráfico de *Code coverage* . . . 30
14 Exemplo de teste unitário . . . 31
15 Exemplo de teste de integração . . . 32
16 Exemplo de teste de sistema . . . 33

Lista de Tabelas

1 Características dos projetos 24
2 Distribuição de projetos por linguagem de programação 26
3 Cobertura de código dos projetos por categoria (%) 31

Conteúdo

Lista de Figuras 8

Lista de Tabelas 9

1 INTRODUÇÃO 11

1.1 Justificativa 12
1.2 Objetivos 13
1.3 Estrutura do Trabalho 13

2 REVISÃO BIBLIOGRÁFICA 14

2.1 React.js 14
2.2 Conceitos básicos de Testes de Software 15
2.3 Recursos de testes no React.js 16
2.4 Trabalhos Relacionados 17
2.5 Considerações Finais 18

3 METODOLOGIA DO ESTUDO 19

3.1 Seleção de projetos *Open Source* 19
3.2 Execução e Análise dos Testes 20
3.3 Ameaças a validade 22

4 ANÁLISE DE RESULTADOS 24

4.1 Caracterização dos projetos selecionados 24
4.2 QP1 - Quais os *frameworks* usados em projetos React.js? 25
4.3 QP2 - Quais linguagens de programação usadas nos projetos? 26
4.4 QP3 - Qual estrutura de teste adotada? 27
4.5 QP4 - Quantitade de LoC de teste em relação a LoC de produção? 27
4.6 QP5 - Quantidade de testes em relação ao Loc de produção? 28
4.7 QP6 - Quantos projetos habilitam code coverage? 29
4.8 Ilustração dos tipos de testes observados. 31

5 CONSIDERAÇÕES FINAIS 34

Referências 35

1 INTRODUÇÃO

O crescimento do uso de aplicações web modernas tem demandado dos desenvolvedores a criação de interfaces gráficas de usuário (GUIs) mais rápidas, interativas e escaláveis a fim de otimizar a experiência do usuário e facilitar a manutenção das mesmas. Uma aplicação web é um programa em que as GUIs são executadas em um navegador da web (Madsen, Lhotak e Tip, 2020). Inicialmente, as GUIs eram desenvolvidas em Hyper Text Markup Language (HTML), mas enfrentavam alguns problemas como a fragilidade do código devido à mutação direta do Document Object Model (DOM) e a dificuldade de processar aplicações que requerem atualização incremental. De acordo com Gackenheimer (2015), o React.js é um *framework* JavaScript, criado originalmente por engenheiros do Facebook para resolver os desafios envolvidos no desenvolvimento de GUIs complexas com conjuntos de dados que mudam ao longo do tempo. Sendo assim, React.js, uma das bibliotecas mais utilizadas e amplamente popular em sites modernos. Segundo a pesquisa do StackOverflow (2024), o React.js é o segundo *framework* mais utilizado com uma pontuação de uso de 39,5%. O React.js se destaca por oferecer uma abordagem declarativa e orientada a objetos.

O React.js adota uma estrutura de componentes reutilizáveis, que permitem a criação de GUIs de forma modular e escalável. Um componente é uma parte das GUIs que tem sua própria lógica e aparência (React, 2023). Um componente React.js é constituído por um conjunto de propriedades que representam parâmetros de entrada necessários para configurar o componente, um estado interno que consiste em valores que podem mudar ao longo do tempo e um método de renderização (Madsen, Lhotak e Tip, 2020). Os componentes não só facilitam a manutenção e atualização do código, mas também melhoram a colaboração entre equipes de desenvolvimento. O React.js utiliza uma sintaxe JavaScript XML (JSX) semelhante a HTML porém mais rigorosa. O JSX é uma extensão de sintaxe para javaScript que permite aos desenvolvedores escrever códigos semelhantes ao HTML dentro de um arquivo JavaScript (Kinsta, 2023). Além disso, o React.js introduz a ideia do Virtual DOM, uma representação em memória da interface do usuário que otimiza atualizações e renderizações, resultando em uma performance significativamente melhorada.

No entanto, o desenvolvimento envolve um ciclo contínuo de criação, manutenção e melhoria, onde a qualidade do produto final é essencial para atender as funcionalidades especificadas no documento de requisitos (Pressman et al., 2010). Sendo assim, os testes de *software* desempenham um papel crucial, permitindo verificar se a aplicação funciona como esperado. De acordo com Delamaro et al. (2013), os testes de *software* permitem que erros sejam descobertos antes do lançamento do *software* com o intuito de garantir que tanto a construção do *software* como o *software* em si estejam em conformidade com o especificado. Existem várias fases na atividade de testes, destacando-se três: os testes

unitários, que se concentram em verificar o comportamento de componentes ou funções individuais da aplicação; os testes de integração, que avaliam como diferentes módulos ou componentes trabalham juntos; e os testes de sistema, que validam a aplicação completa, incluindo seus requisitos técnicos e funcionais (Delamaro et al., 2013).

Porém, realizar testes de forma manual pode ser um processo árduo, propenso a erros humanos e pouco eficiente. Para garantir ainda mais a qualidade do *software*, implementa-se testes automatizados como uma alternativa eficaz, proporcionando uma metodologia sistemática para garantir a funcionalidade e a integridade das aplicações. De acordo com Bernardo e Kon (2008), os testes automatizados são programas ou *scripts* simples que exercitam funcionalidades da aplicação e fazem verificações automáticas de possíveis problemas. Esses testes podem ser executados repetidamente com alta precisão, economizando tempo e recursos a longo prazo.

Os testes em React.js podem ser feitos de duas formas: renderizando árvores de componente e executando a aplicação completa. A primeira refere-se ao teste de unidade e teste de integração, que testam módulos do software separadamente, enquanto a segunda refere-se ao teste de sistema ou *End-to-End*(E2E) que testam a aplicação como um todo. Ao escolher a ferramenta de teste no React.js, é importante considerar fatores como a velocidade de iteração em relação ao ambiente real e a quantidade de *mock*. Apesar da variedade de ferramentas disponíveis, o React.js recomenda duas opções principais: o Jest, um *test runner* JavaScript que permite acessar o DOM através do jsdom e o React Testing Library, um conjunto de utilitários que permitem testar componentes React.js sem depender dos detalhes de implementação.

Por tanto, entender como são criados os testes automatizados no ecossistema React.js e o papel que desempenham, é essencial para garantir a qualidade do *software*, salientando o emprego das melhores práticas de testes automatizados em aplicações React.js.

1.1 Justificativa

O React.js é uma biblioteca que transformou o modo como equipes desenvolvem GUIs ao simplificar a criação de aplicações web através de componentes reutilizáveis e atualizações eficientes no DOM. Sua flexibilidade possibilita a combinação harmoniosa com HTML, permitindo adicionar React.js a uma página HTML já existentes e renderizar componentes interativos em qualquer parte da página (React, 2023). Por outro lado, os testes automatizados, no universo do React.js, são cruciais para garantir o funcionamento adequado das GUIs e prevenir erros provocados por atualizações no código (React Documentation, 2024). Sendo assim, analisar como são implementados os testes automatizados no React.js pode oferecer um rico conhecimento para este tema.

Para entender como os testes automatizados são implementados no React.js, este trabalho propõe analisar testes automatizados em aplicações reais. Com este intuito, serão

coletados dados e estes serão representados em forma de gráficos para compreender as práticas de testes automatizados no React.js.

1.2 Objetivos

O objetivo deste trabalho foi investigar como testes automatizados são desenvolvidos em projetos React.js. Para isso, foram selecionados um conjunto de projetos *open source* que usam React.js e atendem a critérios mínimos estabelecidos, com o intuito de entender como são criados os testes automatizados no ecossistema React.js e saber quais recursos são utilizados para a criação dos mesmos.

