UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS SOBRAL

CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

MARCOS VINÍCIUS ANDRADE DE SOUSA

DESENVOLVIMENTO EM REACT: A INFLUÊNCIA DE CLEAN CODE E DESIGN PATTERNS NA VISÃO DOS DESENVOLVEDORES

SOBRAL

2025

MARCOS VINÍCIUS ANDRADE DE SOUSA

DESENVOLVIMENTO EM REACT: A INFLUÊNCIA DE CLEAN CODE E DESIGN PATTERNS NA VISÃO DOS DESENVOLVEDORES

Memorial apresentado à disciplina de Seminário de Monografia do curso de Engenharia de Computação, como requisito parcial à elaboração do Trabalho de Conclusão de Curso.

Orientador: Prof. Dr. Evilasio Costa Junior

Coorientador: Prof. Dr. Fischer Jônatas Ferreira

SOBRAL

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A568d Andrade de Sousa, Marcos Vinícius.
Desenvolvimento em React: A Influência de Clean Code e Design Patterns na Visão dos Desenvolvedores / Marcos Vinícius Andrade de Sousa. – 2025.
87 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Sobral, Curso de Engenharia da Computação, Sobral, 2025.
Orientação: Prof. Dr. Evilasio Costa Junior.
Coorientação: Prof. Dr. Fischer Jônatas Ferreira.

1. React. 2. Padrões de Projeto. 3. Código Limpo. 4. Desenvolvimento Front-End. 5. Survey e Mineração de Repositórios de Software. I. Título.

CDD 621.39

MARCOS VINÍCIUS ANDRADE DE SOUSA

DESENVOLVIMENTO EM REACT: A INFLUÊNCIA DE CLEAN CODE E DESIGN PATTERNS NA VISÃO DOS DESENVOLVEDORES

Memorial apresentado à disciplina de Seminário de Monografia do curso de Engenharia de Computação, como requisito parcial à elaboração do Trabalho de Conclusão de Curso.

Aprovada em: 13 de Março de 2025

BANCA EXAMINADORA

Prof. Dr. Evilasio Costa Junior (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Fischer Jônatas Ferreira (Coorientador)
Universidade Federal de Itajubá (UNIFEI)

PROF. DR. THIAGO IACHILEY ARAÚJO DE SOUZA
Universidade Federal do Ceará (UFC)

PROF. DR. WENDLEY SOUZA DA SILVA
Universidade Federal do Ceará (UFC)

PROF. DR. ISMAYLE DE SOUSA SANTOS
UNIVERSIDADE ESTADUAL DO CEARÁ (UECE)

RESUMO

CONTEXTO: O desenvolvimento de aplicações web modernas enfrenta um momento de pluralidade com o grande número de plataformas, *frameworks* e ferramentas de desenvolvimento web que surgiram nos últimos anos. Com a popularização de bibliotecas como React, que trazem grande flexibilidade na construção de interfaces de usuário dinâmicas, surgem também desafios relacionados à clareza, manutenção e escalabilidade do código. Nesse cenário, os princípios de *clean code* e os *design patterns* desempenham papéis cruciais, promovendo a legibilidade, a padronização de soluções para problemas recorrentes e a sustentabilidade dos projetos de *software web*.

MOTIVAÇÃO: Embora o uso do React seja amplamente difundido, existem poucos estudos na literatura que exploram, especificamente, como os desenvolvedores *front-end* aplicam *design patterns* e *clean code* em aplicações React. Esta lacuna é especialmente relevante diante da crescente demanda por interfaces de alta qualidade, que sejam manuteníveis e escaláveis.

OBJETIVO: Este estudo visa investigar o uso de *design patterns* e *clean code* por desenvolvedores *front-end* que utilizam React, analisando como essas práticas impactam a manutenção, escalabilidade e colaboração em projetos React. Busca-se também medir o nível de conhecimento dos desenvolvedores sobre *design patterns* nas aplicações React, identificar os principais desafios enfrentados na aplicação desses padrões e destacar as práticas mais eficazes para mitigar problemas relacionados à qualidade do código.

METODOLOGIA: Para alcançar o objetivo proposto, a pesquisa utilizou um *survey* direcionado a desenvolvedores que contribuem em projetos React. Simultaneamente, foi realizada uma mineração de dados em repositórios de código no Github, com foco nas *pull requests* e *issues* relacionadas à *design patterns* e *clean code* em projetos que utilizam React de grandes organizações. A combinação desses dados permitiu identificar padrões e práticas comuns, bem como áreas de melhoria.

RESULTADOS: Os resultados deste estudo são valiosos tanto para desenvolvedores iniciantes quanto para profissionais experientes, permitindo que estudantes se beneficiem das práticas recomendadas e que profissionais aprimorem a qualidade e a manutenção de seus projetos com React. Este trabalho contou com os resultados combinados dos dados provenientes da mineração de 63.209 *threads* em repositórios de software e do *survey* que teve a participação de 23 desenvolvedores (dos quais 4 são internacionais). Ademais, este estudo evidenciou a importância de evitar problemas como *long method* e *duplicate code*, recomendando ainda o

uso de TypeScript para impactar positivamente a legibilidade e a manutenibilidade do código, bem como a adoção de práticas como DRY e *design patterns* (por exemplo, o *singleton*) no desenvolvimento com React. Por fim, o *survey* indicou que 26% dos respondentes declararam não utilizar ou desconhecer *design patterns* para *front-end* (equivalendo a 10% dos participantes de uma questão específica).

Palavras-chave: React. *Design Patterns*. Padrões de Projeto. Desenvolvimento Front-End. *Clean Code*. Código Limpo. *Survey*. Mineração de Repositórios.

ABSTRACT

CONTEXT: The development of modern web applications is experiencing a period of plurality due to the large number of platforms, *frameworks*, and web development tools that have emerged in recent years. With the popularization of libraries such as React, which provide great flexibility in building dynamic user interfaces, challenges related to code clarity, maintenance, and scalability also arise. In this context, the principles of *clean code* and *design patterns* play crucial roles, promoting readability, the standardization of solutions for recurring problems, and the sustainability of web software projects.

MOTIVATION: Although the use of React is widespread, there are few studies in the literature that specifically explore how *front-end* developers apply *design patterns* and *clean code* in React applications. This gap is particularly significant given the growing demand for high-quality interfaces that are maintainable and scalable.

OBJECTIVE: This study aims to investigate the use of *design patterns* and *clean code* by *front-end* developers using React, analyzing how these practices impact maintenance, scalability, and collaboration in React projects. It also seeks to measure developers' level of knowledge about *design patterns* in React applications, identify the main challenges faced in applying these patterns, and highlight the most effective practices to mitigate issues related to code quality.

METHODOLOGY: To achieve the proposed objective, the research utilized a *survey* directed at developers contributing to React projects. Simultaneously, data mining was conducted on code repositories on GitHub, focusing on *pull requests* and *issues* related to *design patterns* and *clean code* in projects using React from large organizations. The combination of these data allowed the identification of common patterns and practices, as well as areas for improvement.

RESULTS: The results of this study are valuable for both novice developers and experienced professionals, allowing students to benefit from recommended practices and professionals to enhance the quality and maintainability of their React projects. This work relied on the combined outcomes from data mined from 63,209 *threads* in software repositories and from the *survey* that included the participation of 23 developers (of which 4 are international). Moreover, this study highlighted the importance of avoiding issues such as *long method* and *duplicate code*, further recommending the use of TypeScript to positively impact code readability and maintainability, as well as the adoption of practices such as DRY and *design patterns* (for example, the *singleton*) in React development. Finally, the *survey* indicated that 26% of respondents reported not using

or being unaware of *design patterns* for the *front-end* (equivalent to 10% of the participants in a specific question).

Keywords: Clean Code. React. Design Patterns. Front-End Development. Survey. Mining Software Repositories.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
API REST	*Representational State Transfer Application Programming Interface*
CLI	*Command Line Interfaces*
CSS	*Cascading Style Sheets*
DIP	*Dependency Inversion Principle*
DOM	*Document Object Model*
DRY	*Don't Repeat Yourself*
HOC	*Higher-Order Components*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
ISP	*Interface Segregation Principle*
JSON	*JavaScript Object Notation*
JSX	Extensão de sintaxe para JavaScript
KISS	*Keep it Simple, Stupid*
LSP	*Liskov Substitution Principle*
OCP	*Open-Closed Principle*
POO	Programação Orientada a Objetos
QP	Questões de Pesquisa
SRP	*Single Responsiblity Principle*
VirtualDOM	*Virtual Document Object Model*
XP	*Extreme Programming*
XSS	*Cross Site Scripting*
YAGNI	*You Aren't Gonna Need It*

SUMÁRIO

1 INTRODUÇÃO

A crescente complexidade das aplicações web modernas e a demanda por interfaces de usuário dinâmicas levaram os desenvolvedores a adotar práticas e ferramentas que pudessem aprimorar a qualidade e a manutenção do código. Nesse contexto, a prática de *clean code* destacou-se como uma abordagem essencial para garantir a clareza e a legibilidade do código (Robert C. Martin, 2009).

O *clean code* transformou a maneira como os projetos são estruturados, tornando o software não apenas funcional, mas também compreensível e facilmente mantido. A aplicação de *clean code* mostrou-se crucial em projetos complexos, onde a colaboração e a evolução contínua se apresentavam como requisitos fundamentais (Robert C. Martin, 2009).

O conceito de *clean code*, defendido por Robert C. Martin (2009), enfatizou a necessidade de produzir código claro e simples. Esse princípio foi considerado indispensável para que o código pudesse ser compreendido e modificado tanto por outros desenvolvedores quanto pelo próprio autor em momentos futuros (LUBURIĆ *et al.*, 2022). Em ambientes colaborativos, a clareza e a manutenção do código constituíram aspectos essenciais para a eficiência dos projetos. Assim, as boas práticas de *clean code* tornam-se aspectos fundamentais para a eficiência e a eficácia do projeto (LJUNG; GONZALEZ-HUERTA, 2022).

No contexto do desenvolvimento com React, uma biblioteca JavaScript amplamente adotada para a criação de interfaces de usuário, a aplicação de *clean code* adquiriu importância ainda maior (PADOLSEY, 2020). O React oferece um alto grau de liberdade e criatividade, permitindo a criação de componentes reutilizáveis e modulares (REACT, 2025).

A flexibilidade do React, que se revelou uma das principais vantagens da ferramenta, também apresentou desafios quando as práticas recomendadas não eram seguidas – como componentes muito grandes, complexos e com múltiplas responsabilidades – não eram seguidas (LAZUARDY; ANGGRAINI, 2022). Dessa forma, a implementação de *clean code* em projetos com React mostrou-se fundamental para a manutenção, evolução e eficiência do código (GOMES *et al.*, 2023).

Outro aspecto relevante consistiu na utilização de *design patterns*, que representam soluções reutilizáveis e bem documentadas para problemas recorrentes no desenvolvimento de software (GAMMA *et al.*, 1994). Esses padrões fornecem uma linguagem comum entre os desenvolvedores e auxiliam na criação de sistemas robustos, flexíveis e de fácil manutenção (GAMMA *et al.*, 1994). Os *design patterns* consolidaram-se como estratégias eficientes para a

padronização e o aprimoramento dos processos de desenvolvimento (GOMES *et al.*, 2023).