1.3 Estrutura do Trabalho

Este trabalho está dividido na seguinte forma: o Capítulo 2 apresenta React.js, conceitos básicos de testes de *software* e os recursos de testes no React.js, além de enunciar os trabalhos relacionados. O Capítulo 3 apresenta a metodologia utilizada para a condução do estudo. No Capítulo 4 têm-se a apresentação e análise dos resultados. Por fim, no Capítulo 5 as conclusões a respeito do que foi estudado neste trabalho são apresentadas.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo são apresentados os conceitos e as principais ferramentas de testes automatizados em aplicações React.js. Portanto, o capítulo é iniciado com React.js na Seção 2.1, os conceitos básicos de testes de *software* na Seção 2.2, os recursos de testes no React.js na Seção 2.3 e os trabalhos relacionados na Seção 2.4.

2.1 React.js

O React.js é uma biblioteca JavaScript desenvolvida pelo Facebook, atualmente Meta, para construir GUIs. O React.js permite criar GUIs a partir de peças individuais chamadas de componente (React, 2023). Os componentes React.js são funções em JavaScript, que permitem utilizar a sintaxe de *tags* chamada de JSX, que é uma extensão do JavaScript popularizada pelo React (React, 2023).

Figura 1: Exemplo de componente React.js

```
1 import React from 'react';
2 function MyButton(){
3     Return(
4         <button>
5         I'm a button
6         <\button>
7     );
8 }
```

Fonte: Adaptado de React.dev

A Figura 1 apresenta um exemplo de componente funcional que cria um botão, denominado `MyButton`, usando React.js. Importa a biblioteca React `import React from 'react'`(linha 1), que é necessária para criar componentes React.js. A função `MyButton()` (linha 2) define o componente funcional, este retorna um bloco JSX que neste caso é o elemento `<button>`. O conteúdo do botão é o texto `"I'm a button"`(linha 5).

React.js permite a junção de componentes, porém não determina como fazer o roteamento e busca de dados. Por isso, para o desenvolvimento de uma aplicação completa com React.js é recomendado o uso de um *framework React full stack* como Next.js ou Remix. Esses *frameworks* são essenciais para o correto funcionamento de aplicações React.js, pois eles fornecem uma série de recursos. Por exemplo, Remix utiliza a técnica *Server Partial Rendering* (SPR) para acelerar o carregamento da página e do Next.js, que é extremamente versátil, suporta SSR e renderiza o lado do cliente (Revelo Community, 2024)

2.2 Conceitos básicos de Testes de Software

Teste de *software* é uma atividade dinâmica, que tem como objetivo executar a aplicação e verificar se o comportamento está de acordo com o esperado (Delamaro et al., 2013). Os testes de *software* são cruciais porque eles permitem garantir que a aplicação satisfaz os requisitos de usuários especificados.

Para testar um *software* deve-se criar uma série de casos de teste com o intuito de identificar falhas e descobrir os possíveis erros. Um caso de teste é uma descrição detalhada de duas classes de entradas: a primeira abrange a configuração de software, a especificação do projeto e o código-fonte; já a segunda inclui a configuração de teste, as ferramentas de teste e os resultados esperados (Pressman et al., 2010).

De acordo com Pressman et al. (2010), existem várias estratégias para testar *software*. Uma delas é aguardar até que a aplicação esteja completamente desenvolvida para então testa-la por inteiro. Outra abordagem consiste em testar cada parte da aplicação à medida que ela é desenvolvida. Por fim, há uma estratégia híbrida, que combina as duas anteriores e é a mais utilizada pelas equipes de desenvolvimento. Essa abordagem adota uma visão incremental dos testes e envolve três tipos principais:

Teste unitário: testa o funcionamento da unidade mais pequena de uma aplicação.

Figura 2: Exemplo de teste unitário

```
1  // função para ser testada
2  function soma(a, b) {
3    return a + b;
4  }
5
6  // teste unitário usando Jest
7  describe('Função soma', () => {
8   test('deve retornar a soma de dois números', () => {
9     expect(soma(2, 3)).toBe(5);
10  });
11
12  test('deve retornar 0 se ambos os números forem 0', () => {
13    expect(soma(0, 0)).toBe(0);
14  });
15 });
```

Fonte: Autoria própria

A Figura 2 apresenta um código de testes unitários em JavaScript usando o *framework* Jest. A função soma() (linha 2 à 4), realiza uma operação simples que recebe dois parâmetros e retorna a soma desses valores. A palavra chave `describe` (linha 7) agrupa os testes que verificam a função `soma()`. Dentro do bloco `describe`, o primeiro teste definido com o método `test` (linha 8), testa a soma de dois números. `expect(soma(2, 3)).toBe(5)` (linha 9) chama a função soma() com os argumentos 2 e 3 e compara o resultado da função com o valor esperado, que é 5. Se a função soma retornar o valor correto `(5)`, o teste será aprovado. Caso contrário, o teste falhará. O segundo teste (linha

12) testa a soma de zeros, com intuito de garantir que a função retorna 0 quando ambos os parâmetros forem 0.

Teste de integração: testa como diferentes componentes de um sistema interagem entre si.

Figura 3: Exemplo de teste de integração

```
1    // Teste de infraestrutura verificando conexão real com o banco
2  const { getUserById } = require('./database'); // Importa função real
3
4  describe('Infra: Teste de Conectividade com o Banco', () => {
5      test('Deve retornar um usuário do banco se o ID for válido', async () => {
6          const user = await getUserById(1); // Consulta real ao banco
7          expect(user).toBeDefined(); // Garante que um usuário foi retornado
8          expect(user.name).toBe('Marcelina'); // Nome esperado para o ID 1
9      });
10
11     test('Deve lançar erro se o banco estiver indisponível', async () => {
12         jest.spyOn(global, 'fetch').mockRejectedValue(new Error('Falha na conexão'));
13         await expect(getUserById(999)).rejects.toThrow('Falha na conexão');
14     });
15 });
```

Fonte: Fonte: Autoria própria

A Figura 3 apresenta a implementação de um teste de infraestrutura para verificar a conectividade e funcionalidade do banco de dados. Primeiramente, a função `getUserById` é importada de um módulo real (linha 1). No primeiro teste (linha 4), verifica-se se um usuário é corretamente retornado do banco ao chamar a função com um `ID` válido (linha 5). O teste garante que o usuário existe (linha 6) e que seu nome corresponde ao esperado (linha 7). Já o segundo teste (linha 9) avalia um cenário de falha de infraestrutura, simulando uma indisponibilidade do banco de dados (linha 10). Para isso, a função *fetch* do JavaScript é mockada para sempre rejeitar a chamada com um erro (linha 11). Em seguida, a expectativa é que chamar `getUserById` com um ID inexistente (linha 12) resulte em uma exceção lançada, indicando falha na conexão.

Teste de sistema ou End-To-End (E2E): centra-se no comportamento geral de uma aplicação tanto funcional como não funcional, testando-a como um todo.

Os testes de *software* são fundamentais para melhorar a qualidade do *software*. Funcionam como um método para garantir que os sistemas desenvolvidos funcionem corretamente e de acordo com as expectativas dos usuários. A adoção de testes testes automatizados surge como uma estratégia crucial para agilizar esse processo, permitindo a execução repetitiva e eficiente de todos os casos de testes (Bernardo e Kon, 2008).

2.3 Recursos de testes no React.js

No React.js existem várias maneiras de testar os componentes, mas em geral podem ser feitos de duas maneiras: renderizando árvores de componentes e executando uma aplicação completa (React Documentation, 2024). Na hora de escolher a ferramenta de

teste no React.js, é necessário considerar o custo de oportunidade entre a velocidade de interação e o ambiente real, também é importante considerar a quantidade de *mocks* que for usar. Existem vários *frameworks* focados em testes no React.js, porém o React.js recomenda dois: o Jest e o React Testing Library (React Documentation, 2024).

O Jest pode ser descrito como: *"Um framework de teste em JavaScript projetado para garantir a correção de qualquer código JavaScript. Ele permite que você escreva testes com uma API acessível, familiar e rica em recursos que lhe dá resultados rapidamente"* (Jest, 2024). Também pode ser descrito como: *"um framework é um compilador de testes que permite acessar o DOM através do jsdom"* (React, 2023). Portanto, Jest é uma ferramenta de teste JavaScript que visa pela simplicidade, oferece uma configuração mínima e executa testes em processos isolados e paralelos para otimizar o desempenho.