No ambiente do desenvolvimento moderno, React beneficiou-se da aplicação de *design patterns* para estruturar aplicações ricas em interação e modularidade (OSMANI, 2023). Tais padrões possibilitaram a separação clara entre a lógica de apresentação, controle e dados, contribuindo para um desenvolvimento mais eficiente e uma manutenção mais previsível (OSMANI, 2023). Ao adotar esses conceitos, a criação de interfaces dinâmicas e reutilizáveis pode ser facilitada, explorando ao máximo as capacidades do cliente sem comprometer a comunicação com o servidor (JARTARGHAR *et al.*, 2022).

Apesar da valiosa capacidade de React em promover a modularidade e a reutilização de componentes, essa característica exige uma abordagem cuidadosa para a organização e clareza do código (LAZUARDY; ANGGRAINI, 2022). Caso o gerenciamento não seja adequado, o código pode tornar-se difícil de compreender e manter conforme o projeto cresce. Dessa forma, este trabalho investiga a adoção de práticas de *clean code* e *design patterns* por desenvolvedores *front-end*.

Foi utilizado um *survey* para identificar padrões de projeto, dificuldades e benefícios associados à adoção de boas práticas de *clean code* entre desenvolvedores *front-end* que utilizam React. Paralelamente, foi feita uma mineração de repositórios do GitHub, com ênfase na análise de discussões e trechos de código presentes em *issues* e *pull requests* que abordavam temas relacionados à qualidade de código.

1.1 Objetivos Gerais do Trabalho

O objetivo deste trabalho consiste em compreender como a aplicação das boas práticas de código impactava a qualidade do software e a eficiência do desenvolvimento *front-end*. Com base nos dados coletados, espera-se que os resultados beneficiem tanto pesquisadores quanto desenvolvedores. Assim, contribuem para a melhoria contínua na qualidade do desenvolvimento de *software*.

1.1.1 Objetivos Específicos

- Identificar o nível de conhecimento dos desenvolvedores quanto a *design patterns* e boas práticas.
- Elaborar um protocolo geral para planejar as estratégias de coleta de dados.

- Elaborar um *survey* a ser respondido por desenvolvedores que usam React.
- Coletar informações referentes a *pull requests* e *issues*, com o intuito de identificar discussões que contribuam para o aprimoramento do trabalho, por meio da mineração de repositórios do GitHub voltados para projetos em React.
- Analisar e discutir os resultados fundamentados nas respostas obtidas por meio das questões de pesquisa definidas no protocolo.

1.2 Estrutura do Trabalho

Este trabalho está organizado em capítulos que contemplam os principais aspectos teóricos e metodológicos da pesquisa. No Capítulo 2, apresenta-se a fundamentação teórica que sustenta o estudo, oferecendo uma base conceitual robusta para a análise dos dados. A seguir, o Capítulo 3 descreve detalhadamente a metodologia adotada.

No Capítulo 4, os resultados obtidos são expostos e analisados, permitindo uma compreensão aprofundada das práticas e desafios identificados. As ameaças à validade do estudo são discutidas no Capítulo 5, enquanto o Capítulo 6 aborda os principais estudos e pesquisas relacionados ao tema. Por fim, as conclusões e sugestões para trabalhos futuros são apresentadas no Capítulo 7, encerrando o trabalho com uma síntese dos achados.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção tem como objetivo apresentar princípios de qualidade de *software*, e o que é desenvolvimento *front-end*, assim como apresentar o React. Inicialmente foi definido o que é a qualidade de código (Seção 2.1), e alguns princípios que estão englobados, como: *clean code*, SOLID e *design patterns*.

Posteriormente, foi iniciado o contexto do desenvolvimento *front-end* (Seção 2.2), o React e seus principais conceitos, como: Ferramentas *Command Line Interfaces* (CLI) para Iniciar com React, Componentes em React, *Hooks*, renderização, rotas, gerenciamento de estado, estilização das páginas, bibliotecas de componentes e integração com *Application Programming Interface* (API).

2.1 Qualidade de Código

A qualidade do código é essencial para o sucesso e a longevidade dos projetos de *software*, principalmente em ambientes *front-end* (PADOLSEY, 2020). Um código bem escrito facilita a manutenção, adaptação e compreensão por diversos desenvolvedores ao longo do tempo (PADOLSEY, 2020). Assim, a clareza e a simplicidade tornam o *software* não apenas funcional, mas também legível, promovendo um trabalho colaborativo eficaz (Robert C. Martin, 2009).

A organização adequada do código está intrinsecamente ligada à sua complexidade, o que afeta diretamente a facilidade de manutenção (Robert C. Martin, 2009). Um código estruturado de forma explicável geralmente apresenta menor complexidade, permitindo identificar de maneira mais simples os caminhos explicáveis e, consequentemente, reduzir a incidência de erros (WIJENDRA; HEWAGAMAGE, 2021).

A aplicação de *design patterns* constitui outra prática que eleva a qualidade do código (LAKSRI *et al.*, 2022). Tais padrões oferecem soluções consagradas para desafios recorrentes, possibilitando a criação de sistemas coesos e eficientes (LAKSRI *et al.*, 2022). No desenvolvimento *front-end* com React, por exemplo, a utilização de componentes reutilizáveis promove a modularidade e a manutenção simplificada (REACT, 2025).

Por fim, a qualidade do código fortalece a colaboração entre os desenvolvedores (FARIAS, 2022). Um código claro e bem estruturado funciona como meio de comunicação não só com a máquina, mas também entre os profissionais que o mantêm (Robert C. Martin, 2009). Essa clareza inicial é vital para evitar complicações futuras e assegurar que o sistema permaneça

compreensível e facilmente modificável (FEATHERS, 2004).

2.1.1 Clean code e Boas Práticas de Programação

O *clean code* é uma abordagem que vai além de simplesmente fazer o código funcionar, enfatizando a importância de torná-lo compreensível, sustentável e facilmente expansível (SILVA *et al.*, 2022). Assim como na organização do código discutida anteriormente, a clareza e a simplicidade são elementos essenciais para garantir a longevidade e a eficácia dos sistemas (Robert C. Martin, 2009).

Analogamente a um autor que redige um livro pensando em seus leitores, o desenvolvedor deve escrever código para os colegas que irão ler, avaliar e dar continuidade ao trabalho (Robert C. Martin, 2009). Ao priorizar uma estrutura organizada e legível, o ambiente colaborativo é aprimorado, reduzindo ambiguidades e facilitando a identificação de erros (SILVA *et al.*, 2022). Essa prática se alinha com os conceitos de manutenção e escalabilidade, reforçando a ideia de que um código bem estruturado é fundamental para a evolução contínua do *software*.

Entre as boas práticas destacam-se a escolha criteriosa de nomes para variáveis e funções, que devem refletir claramente seu propósito (Robert C. Martin, 2009). Evitar redundâncias e comentários desnecessários, bem como adotar uma formatação consistente, são medidas essenciais para manter o código acessível e de alta qualidade (Robert C. Martin, 2009). Dessa forma, a aplicação desses princípios contribui diretamente para a sustentabilidade e eficiência do desenvolvimento (Robert C. Martin, 2009).

2.1.2 Princípios SOLID

Os princípios S.O.L.I.D. oferecem um conjunto de diretrizes essenciais para a organização de funções e estruturas de código, especialmente no contexto da Programação Orientada a Objetos (POO), mas que podem ser adaptados a outros paradigmas. O objetivo central desses princípios é facilitar o desenvolvimento de sistemas que sejam flexíveis a mudanças, com um código mais modular, legível e de fácil manutenção, permitindo a reutilização e evolução contínua (Robert C. Martin, 2019).

Cada princípio que compõe a sigla S.O.L.I.D. contribui para a criação de arquiteturas de *software* robustas e bem estruturadas, como ilustrado na Tabela 1 (Robert C. Martin, 2019). Esses princípios são fundamentais para garantir que o sistema não apenas atenda aos requisitos imediatos, mas também seja uma base sólida para futuros desenvolvimentos e expansões (Robert

C. Martin, 2019).

Tabela 1 – Princípios SOLID, suas definições e benefícios

Princípio SOLID	**Definição**	**Benefício**
Princípio da Responsabilidade Única (*Single Responsiblity Principle* - SRP)	Uma classe deve ter uma única responsabilidade ou motivo para mudar.	Facilita a manutenção e a compreensão do código ao isolar responsabilidades.
Princípio Aberto/Fechado (*Open-Closed Principle* - OCP)	*Software* deve estar aberto para extensão, mas fechado para modificação.	Permite adicionar novas funcionalidades sem alterar o código existente, minimizando a introdução de erros.
Princípio da Substituição de Liskov (*Liskov Substitution Principle* - LSP)	Objetos de uma classe base devem poder ser substituídos por objetos de uma classe derivada sem alterar o comportamento do programa.	Garante que subclasses possam ser usadas de forma intercambiável com suas superclasses, preservando a integridade do sistema.
Princípio da Segregação de Interfaces (*Interface Segregation Principle* - ISP)	Clientes não devem ser forçados a depender de interfaces que não utilizam.	Promove a criação de interfaces específicas e pequenas, facilitando a implementação e a manutenção.
Princípio da Inversão de Dependência (*Dependency Inversion Principle* - DIP)	Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.	Reduz o acoplamento entre módulos, aumentando a flexibilidade e a facilidade de manutenção do sistema.

Fonte: (MARTIN, 2000)

2.1.3 Outros Princípios

You Aren't Gonna Need It (YAGNI), acrônimo para "Você não vai precisar disso", é um princípio da metodologia ágil *Extreme Programming* (XP) que enfatiza a implementação apenas dos recursos necessários no momento presente (HUNT; THOMAS, 2003). O objetivo central do YAGNI é evitar a complicação excessiva e o desperdício de tempo ao desenvolver funcionalidades que podem nunca ser utilizadas. Ao focar nos requisitos essenciais e desencorajar a adição de funcionalidades com base em previsões futuras, o YAGNI promove a simplicidade e a eficiência no desenvolvimento de *software* (HUNT; THOMAS, 2003).

Keep it Simple, Stupid (KISS), é um princípio que defende a escolha da solução mais simples quando há múltiplas opções disponíveis (CRAMER, 2024). A simplicidade é o foco central desse princípio, pois um *design* ou implementação simples evita a complexidade desnecessária, facilitando a manutenção, compreensão e modificação do código. Ao adotar o KISS, os desenvolvedores são encorajados a evitar soluções complicadas que podem introduzir problemas e dificultar o desenvolvimento (CRAMER, 2024).

Don't Repeat Yourself (DRY), é um princípio fundamental na programação que visa eliminar a repetição de código e dados (WILSON *et al.*, 2014). Cada elemento do sistema deve ter uma única representação oficial, o que reduz o risco de inconsistências e erros. Além

disso, o DRY promove a modularização do código, incentivando a reutilização de trechos já implementados, em vez de copiar e colar. Ao seguir esse princípio, o código se torna mais fácil de entender, manter e reutilizar, contribuindo para um desenvolvimento mais eficiente e menos propenso a falhas (WILSON *et al.*, 2014).

2.1.4 Design Patterns

Os *design patterns* são soluções comprovadas e reutilizáveis para problemas recorrentes no *design* de *software* (BERTOLI, 2017). Eles fornecem uma abordagem estruturada para resolver desafios comuns, promovendo a reutilização de soluções e facilitando a comunicação entre desenvolvedores (BERTOLI, 2017). A implementação de *design patterns* pode melhorar a qualidade do *software*, facilitar sua manutenção e promover a escalabilidade e a modularidade (BERTOLI, 2017).

A ideia de que os padrões de projeto foram desenvolvidos especificamente para programação orientada a objetos ficou famosa com o livro de Gamma *et al.* (1994). Embora os padrões de projeto tenham ganhado notoriedade nesse contexto, seus conceitos são aplicáveis em diversas outros paradigmas de programação, inclusive no *front-end* (SILVA *et al.*, 2024).

Os principais *design patterns* são classificados em três categorias (REFACTORING.GURU, 2025):

- **Padrões de Criação**: Envolvem o processo de criação de objetos, ajudando a abstrair a instânciação e a fornecer um controle mais refinado sobre a criação dos mesmos. Exemplos incluem o *Singleton* e o *Factory Method*.
- **Padrões Estruturais**: Abordam a composição de classes e objetos para formar estruturas maiores, facilitando a organização e a gestão das relações entre eles. Exemplos são o *Adapter* e o *Composite*.
- **Padrões Comportamentais**: Focam na interação e comunicação entre objetos, definindo como eles colaboram e distribuem responsabilidades. Exemplos incluem o *Observer* e o *Strategy*.

A Tabela 2 resume os *design patterns* citados como exemplos, por meio de uma breve descrição:

Tabela 2 – Resumo dos principais Design Patterns

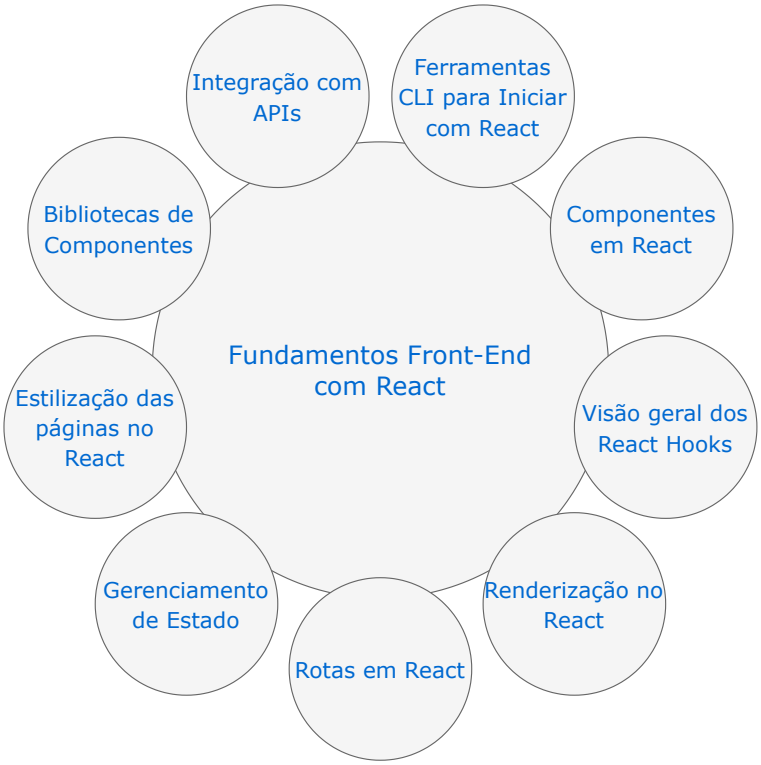
Padrão	**Categoria**	**Descrição**
Singleton	Criação	Garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global a ela.
Factory Method	Criação	Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar.
Adapter	Estrutural	Permite que uma interface incompatível seja compatível com a interface esperada.
Composite	Estrutural	Compondo objetos em estruturas de árvore para representar hierarquias parte-todo.
Observer	Comportamental	Define uma dependência um-para-muitos entre objetos, de forma que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.
Strategy	Comportamental	Define uma família de algoritmos, encapsula cada um e torna-os intercambiáveis.

Fonte: (GAMMA *et al.*, 1994)

2.2 Front-End Web com React

A Figura 1 ilustra, de forma sugerida, os elementos que compõem as etapas da fundamentação teórica deste trabalho, cujo enfoque é o uso do React no desenvolvimento de interfaces no *front-end*. A representação gráfica consolida os principais tópicos e conceitos abordados, fundamentando-se na documentação oficial do React (2025), e visa oferecer uma visão abrangente do tema.

Figura 1 – Estrutura conceitual sugerida para a fundamentação teórica de React



Fonte: Elaborado pelo autor

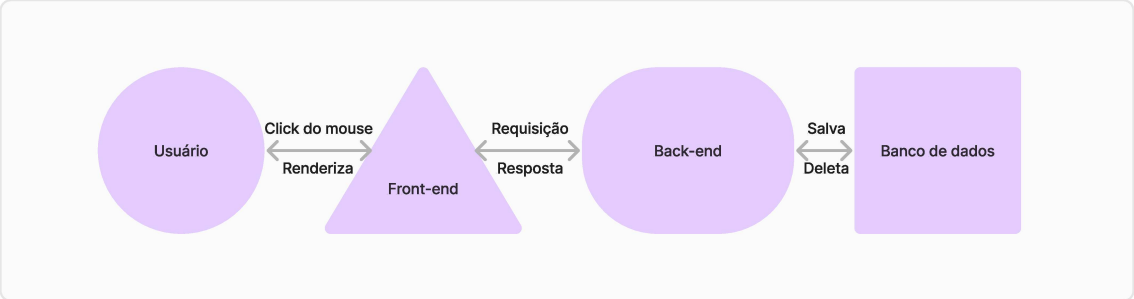
2.2.1 Fundamentos do Front-End

Aplicações web e móveis são comumente divididas em duas áreas principais: *front-end* e *back-end* (LAZUARDY; ANGGRAINI, 2022). No contexto das aplicações web, que são acessadas por navegadores, o desenvolvimento front-end se concentra na criação e na gestão da interface gráfica e da interação do usuário. Os sites são estruturados principalmente por três tecnologias: *HyperText Markup Language* (HTML), *Cascading Style Sheets* (CSS) e JavaScript (MDN, 2024a).

Os navegadores modernos utilizam motores de renderização, conhecidos como *rendering engines*, para exibir conteúdos baseados em HTML e CSS (MDN, 2024b). Além disso, esses navegadores possuem motores de execução de JavaScript, como o *V8 JavaScript Engine* do Google Chrome [1]. Estes motores interpretam JavaScript segundo os padrões ECMAScript e outras especificações tecnológicas, permitindo a execução dinâmica de scripts (NODE, 2024). Com essas tecnologias, é possível construir sites estáticos onde o conteúdo permanece constante (W3SCHOOLS, 2024).

O desenvolvimento *front-end* envolve a criação de interfaces gráficas e a gestão de elementos interativos, como botões e a adaptação da página para diferentes tamanhos de tela (MADURAPPERUMA *et al.*, 2022). Para fornecer conteúdo dinâmico, como atualizações em tempo real, o *front-end* deve se comunicar com o *back-end*. Essa interação é essencial para solicitar e atualizar informações automaticamente (MADURAPPERUMA *et al.*, 2022). O *back-end*, por sua vez, interage com bancos de dados para gerenciar e armazenar dados, como ilustrado na Figura 2.

Figura 2 – Visualização simplificada da comunicação web



Fonte: Elaborado pelo autor

[1] Google Chrome disponível em: <https://www.google.com/intl/pt-BR/chrome/>

O HTML constitui a espinha dorsal de qualquer página na internet, sendo responsável por estruturar o conteúdo de maneira lógica e semântica (MOZILLA, 2025c). Mediante suas *tags*, é possível organizar informações em títulos, parágrafos, listas e links, criando uma hierarquia que facilita a interpretação dos dados pelos navegadores e mecanismos de busca (W3SCHOOLS, 2025b).

Esse conjunto de elementos que compõe o HTML promove a clareza e a acessibilidade do conteúdo, permitindo que tanto o usuário quanto as ferramentas de indexação compreendam o HTML de uma página *web* (MOZILLA, 2025c). Dessa forma, dominar o HTML é indispensável para qualquer desenvolvedor que deseje criar aplicações *web* robustas e bem estruturadas (MOZILLA, 2025c).

O CSS complementa o HTML ao definir a identidade visual das páginas, controlando aspectos como cores, fontes, espaçamentos e posicionamento dos elementos (MOZILLA, 2025a). Essa linguagem de estilos permite a criação de layouts responsivos e adaptáveis a diferentes tamanhos de tela, proporcionando uma experiência agradável em dispositivos variados (MOZILLA, 2025a). Por meio de regras e seletores, o CSS facilita a manutenção do código, permitindo atualizações e ajustes ao *design* da página (W3SCHOOLS, 2025a). A modularidade oferecida por essa ferramenta também possibilita a reutilização de estilos em diversos projetos (W3SCHOOLS, 2025a). Assim, o domínio do CSS é essencial para transformar uma estrutura básica em uma interface atrativa e funcional (MOZILLA, 2025a).

O JavaScript agrega dinamismo e interatividade às páginas, sendo a ferramenta que transforma conteúdos estáticos em experiências interativas (W3SCHOOLS, 2025d). Com ele, desenvolvedores podem manipular os elementos definidos pelo HTML e estilizados pelo CSS, respondendo às ações dos usuários, como cliques e inserção de dados (MOZILLA, 2025f). Essa linguagem possibilita a criação de funcionalidades, como a validação de formulários, animações e carregamento de informações de forma assíncrona, contribuindo para uma experiência de uso mais dinâmica (W3SCHOOLS, 2025d). Dessa forma, o JavaScript demonstra ser um componente indispensável para o desenvolvimento de interfaces *web* (W3SCHOOLS, 2025d).

2.2.2 React

O React é uma biblioteca JavaScript desenvolvida pelo Facebook, que se posiciona como o V do padrão de projeto *Model-View-Controller* (ROBBESTAD, 2016). Um padrão de arquitetura de *software* comumente usado, onde o V significa *View*, e representa o layout da

aplicação (MOZILLA, 2025g). O React é amplamente utilizada para a renderização de elementos da interface de usuário, tais como botões, barras de pesquisa, cabeçalhos, campos de texto e rodapés (REACT, 2025). Dessa forma, onde cada elemento citado é um componente de código modular, o React permite a reutilização desses elementos tanto em diferentes páginas quanto na mesma página (REACT, 2025).

Inicialmente, o React teve suas raízes no XHP [2], uma extensão do PHP [3] que suporta a linguagem de marcação XML. Em 2010, essa extensão permitia que o PHP compreendesse XML, possibilitando a reutilização de componentes de forma similar à adotada pelo React. Embora o XHP tenha sido concebido para reduzir ataques de *Cross Site Scripting* (XSS), o Facebook enfrentou dificuldades para gerenciar aplicações web dinâmicas que exigiam muitas requisições do usuário (KOPPALA, 2021). Essa limitação evidenciou a necessidade de uma novas soluções como o BoltJS e o FaxJS, *frameworks* que evoluíram para o React (KOPPALA, 2021).

2.2.3 Ferramentas CLI para Iniciar com React

As CLI são ferramentas essenciais que possibilitam a execução de comandos diretamente no terminal (AMAZON, 2025). Elas permitem a automatização de processos e a configuração rápida de ambientes para o desenvolvimento de aplicações. Essa abordagem agiliza o fluxo de trabalho, dispensando o uso de interfaces gráficas complexas. Com as CLI, é possível criar, configurar e gerenciar projetos de forma padronizada e eficiente.

Antes de iniciar um projeto, é fundamental ter instalado um gerenciador de pacotes, como o *npm*[4], que é responsável por baixar e organizar as dependências necessárias (MOZILLA, 2025h). Esse *package manager* assegura que as bibliotecas utilizadas estejam nas versões compatíveis, evitando conflitos no ambiente de desenvolvimento (MOZILLA, 2025h). A verificação e a instalação prévia do *npm* garantem um ambiente estável para a criação do projeto (NPMJS, 2025).