React Testing Library é um conjunto de utilitários que permitem testar componentes React.js sem depender dos detalhes de implementação (React Documentation, 2024). O React Testing Library está focado em simular a forma como os usuários interagem com a interface, visa promover testes sustentáveis, que evitem a dependência de detalhes de implantação. O React Testing Library também incentiva a acessibilidade nos aplicativos, aproximando os testes da experiência real do usuário. Cabe ressaltar que ela não é um executor de teste ou uma ferramenta específica, pode ser usada com qualquer *framework* de teste.

Tanto Jest como React Testing Library são ferramentas fundamentais e complementares no que se refere a testes para aplicações React. Enquanto o Jest atua como estruturador e executor de testes, oferecendo uma ampla gama de funcionalidades assertivas, cobertura de código e *mocks*, o React Testing Library é uma biblioteca que foca mais em testar componentes de uma forma que simula as ações semelhantes às das interações de um usuário real, promovendo assim boas práticas de desenvolvimento do *software*. Estas ferramentas, quando usadas juntas, permitem a criação de testes robustos, sustentáveis e de fácil manutenção, garantindo que o comportamento do aplicativo seja validado da forma mais próxima à experiência do usuário final (React Documentation, 2024). Outros *frameworks* de teste utilizados no React.js são o Cypress, Enzyme e Vitest.

Exemplos de código utilizando *frameworks* Jest e React Testing Library vão ser apresentados e explicados com mais detalhes no Capítulo 4.

2.4 Trabalhos Relacionados

Madsen, Lhotak e Tip (2020) apresentam uma semântica operacional formal para o núcleo do *framework* React, modelando as principais operações de montagem, desmontagem e reconciliação de componentes, bem como a semântica de mudanças de estado. Estes definem a noção de um componente "bem-comportado", onde o método *render* retorna apenas componentes de classificação estritamente inferior e provam que esse bom

comportamento é preservado pelas principais operações.

Hasan et al. (2022) encontram vantagens e desvantagens das ferramentas de teste Jest, Enzyme e Cypress para testar aplicativos React.js e propõem uma ferramenta adequada para diferentes níveis de desenvolvimento de *software* que pode fornecer *software* de melhor qualidade. Se centraram na avaliação do desempenho, tempo de execução, taxa de detecção de erros e facilidade de uso das três ferramentas de teste automatizados como os principais parâmetros para selecionar uma ferramenta de teste adequada. Os autores concluíram que, entre as três ferramentas de teste avaliadas (Jest, Enzyme e Cypress), o Jest obteve a pontuação mais alta, com 17 pontos, o Enzyme ficou muito próximo, com 16 pontos, mas o Cypress ficou para trás, com apenas 12 pontos.

Ferreira e Valente (2023) destacam 2.565 instâncias de *code smells* em 10 projetos populares do GitHub usando React.js, e sua principal contribuição é propor o primeiro catálogo de *code smells* específicos para aplicativos JavaScript baseados em React.js. E, o *code smells* com a maior taxa de remoção é *"Large File"* com 50,5% e o *code smells* com as menores taxas de remoção são *" Inheritance over composition"* com 0.9% e *"Direct DOM Manipulation"* com 14,7%.

Ferreira, Borges e Valente (2024) identificam 69 operações distintas de refatoração para aplicativos React.js, categorizadas em quatro grupos principais: 22 refatorações tradicionais, 6 específicas para JavaScript e CSS, e 41 refatorações específicas ou adaptadas ao React.js. Os autores fornecem um catálogo dessas refatorações, incluindo descrições claras e exemplos de código, para orientar a evolução dos projetos React.js e abordar problemas de design. O artigo mapeia as refatorações identificadas para *smells* de códigos específicos em aplicativos React.js, fornecendo aos desenvolvedores orientação prática sobre como abordar e resolver problemas de design.

2.5 Considerações Finais

Nota-se que as os autores discutem diversos pontos sobre o React.js e testes automatizados em aplicações que utilizam React.js para implementar GUIs, os testes são fundamentais para qualquer *software* que pressa oferecer uma ótima experiência ao usuário. Para entender como são utilizados, este estudo se dedica a analisar e apresentar aplicações com testes automatizados implementados e ferramentas de testes mais utilizadas nestas aplicações, pois com estas, é possível entender algumas preferências e práticas no desenvolvimento de GUIs.

3 METODOLOGIA DO ESTUDO

Este capítulo apresenta o passo-a-passo da realização do estudo proposto. Como o objetivo deste trabalho foi investigar como testes automatizados são desenvolvidos em projetos React.js, foram formuladas as seguintes Questões de Pesquisa (QP):

- **QP1 - Quais os *Frameworks* de testes usados em projetos React.js?**: o intuito desta questão é identificar quais são as ferramentas utilizadas para implementar testes automatizados no React.js.

- **QP2 - Quais linguagens de programação usadas nos projetos?**: a proposta desta questão é identificar em quais linguagens de programação são desenvolvidos os testes.

- **QP3 - Qual estrutura de teste adotada?**: o objetivo desta questão de pesquisa é identificar a estrutura dos testes dentro do projeto.

- **QP4 - Quantidade *Line of Code* (LoC) de teste em relação a LoC de produção?**: o objetivo desta questão de pesquisa é saber o número de linhas de código de teste em relação a LoC de produção de cada projeto.

- **QP5 - Quantidade de testes em relação a LoC de produção?**: o propósito desta pergunta é saber a quantidades de testes em relação a LoC de produção.

- **QP6 - Quantos projetos habilitam *code coverage*?**: o intuito desta questão é identificar projetos que habilitam o relatório de cobertura de código.

Desta forma, este capítulo está organizado da seguinte maneira: A Seção 3.1 apresenta como foi feita a seleção dos projetos *open source*. Na Seção 3.2 detalhou-se sobre a execução e análise dos projetos. Por fim, a Seção 3.3 aborda possíveis ameaças à validade do trabalho proposto e considerações finais.

3.1 Seleção de projetos *Open Source*

A escolha dos projetos para análise foi realizada mediante a busca no GitHub, através das *queries*: *Project open source react.js* e *Site react.js* com o filtro *Most Stars*. O GitHub foi a principal fonte escolhida por conta da sua popularidade como repositório de código aberto, abrigando uma enorme quantidade de código que implementam uma variada gama de tecnologias. Para a seleção dos projetos foram usados os seguintes critérios: Idade do projeto, foram escolhidos apenas projetos com, no máximo, quatro anos desde a sua criação, garantindo que sejam relativamente novos e utilizem tecnologias e práticas modernas. Testes automatizados, essa foi uma exigência mínima para captura dos projetos, uma vez que o mesmo é indispensável para os fins deste estudo. React.js, como foco do estudo, todos os projetos deviam utilizar o *framework* React.js para desenvolvimento das GUIs.

3.2 Execução e Análise dos Testes

Após a seleção dos projetos, foi realizado o processo de execução e análise com o intuito de coletar dados para o estudo. Este processo tem as seguintes etapas:

- **Clonagem e configuração dos projetos**. Cada um dos projetos selecionados foi clonado diretamente do GitHub para um ambiente local, onde foram configurados e preparados para a sua execução. Para instalação das dependências dos projetos, foram utilizados os respectivos gerenciadores de pacotes indicados em cada projeto. A maioria dos projetos utiliza o `npm` (Node Package Manager), que é o gerenciador de pacotes de Node.js, a instalação com este gerenciador é feito como mostrado na Figura 4.

  A Figura 4, apresenta o comando `npm install` usado para instalar as dependências de um projeto JavaScript que estão listadas no arquivo `package.json`.

  Figura 4: Comando utilizado para instalar o gerenciador de pacotes de Node.js

  ```
  Maye@DESKTOP-IKRV18Q MINGW64 ~/ofnotes (master)
  $ npm install
  ```

  Fonte: Autoria própria

  Também foi utilizado Yarn que é uma alternativa ao npm criado pelo Facebook em 2016.

  A Figura 5, apresenta o comando `yarn install` usado para instalar as dependências de um projeto JavaScript que estão listadas no arquivo `package.json`.

  Figura 5: Comando yarn

  ```
  Maye@DESKTOP-IKRV18Q MINGW64 ~/ofnotes (master)
  $ yarn install
  ```

  Fonte: Autoria própria

  Utilizar estes gerenciadores envolve o download automático e configuração de todas as dependências, bem como da própria linguagem e *frameworks* utilizados.