Para iniciar um projeto com *Create React App*[5], execute o comando que cria a estrutura inicial do aplicativo (META, 2025d). Após a execução, acesse a pasta do projeto utilizando o comando *cd nome-do-projeto*. Em seguida, instale as dependências necessárias com

[2] XHP disponível em: <https://github.com/phplang/xhp>
[3] PHP disponível em: <https://www.php.net/>
[4] NPM disponível em: <https://docs.npmjs.com/downloading-and-installing-node-js-and-npm>
[5] *Create React App* disponível em: <https://create-react-app.dev/docs/getting-started/>

o comando *npm run install*. Por fim, inicie o servidor de desenvolvimento com o comando *npm run dev*. Veja o passo a passo completo no Listing 2.2.1.

```
1 npx create-react-app nome-do-projeto
2 cd nome-do-projeto
3 npm run install
4 npm run dev
```

Listing 2.2.1 – Passos para iniciar um projeto com Create React App

No caso do *Vite*[6], o procedimento é semelhante e oferece uma configuração moderna e eficiente para projetos em React (VITE, 2023). Para iniciar o projeto, execute o comando da Linha 1 da Listing 2.2.2, que utiliza o *template* React para configurar o ambiente. Em seguida, acesse a pasta do projeto com o comando da Linha 2 (*cd nome-do-projeto*) e, na Linha 3, instale as dependências necessárias usando o comando *npm run install*. Por fim, execute o comando da Linha 4 (*npm run dev*) para iniciar o servidor de desenvolvimento e começar a trabalhar no projeto.

```
1 npm create vite@latest nome-do-projeto -- --template react
2 cd nome-do-projeto
3 npm run install
4 npm run dev
```

Listing 2.2.2 – Passos para iniciar um projeto com Vite

2.2.4 Componentes em React

Os componentes em React constituem a base para a construção de interfaces modulares e reutilizáveis (W3SCHOOLS, 2025e). Essa abordagem possibilita a divisão da interface em partes menores, facilitando o desenvolvimento e a manutenção (META, 2025j). Cada componente é responsável por uma funcionalidade específica, promovendo a organização do código e a escalabilidade da aplicação (META, 2025j). A modularidade favorece também o reuso de código, contribuindo para a padronização do projeto. Dessa forma, a estrutura de componentes se mostra fundamental para o desenvolvimento de interfaces modernas (W3SCHOOLS, 2025e).

Os componentes funcionais em React são funções JavaScript que retornam elementos de interface (META, 2025j). Essa abordagem apresenta uma sintaxe mais simples e concisa se

[6] Vite disponível em: <https://vite.dev/guide/>

comparada aos componentes baseados em classes (ROBBESTAD, 2016). Por serem funções, elas facilitam o entendimento e a manutenção do código, promovendo uma escrita mais direta (META, 2025j). Além disso, a utilização de *hooks* permite gerenciar estado e efeitos colaterais de maneira eficaz (W3SCHOOLS, 2023).

Para fins de ilustração, observe o Listing 2.2.3, que apresenta a implementação de um componente funcional denominado *Saudacao*. Inicialmente, na linha 1, é realizada a declaração do componente, cujo propósito principal é exibir a mensagem "Olá, mundo!" na interface do usuário. A seguir, a partir da linha 2, o componente procede com o retorno de um fragmento de HTML. Essa parte do código é responsável por estruturar e organizar os elementos visuais que serão renderizados na tela. Finalmente, na linha 9, observa-se a exportação do componente *Saudacao*, e isso permite que o componente seja utilizado em outras partes do projeto.

```
1 function Saudacao() {
2   return (
3     <div>
4       <h1>Olá, mundo!</h1>
5     </div>
6   );
7 }
8
9 export default Saudacao;
```

Listing 2.2.3 – Exemplo de um componente funcional em React

2.2.5 Extensão de sintaxe para JavaScript, Importações e Exportações de Componentes

A Extensão de sintaxe para JavaScript (JSX) permite a mistura de HTML com JavaScript para definir a interface de forma declarativa (META, 2025i). Essa abordagem torna o código mais intuitivo, facilitando a visualização da estrutura da aplicação (META, 2025i). Ao utilizar JSX, o desenvolvedor pode escrever componentes de maneira similar à criação de elementos HTML (W3SCHOOLS, 2025g). Essa sintaxe é então transpilada para JavaScript puro, garantindo compatibilidade com os navegadores (META, 2025i). Assim, JSX contribui para a clareza e a organização do código (W3SCHOOLS, 2025g).

Em React, a modularidade e a reutilização de código são promovidas por meio do mecanismo de *import* e *export* (W3SCHOOLS, 2025c). Ao exportar um componente, utiliza-se a palavra-chave *export* para torná-lo acessível em outros arquivos do projeto (MOZILLA,

2025d). Existem duas formas principais: a exportação padrão, com *export default*, e a exportação nomeada (MOZILLA, 2025d). Por sua vez, a importação é realizada com o comando *import*, que permite incorporar componentes exportados em diferentes partes do código (MOZILLA, 2025e).

O Listing 2.2.4 exemplifica como realizar a importação nas linhas 1 e 2 e a exportação na linha 13 de um componente em um projeto React. Nele, um componente é importado a partir de um arquivo específico e, após a sua utilização, o componente *PaginaPrincipal* é exportado na linha 13 para ser empregado em outras seções da aplicação. Dessa forma, o Listing 2.2.4 renderiza na tela o texto "Bem-vindo à página principal!" com um cabeçalho, conforme o retorno apresentado na linha 5, evidenciando a aplicação prática deste processo.

Posteriormente, no Listing 2.2.5, o componente *PaginaPrincipal* criado é importado na linha 2 e utilizado na linha 7, retornando não só todo o componente *PaginaPrincipal* como também um componente chamado *Rodape*. Essa técnica é essencial para a construção de aplicações robustas e modulares, incentivando a divisão de responsabilidades entre os componentes (W3SCHOOLS, 2025c).

```
1  import React from 'react';
2  import Cabecalho from './Cabecalho';
3
4  function PaginaPrincipal() {
5    return (
6      <div>
7        <Cabecalho />
8        <p>Bem-vindo à página principal!</p>
9      </div>
10   );
11 }
12
13 export default PaginaPrincipal;
```

Listing 2.2.4 – Exemplo de importação e exportação de componentes em React

2.2.6 Props e State

No exemplo do Listing 2.2.6, *props* são definidas na linha 3 e *state* na linha 4. Nesse exemplo, um componente chamado *Contador* é criado. Esse componente renderiza na tela um número e um botão, e o seu propósito é incrementar mais um no número da linha 8 toda vez que

```
1  import React from 'react';
2  import PaginaPrincipal from './PaginaPrincipal';
3
4  function LayoutDaPagina() {
5    return (
6      <main>
7        <PaginaPrincipal />
8        <Rodape />
9      </main>
10   );
11 }
12
13 export default LayoutDaPagina;
```

Listing 2.2.5 – Exemplo de uso do componente da página principal

o botão da linha 9 é clicado. O valor inicial do contador é uma *props*, e é definido quando este componente for instanciado por um componente pai. O valor contabilizado é definido a partir de um *state*, onde *contador* é o número para exibição no HTML, e *setContator* é a função que vai receber o novo valor que é definido como o número anterior mais um.

O gerenciamento de dados em React se dá por meio de *props* e *state* (META, 2025a). As *props* são utilizadas para passar informações entre componentes, enquanto o *state* controla os dados internos de um componente que podem ser modificados ao longo do tempo (META, 2025a). Essa distinção permite a criação de interfaces dinâmicas, que se adaptam conforme as interações do usuário (META, 2025f). A combinação de *props* e *state* facilita a comunicação e a atualização dos componentes (META, 2025a).

2.2.7 Renderização Condicional

O exemplo no Listing 2.2.7 ilustra a renderização condicional. Nele, um componente *SaudacaoCondicional* é definido na linha 3, e tem uma *props* chamada *estaLogado* que vai definir se o usuário fez *login*. Na linha 9 tem-se o início chaves ({}) que vai permitir a execução de expressões JavaScript. Dessa forma, uma lógica condicional é definida, onde a *props estaLogado* vai definir se o que vai ser renderizado é a linha 7 ou a linha 9. Portanto, o resultado desse componente para o usuário final vai ser o texto "Bem-vindo, usuário!" quando o usuário tiver feito o *login* na aplicação, e o texto "Por favor, faça login." quando o usuário precisar fazer *login*.

A renderização condicional é uma técnica que permite exibir elementos de forma dinâmica, conforme determinadas condições (META, 2025c). Essa abordagem possibilita alterar

```jsx
import React, { useState } from 'react';

function Contador({ valorInicial }) {
  const [contador, setContador] = useState(valorInicial);

  return (
    <div>
      <p>Contador: {contador}</p>
      <button onClick={() => setContador(contador + 1)}>
        Incrementar
      </button>
    </div>
  );
}

export default Contador;
```

Listing 2.2.6 – Exemplo de uso de *props* e *state* em um componente

a interface da aplicação com base em um estado ou nas *props* (W3SCHOOLS, 2025f). Utilizando operadores condicionais, é possível determinar quais elementos serão renderizados em cada situação (W3SCHOOLS, 2025f). Essa flexibilidade é fundamental para criar experiências de usuário interativas e responsivas (META, 2025c).

```jsx
import React from 'react';

function SaudacaoCondicional({ estaLogado }) {
  return (
    <div>
      {estaLogado ? (
        <h1>Bem-vindo, usuário!</h1>
      ) : (
        <h1>Por favor, faça login.</h1>
      )}
    </div>
  );
}

export default SaudacaoCondicional;
```

Listing 2.2.7 – Exemplo de renderização condicional em React

2.2.8 Composition vs Inheritance

O Listing 2.2.8 exemplifica a prática com um componente que utiliza *composition*. Nele é definido dois componentes, o primeiro chama-se *BordaEstilizada*, é definido na linha 3, e recebe uma *props* com o nome *children*. O segundo componente se chama *DialogoBoasVindas*, é definido na linha 11, e utiliza na linha 13 o primeiro componente. O componente *DialogoBoas-Vindas* define como valor da *props children* o conteúdo presente nas linhas 14 e 15. Dessa forma, o componente *BordaEstilizada* recebe o valor como *props* e renderiza ela na linha 6. Portanto, o resultado desses componentes na interface do usuário vai ser os textos "Bem-vindo"e "Obrigado por visitar nossa aplicação."com uma estilização fictícia *borda-estilizada*.

Em React, a abordagem de *composition* permite a reutilização de componentes combinando-os para criar novas *interfaces* (META, 2025b). Essa técnica promove modularidade e flexibilidade, facilitando a manutenção e o reaproveitamento do código (META, 2025b). A *composition* permite encapsular funcionalidades em componentes menores e independentes, evitando estruturas rígidas e complexas (META, 2025b).

```
1  import React from 'react';
2
3  function BordaEstilizada({ children }) {
4    return (
5      <div className="borda-estilizada">
6        {children}
7      </div>
8    );
9  }
10
11 function DialogoBoasVindas() {
12   return (
13     <BordaEstilizada>
14       <h1>Bem-vindo</h1>
15       <p>Obrigado por visitar nossa aplicação.</p>
16     </BordaEstilizada>
17   );
18 }
19
20 export default DialogoBoasVindas;
```

Listing 2.2.8 – Exemplo de *composition* em React

Por outro lado, a abordagem de *inheritance* pode ser utilizada para estender o

comportamento de um componente base, permitindo a herança de propriedades e métodos (META, 2025b). No entanto, essa técnica pode resultar em código mais rígido, dificultando a manutenção e a reutilização em aplicações maiores (META, 2025b).