- **Execução dos testes automatizados**. Assim que estiverem configurados, foram executados os testes automatizados de cada projeto, através do comando: `npm run test` ou `yarn run test`. Como mostrado na Figura 6.

Figura 6: Comando utilizado para executar os testes

```
Maye@DESKTOP-IKRV18Q MINGW64 ~/retro-board/frontend (develop)
$ npm run test
```

Fonte: Autoria própria

A Figura 7, mostra o resultado de testes automatizados executados com sucesso do projeto Build-ui. A parte inferior do terminal apresenta um resumo da execução dos testes: *Test Suites:* no total de 35, 35 passaram. *Tests:* no total de 220, 220 passaram. *Snapshots:* 8 total. O projeto pode estar utilizando *snapshot testing* para verificar se a interface não sofreu mudanças inesperadas. O tempo total de execução dos testes foi 19.412s

Figura 7: Testes sendo executados

```
 PASS  src/tests/reducers/tree/index/addIndex.test.js
 PASS  src/tests/reducers/tree/index/toggleIndex.test.js
 PASS  src/tests/reducers/tree/index/clearListIndex.test.js
 PASS  src/tests/reducers/tree/index/clearIndex.test.js
 PASS  src/tests/components/DnDBuilder.test.js
 PASS  src/tests/hooks/useTools.test.js
 PASS  src/tests/hooks/dnd/useToolDnD.test.js

Test Suites: 35 passed, 35 total
Tests:       220 passed, 220 total
Snapshots:   0 total
Time:        19.412 s
Ran all test suites.
```

Fonte: Autoria própria

- **Obtenção do relatório de cobertura**. Depois de executar os testes foram gerados os relatórios de cobertura de testes com o intuito de saber quanto do projeto está sendo coberto pelos testes, para isso foram utilizados estes comandos: `npm test - -coverage` ou `yarn test - -coverage`.

  A Figura 8, apresenta um relatório de cobertura de código, que indica quanto do código foi testado. O relatório está organizado em colunas, que representam diferentes métricas de cobertura de testes para os arquivos do projeto. A análise detalhada dessas colunas é a seguinte: *File* (Arquivo): Lista os arquivos do projeto. *% Stmts* (*Statements* - Declarações executáveis): Indica a porcentagem de declarações de código que foram cobertas por testes. *% Branch* (*Branches* - Ramificações de código) Mede a cobertura de diferentes fluxos de execução no código. *% Funcs* (*Functions* - Funções testadas): Percentual de funções cobertas pelos testes. *% Lines* (Linhas executadas): Percentual de linhas do código-fonte que foram executadas pelos testes. *Uncovered Line* s (Linhas não cobertas): Lista as linhas específicas do código que não foram cobertas pelos testes, indicando potenciais áreas sem testes.

Figura 8: Relatório de cobertura

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	82.26	67.66	81.6	85.16	
components	59.45	56.52	49.45	67.95	
Builder.js	100	100	100	100	
BuilderProvider.js	100	100	100	100	
DnDBuilder.js	100	100	100	100	
DnDEvents.js	31.79	15.38	19.51	38.05	...3,79-86,90-101,106-121,126-161,165-180,184-228
DnDListener.js	100	100	100	100	
DnDSource.js	77.94	71.11	62.85	91.66	38,144,259,263,267,271,275,279
Node.js	100	100	100	100	
Panel.js	100	100	100	100	
View.js	100	100	100	100	
Workspace.js	100	100	100	100	
context	100	100	0	100	
BuilderContext.js	100	100	0	100	
ViewContext.js	100	100	100	100	
history/enhancers	100	100	100	100	
accumulator.js	100	100	100	100	

Fonte: Autoria própria

Além de permitir a visualização no terminal, a biblioteca de testes Jest, que também tem suporte nativo para relatórios de cobertura de códigos, cria um arquivo HTML do relatório detalhado.

Foram coletadas as informações sobre as ferramentas de teste por meio da análise do arquivo `package.json`, e as linguagens de programação utilizadas foram identificadas na aba *About*, localizada no canto esquerdo da página do GitHub de cada projeto. Para avaliar a estrutura dos projetos, observou-se como os arquivos de teste estão organizados. Quanto aos *commits*, analisou-se o volume e a diversidade das contribuições realizadas, refletindo o nível de engajamento da comunidade com o projeto. Além disso, foram contabilizadas tanto LoC dos testes utilizando o comando `cloc -match-f='.test.|.spec.'-` quanto as LoC totais do projeto, por meio do comando `cloc .`.

3.3 Ameaças a validade

Uma ameaça potencial ao estudo é a seleção reduzida de projetos, que pode não ser representativa do universo total. Outra preocupação refere-se à execução dos testes automatizados. Dependendo do estágio de desenvolvimento dos projetos, alguns testes podem falhar devido a código desatualizado ou configurações incompatíveis, o que afetaria os índices de cobertura e, consequentemente, a análise de qualidade do software.

Uma limitação adicional identificada diz respeito à ausência de relatórios de cobertura de código em determinados projetos. Essa ausência pode ocorrer por diversos motivos, como desafios técnicos e incompatibilidades entre as ferramentas de cobertura e o ambiente de desenvolvimento. Por exemplo, códigos que dependem de bibliotecas externas podem dificultar a análise da cobertura. Em projetos de grande porte, que envolvem múltiplas ferramentas e linguagens, a configuração dos relatórios pode se revelar particularmente complexa. Ademais, há casos em que equipes desenvolvedoras assumem que a simples

existência de testes automatizados é suficiente, negligenciando a análise detalhada da cobertura de código.

Após a discussão dos desafios e limitações, o Capítulo 4 apresenta os resultados da análise. Esses resultados respondem às questões de pesquisa propostas.

4 ANÁLISE DE RESULTADOS

Neste capítulo serão apresentados e discutidos os resultados obtidos durante a execução do estudo citado no capítulo anterior. A estrutura do capítulo é a seguinte: a Seção 4.1 analisa características do projeto. Nas Seções 4.2 à 4.7 foram discutidas em detalhes os dados colhidos visando responder às questões de pesquisa propostas neste trabalho e na Seção 4.8 foram apresentadas ilustrações dos tipos de testes observados.

4.1 Caracterização dos projetos selecionados

No capítulo anterior foi detalhado como foi feita a seleção dos projetos. Para identificar projetos mais recentes e a popularidade dos projetos foram recolhidas informações como o número de estrelas e a data do último *commit*. Na Tabela 1 são apresentados os 20 projetos.

Tabela 1: Características dos projetos

Projetos	**Estrelas**	***commit***	**Data útimo *commit***	**Testes**
Kbar	4800	182	8 meses	15
Hacker News Clone React/GraphQL	4400	142	2 anos	58
Meli	2400	170	1 ano	90
Cuttlebelle	545	413	1 ano	124
Nextjs-Woocommerce	459	3610	10 horas	6
Front - End (OperationCode)	367	5005	5 meses	228
5Calls	373	948	4 anos	45
Discohook	372	1217	4 meses	110
Retropected	774	966	5 meses	100
E-Commerce site	208	140	7 meses	1
Developer-Portfolio	177	342	1 ano	1
Build-ui	172	240	2 anos	220
OfNotes	126	152	3 anos	100
My Ticket Manager	17	38	4 anos	24
React Testing Application	1	7	2 anos	5
Cards Against Containers	1	153	3 anos	14
rtl-book	0	15	1 ano	9
SaaS de RH	0	18	3 anos	51
Paint-picture	0	311	3 meses	17
Bytebank	0	2	3 meses	15

Fonte: Autoria própria

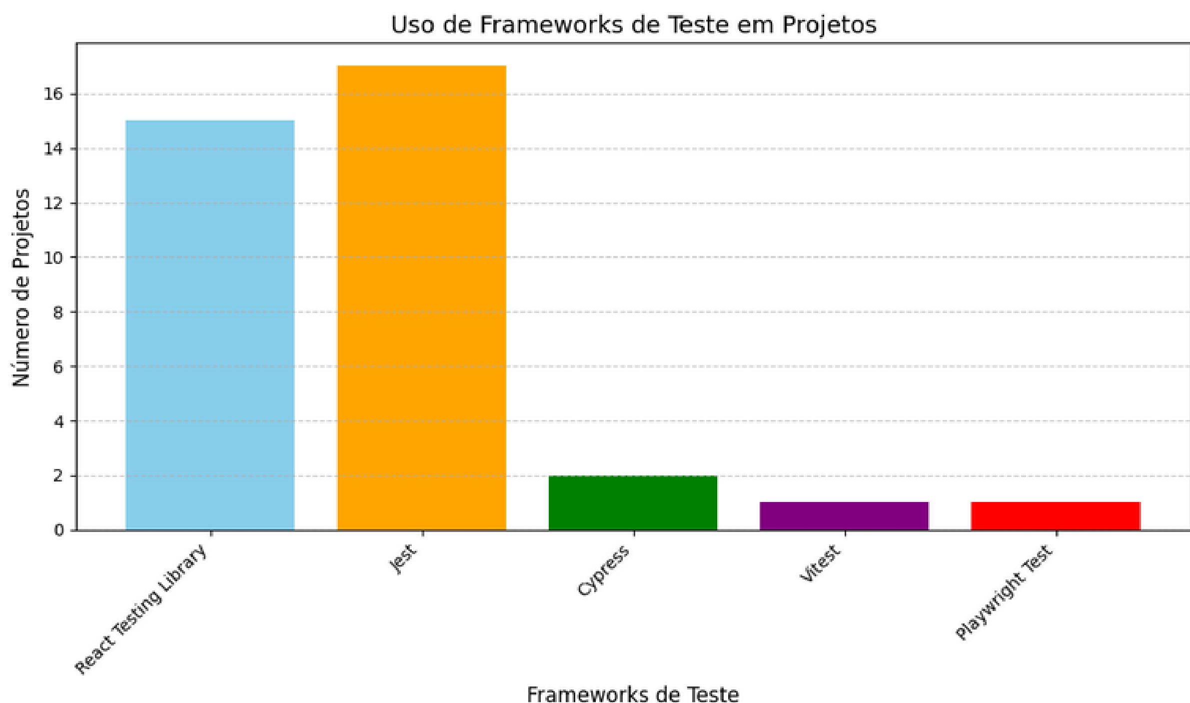
Entre os projetos com mais estrelas temos o Kbar, Hacker News Clone e Meli. O Kbar é um projeto que consiste em um componente React simples *plug-n-play* para adicionar uma interface rápida e portátil ao seu site, O Hacker News Clone é um projeto que tem a intenção de ser um exemplo ou *boilerplate* para ajudar a estruturar projetos usando tecnologias prontas para produção, e Meli é uma plataforma para implementar sites estáticos e aplicações *front end* facilmente. Em relação a seleção de projetos mais recentes

temos que o projeto mais recente tem 10 horas e o mais antigo tem 4 anos desde o último *commit*.

4.2 QP1 - Quais os *frameworks* usados em projetos React.js?

Quando analisados os projetos foi possível observar a frequência com que os diferentes *frameworks* de testes são utilizados. A Figura 9 apresenta um gráfico cujo eixo horizontal (eixo X) representa os 5 *frameworks* de testes observados nos projetos, cada um representado com uma coluna individual distinguidas por cores diferentes onde azul, laranja, verde, lilás e vermelho correspondem a React Testing Library, Jest, Vitest, Cypress e Playwright Test respectivamente. O eixo vertical (eixo Y), representa a quantidade de projetos, com valor mínimo 0 e valor máximo 17. As colunas distinguidas por cores representam quantos projetos usaram ao menos uma vez os *frameworks*.

Figura 9: Frequência dos *frameworks* de teste



Fonte: Autoria própria

Deste gráfico, observamos que os *frameworks* Jest com um total de 17 projeto que repesenta o 85% dos projetos e React Testing Library com um total de 15 projeto que representa o 75% dos projetos são mais utilizados. Isso acontece porque o React Testing Library já vem integrado com o React.js, foi criado especificamente para trabalhar com React.js e Jest é recomendado pela equipe de React.js como o *framework* padrão para testar aplicativos React.js. O React Testing Library incentiva a escrita de testes que

simulam como um usuário final interage com as GUIs. O Jest é mais do que uma biblioteca de testes, inclui: *test runner*, um *mocking framework* permite simular dependências ou funções e um sistema de *snapshot testing*[1] ideal para comparar componentes visuais React.js.

Outra análise é sobre os *frameworks* usados com pouca frequência, isso se deve, porque estes foram projetados para um foco específico. Como no caso do Cypress e Playwright Test são direcionados a testes de sistema. Em relação a Vistest, é semelhante a Jest, mas otimizado para *software* modernos com Vite.

4.3 QP2 - Quais linguagens de programação usadas nos projetos?

A Tabela 2 apresenta as frequências com que foram utilizadas as diferentes linguagens de programação nos projetos.

Tabela 2: Distribuição de projetos por linguagem de programação

Linguagem de Programação	Número de Projetos
JavaScript	20
TypeScript	14
CSS	14
HTML	14
SCSS	3
Python	2
Go	2

Fonte: Autoria própria

Observando a tabela percebemos que a linguagem mais utilizada é JavaScript com um total 20 projetos, seguido de TypeScript, CSS e HTML ambas as três aparecem em 14 projetos, uma possível explicação para isso é que o React.js é uma biblioteca construída especificamente para JavaScript, isso faz com que React.js seja otimizado para trabalhar com JavaScript.

Os projetos também utilizaram TypeScript por que ajuda na robustez do código pois adiciona tipagem estática e recursos avançados ao desenvolvimento de aplicações. O CSS é utilizado nos projetos para garantir GUIs mais atrativas e personalizadas e o HTML para definir a estrutura das GUIs de maneira declarativa.

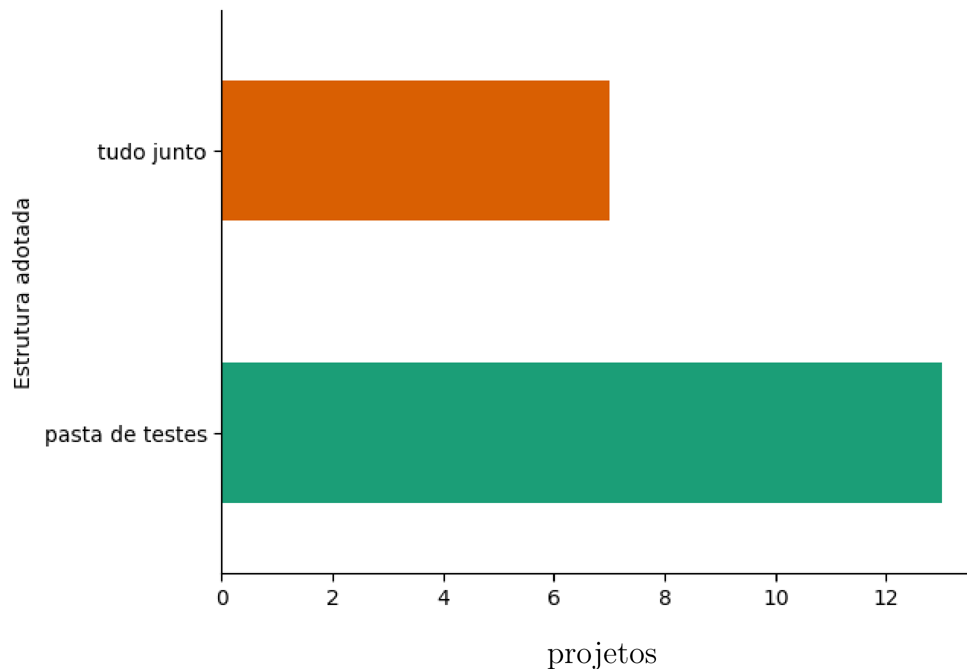
No caso das outras linguagens menos utilizadas uma possível explicação seria que foram utilizadas pelas demandas específicas dependendo de cada projeto. O Python e GO são utilizadas para fornecer dados ao Front End e o SCSS que não é uma linguagem de programação em si, mas sim, uma extensão do CSS utilizada para organizar melhor os estilos visuais.