O Listing 2.2.9 apresenta um exemplo de *inheritance* em React. Nele, os componentes são definidos como classes nas linhas 3 e 12. O componente *Botao* estende a classe padrão importada do React na linha 1 e define o método *handleClick*, que exibe *Clicado* no console ao ser acionado. Em seguida, o componente *BotaoPrimario* herda esse método de *Botao* e sobrescreve o método de renderização para aplicar uma estilização diferenciada (com a classe *primario* na linha 14), alterando a aparência e o rótulo do botão, mas mantendo a ação de clique original. O resultado desse componente na interface do usuário é um botão que imprime "Clicado" no console, e esse botão final tem uma estilização fictícia *primario*.

```
1  import React, { Component } from 'react';
2
3  class Botao extends Component {
4    handleClick() {
5      console.log("Clicado");
6    }
7    render() {
8      return <button
       ↪  onClick={this.handleClick.bind(this)}>Clique</button>;
9    }
10 }
11
12 class BotaoPrimario extends Botao {
13   render() {
14     return <button className="primario"
       ↪  onClick={this.handleClick.bind(this)}>Clique aqui</button>;
15   }
16 }
17
18 export default BotaoPrimario;
```

Listing 2.2.9 – Exemplo de *inheritance* em React

2.2.9 Visão geral dos React Hooks

Nesta seção, apresenta-se uma visão geral dos *Hooks* integrados no React. Introduzidos na versão 16.8 em 2018, os *Hooks* possibilitaram que funcionalidades, antes exclusivas dos componentes baseados em classes, fossem utilizadas em componentes funcionais (BUGL,

```
1  import { useState, useEffect } from "react";
2
3  export const Hooks = () => {
4    const [email, setEmail] = useState("sample@invalid.com");
5
6    useEffect(() => {
7      setTimeout(() => {
8        setEmail("another_sample@invalid.com");
9      }, 10000);
10   });
11
12   return (
13     <div>
14       <p>{email}</p>
15     </div>
16   );
17 };
```

Listing 2.2.10 – Exemplificação dos *Hooks useState* e *useEffect*

2019). Dessa forma, os *Hooks* facilitam o acesso a conceitos fundamentais do React, como *props*, *contexts*, *refs*, estados e o ciclo de vida dos componentes (META, 2023). Essa inovação promove uma abordagem mais direta e intuitiva no desenvolvimento de interfaces (BUGL, 2019).

O Listing 2.2.10 ilustra o exemplo prático que renderiza na interface do usuário um email, e troca esse email depois de 10 segundos. Na linha 4, o *Hook* de estado, *useState*, um estado empregado para armazenar dados que precisam ser monitorados pela aplicação (W3SCHOOLS, 2023). Em seguida, na linha 6, o *Hook* de efeito, *useEffect*, é utilizado para gerenciar efeitos colaterais, como chamadas a uma API, atualizações do *Virtual Document Object Model* (VirtualDOM) e controle de temporizadores (W3SCHOOLS, 2023). Esse exemplo demonstra como os *Hooks* tornam o código mais organizado e legível (META, 2023). Dessa forma, o uso combinado desses recursos reflete uma abordagem prática e modular.

Para garantir o correto funcionamento dos *Hooks*, três regras fundamentais devem ser seguidas: eles devem ser chamados apenas dentro de componentes funcionais, sempre no início desses componentes (ou seja, antes do retorno), e não podem ser invocados de forma condicional (W3SCHOOLS, 2023). O descumprimento dessas diretrizes pode ocasionar comportamentos inesperados e dificultar a manutenção do código.

Além disso, os *Hooks* foram criados para solucionar desafios específicos enfrentados durante o desenvolvimento com React (META, 2023). A primeira motivação reside na necessidade de compartilhar estados entre componentes sem alterar a hierarquia (W3SCHOOLS, 2023).

Em seguida, buscou-se resolver a dispersão e o acoplamento inadequado da lógica de estado, que muitas vezes separa funções inter-relacionadas e une lógicas independentes (W3SCHOOLS, 2023). Por fim, os *Hooks* visam facilitar a compreensão do código, especialmente quando comparados aos métodos baseados em classes (META, 2023). Essas motivações evidenciam a importância dos *Hooks* na evolução das práticas de desenvolvimento.

O React disponibiliza diversos *Hooks* nativos e ainda permite a criação de *Hooks* customizados, bem como a integração com *Hooks* oferecidos por bibliotecas externas, como *Apollo Client* [7], *React Redux* [8] e *React Query* [9]. Essa flexibilidade possibilita encapsular funcionalidades específicas e reutilizá-las em diferentes partes da aplicação. Entre os *Hooks* nativos, podem ser destacados: os *State Hooks* (*useState* e *useReducer*), os *Context Hooks* (*useContext*), os *Ref Hooks* (*useRef* e *useImperativeHandle*), os *Effect Hooks* (*useEffect*, *useLayoutEffect* e *useInsertionEffect*) e os *Performance Hooks* (*useMemo*, *useCallback*, *useTransition* e *useDeferredValue*) (W3SCHOOLS, 2023).

2.2.10 Renderização no React

Para compreender o processo de renderização no React, é fundamental entender o que é o *Document Object Model* (DOM). O DOM é uma interface de programação que representa a estrutura de documentos HTML ou XML como uma árvore de objetos, onde cada elemento da página corresponde a um nó (MOZILLA, 2025b). Essa representação permite que linguagens de script, como JavaScript, acessem e manipulem dinamicamente os elementos e a organização do conteúdo (MDN, 2024a). Contudo, a manipulação direta do DOM pode ser custosa, pois cada alteração pode desencadear reprocessamentos que impactam o desempenho (META, 2025h).

Para mitigar tais desafios, o React adota o conceito de VirtualDOM (META, 2025h). Essa abordagem consiste em manter uma versão simplificada e em memória do DOM real, ou seja, uma representação ideal sincronizada com o DOM real por meio da biblioteca ReactDOM (META, 2025h). Ao ocorrer uma mudança no estado de um componente, o React atualiza primeiramente o VirtualDOM, permitindo um processo de reconciliação que identifica e aplica apenas as alterações necessárias (ROBBESTAD, 2016). Esse mecanismo, conhecido como *diffing*, minimiza as operações onerosas e otimiza a renderização (META, 2025h).

O React não interage diretamente com o DOM real; todas as alterações ocorrem no

[7] *Apollo Client* disponível em: <https://www.apollographql.com/docs/>
[8] *React Redux* disponível em:<https://react-redux.js.org/>
[9] *React Query* disponível em: <https://tanstack.com/query/v3/docs/react/overview>

VirtualDOM. Quando um componente sofre alteração de estado, o *framework* compara a nova versão com a anterior, aplicando somente as mudanças imprescindíveis (ROBBESTAD, 2016). Essa estratégia possibilita atualizações isoladas, onde alterações em determinados elementos não interferem em outros (ROBBESTAD, 2016). Dessa forma, a renderização torna-se mais eficiente, contribuindo para o desempenho geral da aplicação.

Adicionalmente, o React orienta que o desenvolvedor não acesse ou modifique diretamente os elementos da interface por meio do DOM (META, 2025e). Em vez disso, recomenda-se a utilização de *refs* para atrelar um nó a uma referência, permitindo o acesso indireto (META, 2025e). Essa abordagem, realizada por meio do hook *useRef*, garante que as manipulações ocorram via atualização do VirtualDOM, mantendo a consistência da interface (META, 2025e).

O Listing 2.2.11 renderiza na interface do usuário um campo para digitar, e um botão que quando clicado coloca o foco do navegador no campo de texto. Esse exemplo mostra como usar o *hook useRef* para acessar um elemento *input*. Primeiro, o *useRef* é instanciado na linha 4. Em seguida, ele é atribuído ao elemento na linha 12. Por fim, o acesso ao elemento ocorre na linha 6 por meio do método correspondente.

```
1  import { useRef } from 'react';
2
3  function MeuComponente() {
4    const inputRef = useRef(null);
5
6    function handleClick() {
7      inputRef.current.focus();
8    }
9
10   return (
11     <>
12       <input ref={inputRef} type="text" />
13       <button onClick={handleClick}>focus</button>
14     </>
15   );
16 }
17
18 export default MeuComponente;
```

Listing 2.2.11 – Acessando um elemento com *useRef*

2.2.11 Rotas em React

A navegação em aplicações React é essencial para construir interfaces interativas e dinâmicas (META, 2023). O conceito de *routing* permite que o usuário seja direcionado para diferentes páginas sem o recarregamento completo da aplicação (ROUTER, 2023). Em sites tradicionais, cada mudança de rota implica a recarga de todos os recursos, o que pode comprometer a experiência do usuário (ROUTER, 2023). Em contrapartida, aplicações SPA (*Single Page Application*) atualizam apenas os componentes necessários, proporcionando uma navegação mais fluida e eficiente. Dessa forma, a utilização de bibliotecas especializadas torna o processo de transição entre páginas mais ágil e responsivo (MOZILLA, 2025i).

Para implementar o *routing* no React, utiliza-se a biblioteca React Router Dom, que possibilita a atualização da URL e o carregamento de novos componentes sem a necessidade de recarregar toda a página (ROUTER, 2023). A instalação dessa ferramenta é simples e pode ser realizada via NPM, conforme demonstrado no Listing 2.2.12 (ROUTER, 2023). Essa etapa é fundamental para que a aplicação possa gerenciar suas rotas de forma eficaz e escalável, integrando componentes responsáveis pela definição e exibição dos conteúdos.

No Listing 2.2.13, é apresentada a estrutura básica de um componente *Router* utilizando os componentes *BrowserRouter*, *Routes*, *Route* e *Outlet*. O *BrowserRouter* atua como o contêiner principal que engloba as rotas, enquanto o *Outlet* injeta o conteúdo específico de cada rota em um layout comum, composto por elementos como *header* e *footer*. Essa organização modular facilita a manutenção e a escalabilidade da aplicação, permitindo a criação de layouts consistentes para todas as páginas (ROUTER, 2023).

O exemplo do Listing 2.2.14 demonstra a integração dos componentes *Link* e *Route* para a navegação entre páginas. Nesse caso, a rota principal exibe um menu com um link que direciona para a rota de perfil, evitando o recarregamento completo da página. A utilização do componente *Link* assegura transições suaves entre as páginas, reforçando a eficiência do *routing* do lado do cliente. Esse exemplo fornece ao desenvolvedor todos os elementos necessários para a implementação de uma navegação robusta em aplicações React.

```
1 npm i react-router-dom
```

Listing 2.2.12 – Comando para instalar o React Router Dom

```jsx
import { BrowserRouter, Routes, Route, Outlet } from "react-router-dom";
import React from "react";

export const Router = () => {
  return (
    <BrowserRouter>
      <Routes>
        <Route
          path="/"
          element={
            <main>
              <header>header</header>
              <Outlet />
              <footer>footer</footer>
            </main>
          }
        >
          <Route path="/" element={<h1>menu principal</h1>} />
          <Route path="/perfil" element={<h1>perfil do usuário</h1>} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
};
```

Listing 2.2.13 – Componente Router Inicial

2.2.12 Gerenciamento de Estado

O gerenciamento de estado em aplicações React é fundamental para o desenvolvimento de interfaces dinâmicas e escaláveis (META, 2025g). A utilização do *createContext* possibilita o compartilhamento de dados sem a necessidade de repassar propriedades manualmente (REACT, 2025). Essa técnica simplifica a comunicação interna entre componentes e promove uma estrutura de código mais limpa (DODDS, 2025). Adicionalmente, integra-se facilmente com bibliotecas como Redux [10] e MobX [11] para cenários mais complexos (REDUX, 2025). Este tutorial apresenta exemplos práticos sobre o tema.