[1]Snapshot testing é uma técnica de teste automatizado usada para verificar se a interface de um componente ou saída de uma função permanece consistente ao longo do tempo.

4.4 QP3 - Qual estrutura de teste adotada?

Na terceira questão de pesquisa, é apresentado como são estruturados os testes nos projetos. A estrutura dos testes dentro de um projeto é importante, uma boa estrutura de testes facilita muito a manutenção e a escalabilidade do projeto.

Figura 10: Gráfico de estrutura dos testes



projetos

Fonte: Autoria própria

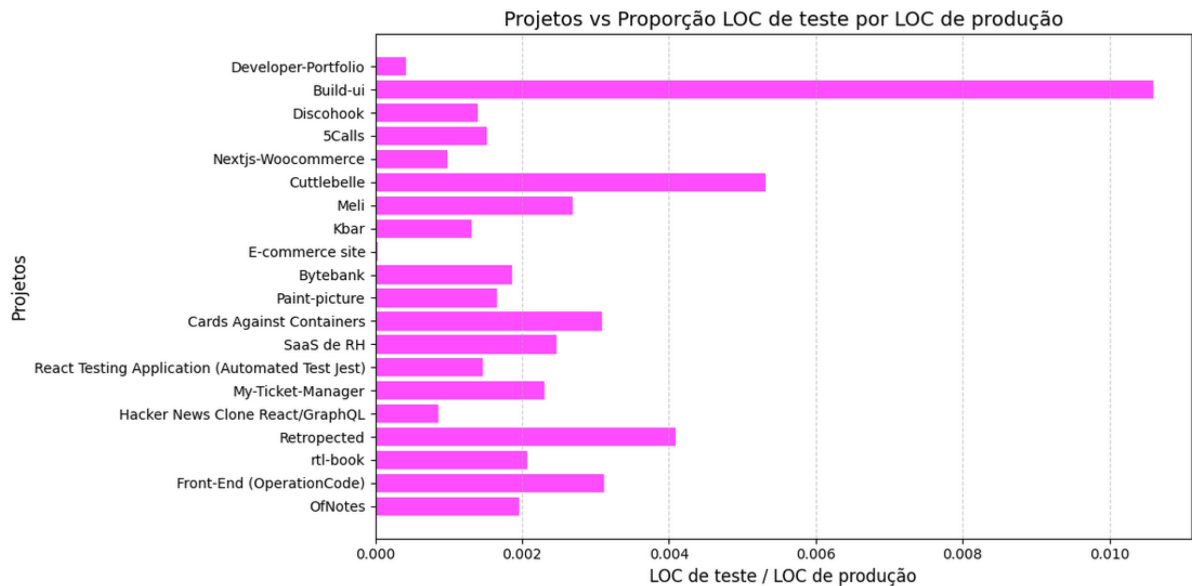
A Figura 10 apresenta um gráfico cujo eixo horizontal (eixo X) representa os projetos. O eixo vertical (eixo Y) representa a estrutura dos testes nos projetos, para isso foram estipuladas duas categorias: *tudo junto*, refere-se a que os arquivos de testes estão misturados com outros arquivos do projetos e *Pasta de teste*, refere-se a que os arquivos de testes estão agrupados em uma única pasta, separados dos outro arquivos do projeto. É possível concluir que a maioria dos projetos adotaram a estrutura *pasta de testes* um dos motivos pode ser porque esta estrutura é a melhor em termos de organização e manutenção, pois facilita a localização e atualização dos testes.

4.5 QP4 - Quantitade de LoC de teste em relação a LoC de produção?

A quantidade de Loc de teste pode revelar informações de aspectos fundamentais sobre a qualidade do software, testes de cobertura e a manutenção do software. A Figura 11 mostra um gráfico cujo eixo horizontal (eixo X), representa a quantidade de Loc de teste

em relação ao Loc de produção e o eixo vertical (eixo Y), representa os projetos.

Figura 11: Gráfico de proporção de Loc de teste em relação ao Loc de produção



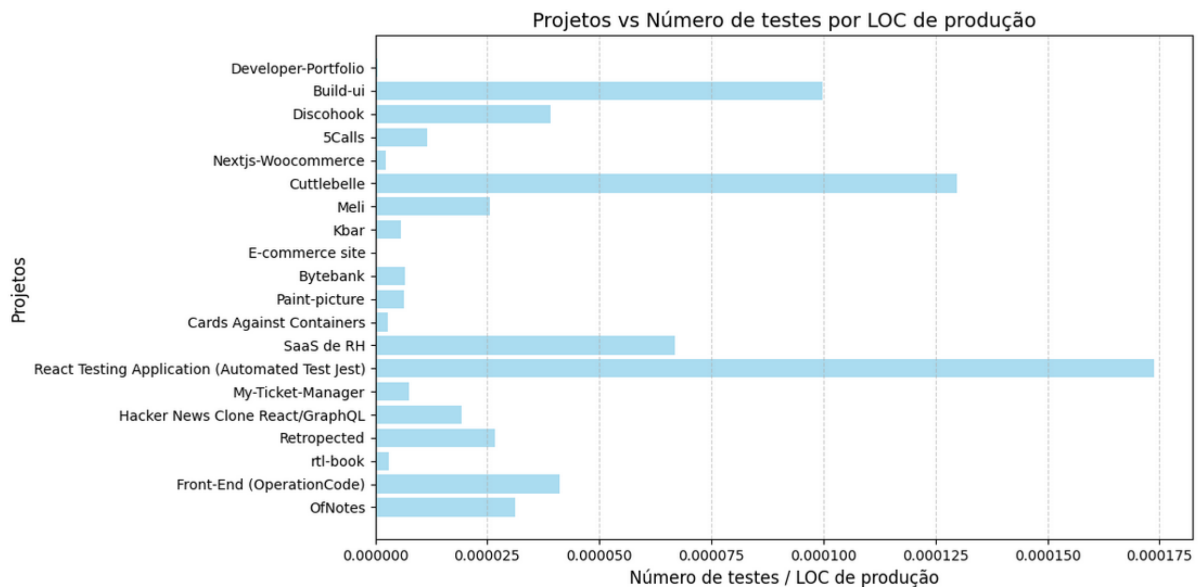
Fonte: Autoria própria

Observando o gráfico podemos constatar que os projetos com maior quantidade de Loc de testes em relação ao Loc de produção são : Build-ui, Cuttlebelle, Retropedted, Cards Aganist Containers e Front-End (Operation Code). O projeto Build-ui se destaca no gráfico por ter uma proporção muito alta de Linhas de Código (LoC) de teste em relação às LoC de produção, indicando um forte investimento em testes automatizados. Por outro lado, os projetos com menos Loc de testes são: React Testing Application, E - commerce site, Developer - portfolio e Next.js -woocommerce.

4.6 QP5 - Quantidade de testes em relação ao Loc de produção?

Embora a quantidade de testes não indique a qualidade dos mesmos nem a cobertura total do projeto, é muito útil saber a quantidade de testes por que um grande número de testes pode indicar que houve maior esforço para cobrir diferentes aspectos da aplicação. A Figura 12 mostra um gráfico cujo eixo horizontal (eixo X), representa a quantidade de testes em relação Loc de código dos projetos e o eixo vertical (eixo Y), representa os projetos.