Antes de explorar os exemplos, observe que o Listing 2.2.15 implementa o contexto para gerenciar o estado que define o tema de cores de uma página *web*. Já o Listing 2.2.16 demonstra a utilização do contexto na aplicação. Renderizando para o usuário o tema atual, e um botão para trocar entre os temas. Essas referências facilitam a compreensão da estrutura modular

[10] Redux disponível em: <https://redux.js.org/>
[11] MobX disponível em: <https://mobx.js.org/>

```
1  import { BrowserRouter, Routes, Route, Outlet, Link } from "react-router-dom";
2
3  export const Router = () => (
4    <BrowserRouter>
5      <Routes>
6        <Route
7          path="/"
8          element={
9            <main>
10             <header>header</header>
11             <Outlet />
12             <footer>footer</footer>
13           </main>
14         }
15       >
16         <Route
17           path="/"
18           element={
19             <>
20               <h1>menu principal</h1>
21               <Link to="/perfil">Perfil</Link>
22             </>
23           }
24         />
25         <Route
26           path="/perfil"
27           element={
28             <>
29               <h1>perfil do usuário</h1>
30               <Link to="/">Voltar</Link>
31             </>
32           }
33         />
34       </Route>
35     </Routes>
36   </BrowserRouter>
37 );
```

Listing 2.2.14 – Componente Router Completo

adotada. Os exemplos ilustram a centralização do estado e evitam o repasse desnecessário de propriedades.

No Listing 2.2.15, é criado um contexto para alternar entre os modos *light* e *dark*. O código define o *ThemeContext* na linha 3 com o *createContext* e configura um *Provider* na linha 5 que encapsula os componentes consumidores. Utiliza-se o *hook useState* para armazenar o valor do tema. Uma função de alternância é definida na linha 8 para atualizar o estado conforme a interação do usuário.

O Listing 2.2.16 exemplifica o uso do contexto em um componente principal denominado *App*. Nele, o *ThemeProvider* na linha 6 envolve o *ThemedComponent* na linha 7, que consome o estado global. Essa estrutura centraliza o gerenciamento do estado e evita a propagação de propriedades. O exemplo reforça a importância de uma arquitetura modular e escalável, pois nele é evidenciado o padrão *composition* aplicado junto com o *Context API* do React.

Em resumo, a utilização do *createContext* com os *hooks* do React é uma solução leve e eficaz para o gerenciamento de estado (DODDS, 2025). A centralização dos dados elimina a necessidade de repassar propriedades entre múltiplos componentes (REACT, 2025). Essa abordagem resulta em um código organizado e de fácil manutenção (REACT, 2025). Embora bibliotecas externas possam oferecer soluções mais robustas, o padrão nativo do React é suficiente para muitos casos (TOOLKIT, 2025).

2.2.13 Estilização das páginas no React

O React não impõe uma metodologia única para a estilização de páginas HTML (META, 2023). Embora seja possível utilizar técnicas tradicionais de CSS em uma página estática, o React traz nuances que precisam ser observadas. Por exemplo, o atributo *class* do HTML é substituído por *className* no React (REACT, 2025). Essa alteração, simples à primeira vista, é crucial para evitar conflitos durante o desenvolvimento. Assim, compreender essas diferenças é essencial para a correta aplicação dos estilos.

Existem três abordagens principais para estilizar uma página React: *inline styling*, *CSS stylesheet* e *CSS Modules* (W3SCHOOLS, 2025h). No primeiro método, os estilos são aplicados diretamente no atributo *style* de um componente (W3SCHOOLS, 2025h). Conforme demonstrado no Listing 2.2.17, o componente define um objeto JavaScript para configurar propriedades de estilo, onde nomes compostos, como *font-family*, são escritos em *camelCase* (W3SCHOOLS, 2025h). Essa técnica é útil para ajustes rápidos e testes de estilos (W3SCHOOLS, 2025h).

A segunda abordagem consiste na utilização de uma *CSS stylesheet*, na qual os estilos são definidos em um arquivo *.css* separado e importado no componente. No Listing 2.2.18, observa-se a importação da folha de estilo, enquanto o Listing 2.2.19 apresenta seu conteúdo (W3SCHOOLS, 2025h). Vale notar que, por ser global, a estilização aplicada afeta todos os elementos que compartilham o mesmo seletor, como a tag *h1*. Essa característica deve ser

```
import React, { createContext, useState, useContext } from 'react';

const ThemeContext = createContext();

function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {
    setTheme(prevTheme => prevTheme === 'light' ? 'dark' : 'light');
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

function ThemedComponent() {
  const { theme, toggleTheme } = useContext(ThemeContext);

  return (
    <div>
      <p>The current theme is {theme}</p>
      <button onClick={toggleTheme}>Toggle Theme</button>
    </div>
  );
}

export { ThemeProvider, ThemedComponent };
```

Listing 2.2.15 – Exemplo de gerenciamento de estado com *createContext*

considerada ao planejar a arquitetura dos estilos na aplicação.

Por fim, a terceira alternativa é o uso de *CSS Modules*, que permite a aplicação de estilos de forma encapsulada (W3SCHOOLS, 2025h). Inicialmente, cria-se um arquivo com extensão *.module.css*, onde são definidos módulos de estilo, como exemplificado no Listing 2.2.20. Em seguida, esses módulos são importados para o componente – usualmente por meio de um objeto denominado *styles* – e aplicados às *tags* por meio do atributo *className*, conforme visto no Listing 2.2.21. Essa abordagem minimiza conflitos e facilita a manutenção dos estilos.

```
import React from 'react';
import { ThemeProvider, ThemedComponent } from './ThemeContext';

function App() {
  return (
    <ThemeProvider>
      <ThemedComponent />
    </ThemeProvider>
  );
}

export default App;
```

Listing 2.2.16 – Exemplo de utilização do contexto no componente *App*

```
const HelloWorld = () => {
  return (
    <div style={{ border: "solid" }}>
      <h1 style={{ color: "red", fontFamily: "fantasy", textAlign:
      ↪  "center" }}>Hello World!</h1>
    </div>
  );
};

export default HelloWorld;
```

Listing 2.2.17 – Componente HelloWorld com estilização *inline*

```
import "./HelloWorld.style.css";

const HelloWorld = () => {
  return (
    <div>
      <h1>Hello World!</h1>
    </div>
  );
};

export default HelloWorld;
```

Listing 2.2.18 – Componente HelloWorld importando uma folha de estilo

```css
div {
  border: solid;
}

h1 {
  color: red;
  font-family: fantasy;
  text-align: center;
}
```

Listing 2.2.19 – Arquivo *HelloWorld.style.css*

```css
.bordaSolida {
  border: solid;
}

.tituloVermelho {
  color: red;
  font-family: fantasy;
  text-align: center;
}
```

Listing 2.2.20 – Arquivo *HelloWorld.module.css*

```jsx
import styles from "./HelloWorld.module.css";

const HelloWorld = () => {
  return (
    <div className={styles.bordaSolida}>
      <h1 className={styles.tituloVermelho}>Hello World!</h1>
    </div>
  );
};

export default HelloWorld;
```

Listing 2.2.21 – Componente HelloWorld com estilização por módulos

2.2.14 Bibliotecas de Componentes

Nesta seção, apresentamos bibliotecas e frameworks de componentes prontos, como Material-UI [12] e Shadcn [13], que aceleram o desenvolvimento de interfaces. Estas ferramentas facilitam a criação de layouts modernos e a padronização visual dos projetos, promovendo a reutilização de código e a agilidade na implementação (COYIER, 2008). O uso de soluções

[12] Material-UI disponível em: <https://mui.com/material-ui/>
[13] Shadcn disponível em: <https://ui.shadcn.com/>

consolidadas contribui para a redução de erros e a melhoria da manutenção. Ademais, a documentação e a comunidade ativa de cada biblioteca oferecem suporte robusto para os desenvolvedores. Dessa forma, evidencia-se a importância de integrar tais recursos na engenharia de *software*.

Segue o Listing 2.2.22, que apresenta o comando utilizado para instalar o Material-UI, uma biblioteca de componentes React de código aberto que implementa o *Material Design* do Google. De forma análoga, o Listing 2.2.23 ilustra o comando para a instalação do Shadcn. Ressalta-se que, conforme o *framework* ou a CLI adotada na inicialização do projeto, a configuração pode variar, sendo recomendada a consulta à documentação oficial para garantir a correta implementação.

```
1 npm install @mui/material @emotion/react @emotion/styled
```

Listing 2.2.22 – Instalação do *Material-UI*

```
1 npx shadcn init
```

Listing 2.2.23 – Instalação do *Shadcn*

O próximo exemplo apresenta a importação e o uso de um componente do Material-UI em um projeto React. O Listing 2.2.24 ilustra como o componente *Button* pode ser importado e incorporado em uma função que define um componente funcional. Esse exemplo vai renderizar na interface do usuário um botão com a estilização definida na biblioteca Material-UI, com a variação de botão *contained*, cor *primary* e com o texto "Clique aqui" definido na linha 8 do componente.

A prática de usar bibliotecas de componentes permite a criação de interfaces interativas com *design* padronizado, facilitando a manutenção e a escalabilidade do código. Ao utilizar componentes prontos, o desenvolvedor concentra-se na lógica de negócio, sem se preocupar com detalhes de estilização.

Por fim, o seguinte exemplo demonstra como importar e utilizar um componente do Shadcn em um ambiente React. No Listing 2.2.25, o componente *Card* é incorporado em uma função que renderiza uma estrutura com cabeçalho e corpo, evidenciando a modularização da interface. Essa abordagem permite a reutilização do componente em diferentes partes do projeto, promovendo um código mais organizado e limpo. A prática de importar componentes prontos reforça a eficiência no desenvolvimento de interfaces.

```jsx
import React from 'react';
import Button from '@mui/material/Button';

function App() {
  return (
    <div>
      <Button variant="contained" color="primary">
        Clique aqui
      </Button>
    </div>
  );
}

export default App;
```

Listing 2.2.24 – Uso do componente *Button* do Material-UI

```jsx
import React from 'react';
import { Card } from 'shadcn';

function DashboardCard() {
  return (
    <Card>
      <Card.Header>
        <h2>Dashboard</h2>
      </Card.Header>
      <Card.Body>
        Conteúdo do cartão.
      </Card.Body>
    </Card>
  );
}

export default DashboardCard;
```

Listing 2.2.25 – Uso do componente *Card* do Shadcn

2.2.15 Integração com APIs

Inicialmente, torna-se necessário compreender os fundamentos que permitem a comunicação entre uma *Representational State Transfer Application Programming Interface* (API REST) e um *front-end* desenvolvido em React (REDHAT, 2023). Para isso, é imprescindível definir o conceito de API REST e esclarecer o funcionamento do *AJAX* (REDHAT, 2023). Esses conceitos formam a base para a integração entre aplicações e possibilitam a criação de sistemas

robustos e escaláveis (REDHAT, 2023).

Uma API REST é uma interface de programação que segue um conjunto de restrições arquiteturais, como a arquitetura cliente/servidor, requisições *Hypertext Transfer Protocol* (HTTP) e comunicação sem estado (REDHAT, 2023). Além disso, destaca-se a importância de uma interface uniforme, a existência de um sistema de camadas e hierarquias, bem como o uso de cache para otimização. Esses atributos padronizados facilitam a interação entre diferentes sistemas, promovendo uma comunicação eficiente e estruturada.