Figura 12: Gráfico de quantidade de testes em relação a Loc de produção



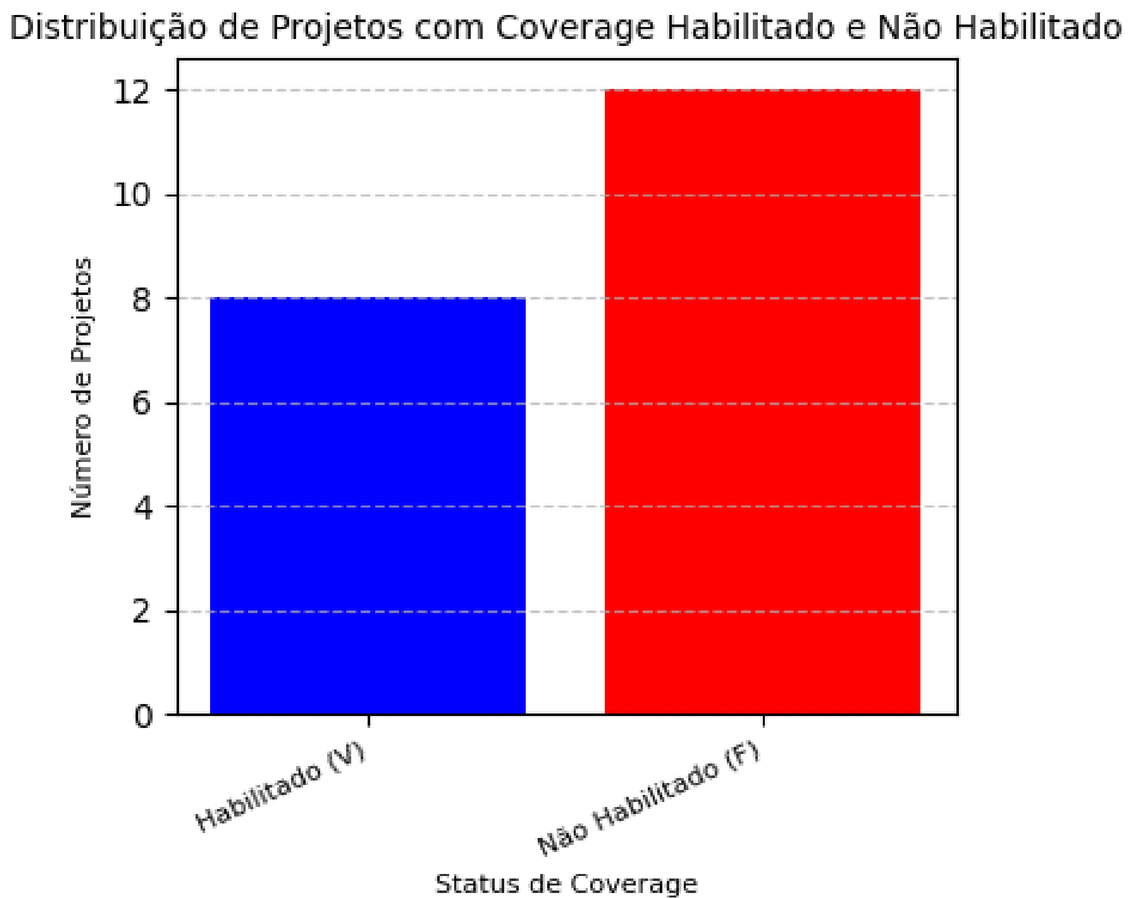
Fonte: Autoria própria

Neste gráfico podemos observar que os projetos com maior quantidade de testes em relação ao Loc de produção são: React Testing Application, Cuttlebelle, Build-ui, SaaS de RH, Front-End (operation Code) e Discohook. Por outro lado, temos os projetos com menos testes E-commerce site, Developer-portifolio, Nextjs-woocommerce, rtl-book e Cards Against Containers.

4.7 QP6 - Quantos projetos habilitam code coverage?

O *Code coverage* ou Relatório de cobertura é uma métrica essencial para avaliar a abrangência dos testes em um aplicativo, indicando quais partes do código estão sendo testadas e quais não. A Figura 13 apresenta um gráfico onde o eixo horizontal (eixo X), representa as categorias: aqueles que habilitam o relatório de cobertura e aqueles que não habilitam. O eixo vertical (eixo Y), representa a quantidade de projetos em cada categoria.

Figura 13: Gráfico de *Code coverage*



Fonte: Autoria própria

O gráfico mostra que 8 projetos que representam 40% dos projetos habilitam o relatório de cobertura, enquanto que 12 que representam 60% dos projetos não o habilitam. Em relação aos projetos que não habilitam relatório de cobertura, isso pode acontecer por diferentes motivos, porém, a falta de relatório cobertura pode acarretar grandes desvantagens.

A Tabela 3 apresenta as porcentagens dos resultados obtidos dos projetos que habilitam o relatório de cobertura. Observando a tabela percebe-se que o projeto que apresenta maior cobertura é o Front-End (Operation code), possui 100% em todas as métricas. Os projetos GinaAdzani_9 e o Build_ui têm uma boa cobertura mas poderia melhorar em *functions* e *branches* respectivamente. O cuttlebelle, Kbar e Meli têm uma cobertura moderada que oscila entre 50% a 80% e O Retropected e Site apresentam uma cobertura abaixo de 50%.

Tabela 3: Cobertura de código dos projetos por categoria (%)

Projetos	% Statements	% Branches	% Functions	% Lines
GinaAdzani_9	83.81	80	68.49	85.4
Cuttlebelle	57.43	59.09	53.21	57.43
Build-ui	82.26	67.66	81.6	85.16
Kbar	68	47.41	61.37	68.8
Retropected	11.64	53.27	20.33	11.64
Site	16.64	18.44	15.86	15.73
Meli	53.57	27.16	17.72	62.42
Front-End (operation code)	100	100	100	100

Fonte: Autoria própria

4.8 Ilustração dos tipos de testes observados.

Teste unitário

Figura 14: Exemplo de teste unitário

```
1  import React from "react";
2  import { render } from "test-utils";
3  import { Layout } from "components";
4
5  describe("Layout component", () => {
6    it("matches snapshot", () => {
7      const { asFragment } = render(<Layout>test</Layout>);
8      expect(asFragment()).toMatchSnapshot();
9    });
10 });
```

Fonte: Projeto OfNotes

A Figura 14 apresenta um teste unitário extraído do projeto OfNotes. Este teste primeiro importa o React (linha 1), importa a função `render()` que vem do arquivo test-utils (linha 2) e importa o componente Layout que será testado (linha 3). Em seguida define um grupo de teste para o componente Layout,`describe("Layout component"` (linha 5). Define um caso de teste `it("matches snapshot"` (linha 6). Renderiza o componente Layout com o texto "test"como filho `const  asFragment  = render(<Layout>test</Layout>)` (linha 7). Usa o expect() do jest para fazer uma assertiva (linha 8).

Este teste utiliza as *frameworks* de teste Jest e React Testing Libary. Também usa React.js para renderizar o componente Layout.

Teste de integração

Figura 15: Exemplo de teste de integração

```
1  import React from 'react';
2
3  import {render, screen, fireEvent} from '@testing-library/react';
4  import '@testing-library/jest-dom';
5
6  import Builder from '../../components/Builder';
7  import DnDBuilder from '../../components/DnDBuilder';
8
9  // All other tests concerning root
10 // DnDSource element and logic related
11 // to handling only transfers with
12 // corresponding transferType are taken
13 // care of in DnDListener test suite.
14
15 describe('<DnDBuilder />', () => {
16
17         test('should pass non-dnd props down to root element', () => {
18         const handleClick = jest.fn();
19         render(<Builder>
20                 <DnDBuilder
21                 onClick = {handleClick}
22                 data-testid = 'builder'
23                 >
24                 <p>Child Prop</p>
25                 </DnDBuilder>
26         </Builder>);
27         const builder = screen.getByTestId('builder');
28         fireEvent.click(builder);
29         expect(builder).toBeInTheDocument();
30         expect(screen.getByText(/child/i)).toBeInTheDocument();
31         expect(handleClick).toHaveBeenCalledTimes(1);
32         });
33
34 });
```

Fonte: Projeto Build-ui

A Figura 15 apresenta um teste de integração para o componente `<DnDBuilder>` usando `Jest` e `Testing Library`. Primeiro, importa as bibliotecas necessárias, incluindo `React` ,`Testing Library` e `Jest-DOM` (linha 1 - 7). Em seguida, cria um bloco `describe` para agrupar testes e define um teste chamado `"should pass non-dnd props down to root element"` (linha 15). Uma função `mock (jest.fn())` é criada para monitorar cliques (linha 18). O componente `<DnDBuilder>` é renderizado com `onClick` e `data-testid="builder"` (linha 19 - 26). O teste busca o elemento renderizado, simula um clique com `fireEvent.click()` (linha 28), verifica se o elemento e o texto do filho estão no DOM e confirma se a função *mock* foi chamada uma vez.