O termo *AJAX* significa *Asynchronous JavaScript e XML* (MOZILLA, 2023). Embora o *XML* tenha sido o formato predominante inicialmente, atualmente o *JavaScript Object Notation* (JSON) é amplamente utilizado por ser mais leve e simples. Essa abordagem assíncrona permite que dados sejam atualizados na interface sem a necessidade de recarregar toda a página, contribuindo para uma experiência de usuário mais fluida e interativa.

Para simplificar o desenvolvimento e aprimorar o entendimento, opta-se pelo uso do cliente AXIOS [14]. Essa biblioteca JavaScript possibilita o envio de requisições HTTP, o tratamento de promessas e a manipulação de funcionalidades do *AJAX*, convertendo automaticamente as respostas das API para JSON (AXIOS, 2023). Devido a essas características, o AXIOS revela-se uma escolha interessante para ser utilizado tanto no *front-end* quanto no *back-end* (KHOLMATOV, 2023).

Como exemplo prático, o Listing 2.2.26 apresenta os comandos necessários para instalar o AXIOS em um projeto React. Esses comandos devem ser executados no terminal, conforme o gerenciador de pacotes adotado pelo desenvolvedor, demonstrando a facilidade de integração da biblioteca ao ambiente de desenvolvimento com o uso de um *package manager*.

```
1 npm install axios
```

Listing 2.2.26 – Comandos para instalar o AXIOS

Ademais, o Listing 2.2.27 exemplifica a implementação de uma função para obter dados de um usuário por meio da API do GitHub. No código, o AXIOS é configurado para acessar o *endpoint* de usuários na linha 3, realizando múltiplas requisições (ver da linha 9 até linha 13) que capturam informações relevantes como repositórios, seguidores e outros dados e os armazena em um possível estado na linha 22. Assim, o exemplo ilustra de forma prática como integrar um *front-end* com uma API utilizando o AXIOS.

[14] AXIOS disponível em: <https://axios-http.com/ptbr/docs/intro>

```
import axios from "axios";

const apiGithub = axios.create({
baseURL: "https://api.github.com/users",
});

const fetchData = async (userName) => {
try {
  const result = await apiGithub.get(`/${userName}`);
  const stars = await apiGithub.get(`/${userName}/starred`);
  const repos = await apiGithub.get(`/${userName}/repos`);
  const followers = await apiGithub.get(`/${userName}/followers`);
  const following = await apiGithub.get(`/${userName}/following`);
  const userData = {
    ...result.data,
    stars: stars.data.length,
    repos_list: repos.data,
    stars_list: stars.data,
    followers_list: followers.data,
    following_list: following.data,
  };
  setData(userData);
} catch (e) {
  console.log(e);
  setData({});
}
};
```

Listing 2.2.27 – Consulta na API do Github com AXIOS

3 METODOLOGIA

Este capítulo apresenta a metodologia empregada neste estudo para conduzir um trabalho que investiga as práticas de *clean code* e *design patterns* praticadas por desenvolvedores. Este trabalho é dividido em duas partes, as quais são: aplicação de um *survey* a desenvolvedores e a mineração de repositórios provenientes de aplicações de código aberto no GitHub. Essa estratégia visa proporcionar uma compreensão da adoção de *clean code* e *design patterns* no desenvolvimento de *software* em específico de *front-end* de aplicações *web*.

Na primeira parte do estudo, a coleta dos repositórios de *software* é realizada por meio do consumo de uma API, que possibilitaram extrair informações relevantes de *issues* e *pull requests* de repositórios previamente selecionados. Esse método garante a obtenção de um conjunto consistente e abrangente de dados. O procedimento está detalhado nas Seções 3.3 e 3.4, evidenciando o rigor adotado.

Na segunda parte do estudo, a condução do estudo é organizada em duas etapas principais. A primeira envolve a análise do perfil profissional dos desenvolvedores, utilizando as respostas do *survey* para oferecer um contexto enriquecido sobre os participantes. A segunda etapa concentra-se na extração e organização dos dados dos repositórios, identificando discussões significativas relacionadas ao uso de React em práticas de *clean code* e *design patterns*.

3.1 Questões de pesquisa

Este estudo investiga a adoção de *clean code* e *design patterns* no desenvolvimento front-end com React, analisando seu impacto na qualidade do código e na eficiência do desenvolvimento, além de buscar preencher uma lacuna na literatura acadêmica sobre a integração dessas práticas no contexto do React. Para alcançar esse objetivo, foram formuladas quatro Questões de Pesquisa (QP), cujas respostas são identificadas com base do *survey* com desenvolvedores e da mineração de repositórios de *software* no GitHub. A seguir, são apresentadas as QP que guiam este estudo:

QP1: Os desenvolvedores conseguem identificar boas práticas de *clean code* e e o uso de *design patterns* nas suas implementações em React? Esta questão busca identificar o nível de conhecimento dos desenvolvedores sobre suas aplicações. Dessa forma, pode-se compreender melhor o perfil dos desenvolvedores participantes, e então indicar um nível de exposição dos usuários a práticas de qualidade de *software* e ao React.

QP2: Quais as dificuldades e os benefícios de seguir *design patterns* em projetos com React? O intuito dessa questão é compreender os desafios e as vantagens da adoção de *design patterns* no desenvolvimento com React, considerando o nível de conhecimento dos desenvolvedores identificado na **QP1**. Busca-se avaliar como esses padrões influenciam a qualidade do código e a eficiência do desenvolvimento. Além de mapear os desafios experimentados pelos desenvolvedores, também investiga-se as vantagens constatadas na prática do desenvolvimento com React. Essa análise é essencial para entender a relevância dos *design patterns* na organização e manutenção do código, auxiliando no desenvolvimento de boas práticas que possam otimizar a produtividade e a colaboração entre os desenvolvedores.

QP3: Como os desenvolvedores utilizam as técnicas de *clean code* e *design patterns* ao desenvolver seus sistemas com React? O objetivo dessa questão é analisar de que maneiras os desenvolvedores aplicam na prática os conceitos de *clean code* e *design patterns* em projetos com React. A partir das respostas do *survey* e da mineração de repositórios no GitHub, busca-se identificar padrões de codificação, boas práticas adotadas e possíveis divergências entre a teoria e a implementação real. Com isso, pretende-se compreender se há um alinhamento entre o conhecimento teórico e a aplicação prática dessas técnicas, além de fornecer insights sobre como essas abordagens impactam a qualidade e a manutenção do código.

QP4: Quais são os artefatos de mais reuso no React? O objetivo dessa questão é identificar quais artefatos de reuso são mais utilizados no desenvolvimento com React, considerando tanto os dados coletados no *survey* quanto a mineração de repositórios no GitHub. Busca-se compreender quais abordagens, bibliotecas, frameworks, modelos, API's e etc, são mais adotados pelos desenvolvedores, e como esses artefatos contribuem para a modularidade, a eficiência e a manutenção do código.

3.2 Etapas da condução do estudo

A Figura 3 ilustra as etapas deste estudo, os quais estão divididos em três fases. A primeira fase consiste na elaboração de um protocolo geral (APÊNDICE C), incluindo a definição dos objetivos de pesquisa, a identificação do público-alvo, a formulação das perguntas do *survey* e a seleção das palavras-chave para a mineração de repositórios de *software*.

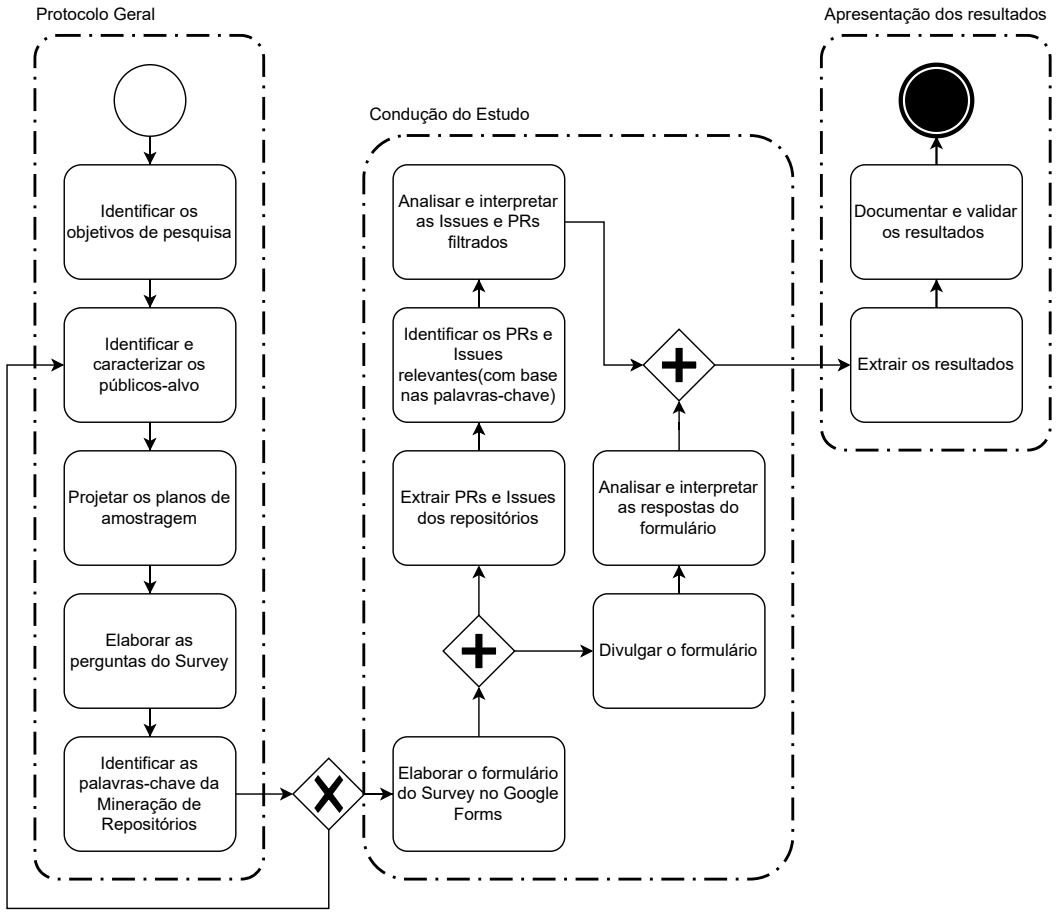
Em seguida, a segunda fase envolve a criação do *survey* no Google Forms [1], seguida de sua divulgação. Paralelamente, procede-se à extração e filtragem com base nas palavras-chave

[1] Google Forms disponível em: <https://workspace.google.com/intl/pt-BR/products/forms/>

dos *pull requests* e *issues* dos repositórios previamente selecionados. Essa etapa é fundamental para coletar dados relevantes e garantir a representatividade das informações.

A etapa final abrange a análise e interpretação dos *pull requests* e *issues*, juntamente com os resultados obtidos pelo *survey*. Por fim, os resultados foram documentados e submetidos a validações cruzadas, permitindo responder às QP. Essa análise integrada contribui para a robustez das conclusões do estudo.

Figura 3 – Metodologia para a condução do estudo



Fonte: Próprio autor

3.3 Configuração do survey

As perguntas do formulário (APÊNDICE A) são organizadas para facilitar a legibilidade, e são agrupadas em três categorias principais. A primeira categoria abrange questões exclusivamente relacionadas ao React. A segunda categoria concentra-se em *design patterns*. A terceira categoria aborda práticas associadas a *clean code*. Esse agrupamento contribui para uma análise estruturada dos dados.