Teste de sistema

Figura 16: Exemplo de teste de sistema

```
1  import { test, expect } from '@playwright/test';
2  test.describe('Produkter', () => {
3    test.beforeEach(async ({ page }) => {
4          await page.goto('http://localhost:3000');
5    });
6    test('Test at vi kan kj pe produktet', async ({ page }) => {
7      await page.getByRole('link', { name: 'Test simple' }).first().click();
8
9          // Expects the URL to contain test-simple
10     await page.waitForURL('http://localhost:3000/produkt/test-simple?id=29', {
11         waitUntil: 'networkidle',
12         });
13     await expect(page).toHaveURL(/.*simple/);
14     await expect(page.getByRole('button', { name: 'KJ P' })).toBeVisible();
15     await page.getByRole('button', { name: 'KJ P' }).click();
16     await page.locator('#header').getByText('1').waitFor();
17     await expect(page.locator('#header').getByText('1')).toBeVisible({
18         timeout: 5000,
19         });
20
21     await page.getByRole('link', { name: 'Handlekurv' }).click();
22     await page.locator('section').filter({ hasText: 'Handlekurv' }).waitFor();
23     // Check that that Handlekurv is visible
24     await expect(
25         page.locator('section').filter({ hasText: 'Handlekurv' }),
26         ).toBeVisible();
27     // Check that we can go to Kasse
28     await page.getByRole('button', { name: 'G   TIL KASSE' }).click();
29     await page.waitForURL('http://localhost:3000/kasse', {
30     waitUntil: 'networkidle',
31         });
32     await expect(
33         page.locator('section').filter({ hasText: 'Kasse' }),
34         ).toBeVisible();
35         // Check that we can type something in Billing fields
36     await page.getByPlaceholder('Etternavn').fill('testetternavn');
37     await page.getByPlaceholder('Etternavn').waitFor();
38     await expect(page.getByPlaceholder('Etternavn')).toHaveValue(
39         'testetternavn',
40         );
41   });
42 });
```

Fonte: projeto Nextjs-Woocommerce.

A Figura 16 apresenta um teste de sistema com Playwright para validar o fluxo de compra de um produto. Primeiro, importa `test` e `expect` (linha 1). Define a suíte `"Produkter"` e configura ações antes de cada teste (linha 2). O teste `"Test at vi kan kjøpe produktet"` clica no link `"Test simple"`, verifica a URL e a visibilidade do botão `"KJØP"`, adiciona o produto ao carrinho e confirma a atualização no cabeçalho (linha 6 - 18). Depois, navega para o carrinho, acessa o `checkout` (linha 28) e verifica a seção `"Kasse"` (linha 32). Por fim, preenche o campo `"Etternavn"` e valida seu valor (linha 38), garantindo a correta execução do fluxo de compra.

5 CONSIDERAÇÕES FINAIS

Este trabalho teve como objetivo investigar como testes automatizados são desenvolvidos em projetos React.js. Os resultados apresentados no Capítulo 4 permitem ter uma percepção de como são implementados os testes em projetos React.js. Analisando os *frameworks* de testes mais utilizados, nota-se que React Testing Library e Jest são os mais utilizados em relação aos outros, uma explicação para isso é que além de serem recomendados pela equipe do React.js ambos combinados oferecem praticidade para escrever e executar testes.

Em relação às linguagens de programação, era previsto de certa forma que a linguagem mais utilizada fosse JavaScript dado que o React.js foi projeto para funcionar no ecossistema JavaScript, outras linguagens que também destacaram foram TypeScript, HTML e CSS ambas as três são importantes porque cada uma delas complementam o React.js para criar aplicações mais robustas, estilizadas e com uma boa estrutura visual.

Com respeito a estrutura dos testes dentro dos projetos, foi observado que a maioria dos projetos seguem uma estrutura de pasta de teste, esta prática é a mais adequada porque ajuda a manter o código mais organizado, eficiente e facilita a manutenção.

Quanto à quantidade de testes e sua relação com as LoC de produção destacam-se 6 projetos: React Testing Application, Cuttlebelle, Build-ui, SaaS de RH, Front-End (operation Code) e Discohook. Estes 6 projetos apresentam uma maior proporção de testes em relação ao LoC de produção, isso indica maior confiança na automação e menor dependência de testes manuais ao mesmo tempo que mostra o nível de detalhe nos testes.

Sobre a quantidade de LoC de teste em relação às Loc de produção destacam 5 projetos com uma maior quantidade de LoC de testes em relação ao LoC de produção: Build-ui, Cuttlebelle, Retropedted, Cards Aganist Containers e Front-End (Operation Code), desses projetos é possível concluir que os testes cobrem grande parte desses projetos, embora não seja uma métrica direta de cobertura dá uma noção do compromisso com a validação de funcionalidades dos projetos.

Sobre os projetos que habilitam relatório de cobertura foi observado que dos 20 projetos selecionados, 8 habilitam e 12 não habilitam. Dos 8 projetos que habilitam, apenas um tem uma cobertura de 100%, o Front-End (operationCode). O GinaAdzani_9 e Build_ui possuem uma cobertura acima de 80%, mas ainda podem melhorar em functions e branches, respectivamente. Já os projetos Cuttlebelle, Kbar e Meli apresentam uma cobertura que varia entre 50% e 80%. Por outro lado, Retropected e Site possuem cobertura inferior a 50%.

Para futuras pesquisas, sugere-se uma análise comparativa entre React Testing Library, Jest e outros *frameworks* de teste como Cypress e Mocha, para entender as vantagens e limitações de cada um, considerando diferentes tipos de aplicações Single-Page Applications (SPAs), Progressive Web Applications (PWAs), entre outras.

Referências

1 MADSEN, M.; LHOTAK, O.; TIP, F. A semantics for the essence of react. In: *European Conference on Object-Oriented Programming*. [S.l.: s.n.], 2020.

2 GACKENHEIMER, C. *Introduction to React*. [S.l.]: Apress, 2015.

3 STACKOVERFLOW. *Stack Overflow Developer Survey 2024: Technology*. 2024. Acessado em: 10 dez. 2024. Disponível em: <https://survey.stackoverflow.co/2024/technology/>.

4 REACT. *React – A JavaScript library for building user interfaces*. 2023. Acesso em: 25 set. 2024. Disponível em: <https://react.dev/>.

5 KINSTA. *Guia de Sintaxe JSX*. 2023. Acessado em 23 de novembro de 2024. Disponível em: <https://kinsta.com/pt/base-de-conhecimento/sintaxe-jsx/>.

6 PRESSMAN, R. S. et al. A practitioner's approach. *Software Engineering*, v. 2, p. 41–42, 2010.

7 DELAMARO, M. et al. *Introdução ao teste de software*. [S.l.]: Elsevier Brasil, 2013.

8 BERNARDO, P. C.; KON, F. A importância dos testes automatizados. *Engenharia de Software Magazine*, v. 1, n. 3, p. 54–57, 2008.

9 React Documentation. *Testando componentes React*. 2024. Acessado em: 05 fev. 2025. Disponível em: <https://pt-br.legacy.reactjs.org/docs/testing.html>.

10 Revelo Community. *Testes Automatizados: Tipos, Ferramentas e Boas Práticas*. 2024. Acessado em: 01 fev. 2025. Disponível em: <https://community.revelo.com.br/testes-automatizados-tipos-ferramentas-e-boas-praticas/>.

11 Jest. *Jest: Framework de Testes JavaScript*. 2024. Acessado em: outubro 2024. Disponível em: <https://jestjs.io/pt-BR/>.

12 HASAN, M. M. et al. Testing react single page web application using automated testing tools. In: *ENASE*. [S.l.: s.n.], 2022. p. 469–476.

13 FERREIRA, F.; VALENTE, M. T. Detecting code smells in react-based web apps. *Information and Software Technology*, v. 155, p. 107111, 2023. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584922002208>.

14 FERREIRA, F.; BORGES, H. S.; VALENTE, M. T. Refactoring react-based web apps. *Journal of Systems and Software*, Elsevier, v. 215, p. 112105, 2024.