Para cada pergunta há um identificador no formato “X.Y”. O número “X” indica o tópico correspondente, enquanto “Y” representa a posição sequencial da pergunta dentro do tópico. Essa codificação padronizada facilita a organização e a referência cruzada. Com esse sistema, torna-se mais simples rastrear e interpretar as respostas. A análise dos resultados é, assim, realizada de maneira mais eficiente.

Em uma seção distinta, porém presente no mesmo formulário, é elaborada uma série de perguntas para caracterizar o perfil do desenvolvedor (APÊNDICE B). Essa seção visa identificar aspectos profissionais e comportamentais. Todas as questões apresentadas são de múltipla escolha, permitindo uma rápida quantificação das respostas. Essa abordagem fortalece a avaliação do perfil profissional dos participantes.

Para a aplicação da pesquisa, foram elaborados dois formulários: o primeiro contendo as seções e perguntas descritas nos (APÊNDICES A e B), e o segundo, com as mesmas questões e estrutura, porém traduzido para o inglês. O primeiro formulário foi distribuído para comunidades acadêmicas e grupos de desenvolvedores no Brasil por email, enquanto o segundo, em inglês, foi compartilhado com os desenvolvedores cujos emails foram extraídos dos repositórios do Github utilizados na mineração. Estes desenvolvedores representam uma comunidade global, abrangendo profissionais de diversas partes do mundo.

3.4 Condução da mineração de repositórios de Software

Para a mineração de repositórios, foram selecionadas as três empresas mais populares que dispõem de perfil no GitHub, utilizando a contagem de estrelas — uma métrica usada no Github para um usuário favoritar o repositório — como critério de seleção. Para cada perfil de empresa foi selecionado o maior repositório de código aberto com a marcação React, também levando em consideração o número de estrelas. Dessa forma, os repositórios selecionados foram o *fluentui* [2] da Microsoft, *blockly-samples* [3] do Google e *superset* [4] da Apache.

A coleta dos dados foi realizada utilizando uma API pública, com um código [5] desenvolvido em Python. Esse código foi responsável por baixar todas as *threads* — que correspondem às 63.209 *issues* e *pull requests* — dos repositórios selecionados, realizando

[2] Repositório do *fluentui* disponível em: <https://github.com/microsoft/fluentui>
[3] Repositório do *blockly-samples* disponível em: <https://github.com/google/blockly-samples>
[4] Repositório do *superset* disponível em: <https://github.com/apache/superset>
[5] Código para minerar repositórios disponível em: <https://github.com/MarcosVini9999/mine-repositories-github>

requisições *POST* na *API GraphQL* [6] e armazenando as respostas localmente em arquivos no formato JSON.

Cada *thread* salva localmente contém informações detalhadas de uma *issue* ou *pull request*, como o usuário que criou o tópico, título, descrição, usuários que participaram do tópico com seus respectivos comentários, data de criação e data de fechamento, além de outros metadados, como IDs, URLs, emails e sites pessoais.

Com base no conteúdo apresentado no Capítulo 2, foi definido um conjunto de palavras-chave para realizar a filtragem inicial das *threads*. O Quadro 1 apresenta todas as palavras-chave utilizadas nesse processo. Essas palavras-chave servem como critério para identificar *threads* potencialmente relevantes para os tópicos de *clean code* e *design patterns*.

Quadro 1 – Palavras-chave relacionadas a padrões de design e conceitos de programação

Design Patterns: Singleton, Factory Method, Adapter, Composite, Observer, Strategy, HOC, Higher Order Component, Hooks, Hook, Presentational, Render Props, Client Side, CSR, Static Generation, SSG, SSR, Progressive Hydration, Selective Hydration, Server Side Rendering, Static Rendering, Parameterize, Partial Application, Call-Backs With Continuations, Maps, Monoids, Monads. Clean Code: YAGNI, KISS, DRY, SOLID, Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle, Strive For Totality, Static Types.

Fonte: Próprio autor

Para otimizar o processo de filtragem, foi desenvolvido um código em *BASH* para o ambiente Linux. Esse código percorre os arquivos JSON armazenados localmente, aplicando o filtro baseado nas palavras-chave apresentados no Quadro 1. A filtragem foi feita diretamente no conteúdo textual das *threads* (títulos, descrições e comentários), identificando discussões relevantes. As *threads* que atendem aos critérios definidos tem seus IDs extraídos e salvos em um arquivo *.txt*.

Com os IDs das *threads* filtradas, a análise manual foi iniciada. Cada *thread* pode ser acessada por meio da URL correspondente ao seu ID. Durante essa análise manual, duas pessoas leem e interpretam se a *thread* estava relacionada aos tópicos de *clean code* ou *design patterns*, e qual foi a discussão principal dessa *thread* que a faz se encaixar em um dos tópicos. Por fim, a etapa de análise manual foi concluída, e as *threads* que se enquadravam nos tópicos foram selecionadas para compor a análise final da mineração de repositórios.

[6] *API GraphQL* do Github disponível em: <https://docs.github.com/pt/graphql>

3.5 Análise e interpretação dos resultados

Para a análise dos resultados, foi desenvolvido um ambiente de processamento e visualização dos dados. Primeiramente, para o formulário, utilizou-se um *Jupyter Notebook* [7] na plataforma *Google Colab* [8]. Nesse ambiente, os arquivos *.csv* dos dois formulários, cada um em uma linguagem distinta, português e inglês, foram importados e processados. Cada pergunta é analisada individualmente e representada por meio de gráficos ou anotações que permitem uma melhor assimilação dos dados.

Para a mineração de repositórios de *software*, foi criada uma planilha [9] no *Google Planilhas* para organizar e sistematizar os dados coletados. Nessa planilha, registraram-se informações referentes ao protocolo geral (APÊNDICE C) e às palavras-chave utilizadas, compondo uma tabela com dados relevantes das *threads* analisadas. Cada linha da tabela contém informações como o link para a *thread*, seu ID, o usuário do GitHub, a descrição original, o trecho relevante da *thread*, bem como a categorização do tópico abordado. Além disso, foi feita uma descrição do propósito de cada *thread*, seguida de uma análise crítica e uma interpretação final.

[7] *Jupyter Notebook* disponível em: <https://jupyter.org/>
[8] *Google Colab* disponível em: <https://colab.google/>
[9] Dados coletados na mineração de repositórios: <https://docs.google.com/spreadsheets/d/1CS1-YcUQz5WoeKu7Q9HnFDxtvKoQ_k9ydvXSuJJJTNE/edit?usp=sharing>

4 RESULTADOS

A Seção 4.1 é dedicada à apresentação dos resultados do *survey*, dividida em duas seções: a primeira aborda o perfil profissional dos desenvolvedores participantes, enquanto a segunda detalha as respostas relacionadas aos tópicos de React, *clean code* e *design patterns*. A Seção 4.2 apresenta os resultados da mineração de repositórios, estruturada em três seções, a primeira aborda *clean code*, a segunda sobre *design patterns*, e a terceira apresenta considerações sobre os demais resultados da mineração. Por fim, a Seção 4.3 discute as respostas das Questões de Pesquisa.

4.1 Survey

Esta seção está organizada em duas seções, cada uma dedicada a uma análise específica dos resultados do *survey*. Primeiro, são apresentados os resultados sobre o perfil dos desenvolvedores que responderam ao *survey*. Já a Seção 4.1.2 concentra-se nos resultados relacionados a tópicos técnicos, abordando React, *clean code* e *design patterns*. Dessa forma, são apresentadas as respostas de 23 desenvolvedores.

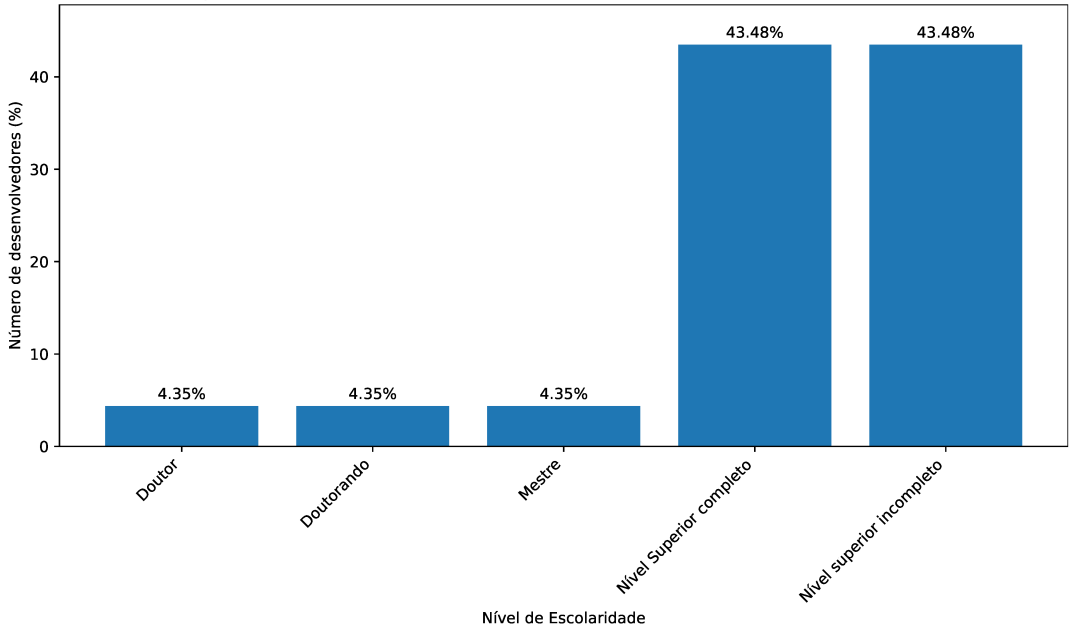
4.1.1 Perfil

Para compreender melhor o perfil dos desenvolvedores participantes deste estudo (17,39% são desenvolvedores estrangeiros), foi coletado um conjunto de dados profissionais. Esses dados incluem aspectos como o nível de escolaridade, cargo atual, tempo de experiência profissional e envolvimento com desenvolvimento de software.

A Figura 4 apresenta a distribuição dos participantes segundo o nível de escolaridade. Observa-se que o grupo é composto predominantemente por profissionais com formação técnica ou acadêmica avançada, o que pode refletir uma maior exposição a práticas de qualidade de software. Entre os participantes, 4,35% possuem titulação de Doutor, 4,35% Doutorando 4,35% e 4,35% Mestre. A maioria dos participantes do *survey* possui Ensino Superior Completo (43,48%) ou Ensino Superior em Curso (43,48%).

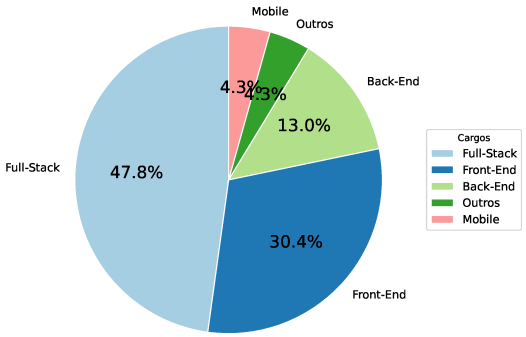
A Figura 5 apresenta a distribuição dos cargos ocupados pelos participantes no momento da realização do *survey*. Verifica-se que a maior parte dos respondentes atua como desenvolvedores *full-stack* (47,8%), seguidos por desenvolvedores *front-end* (30,4%), *back-end* (13%), *mobile* (4,3%) e outros (4,3%). Esses dados evidenciam a relevância profissional dos

Figura 4 – Distribuição do nível de escolaridade



Fonte: Elaborado pelo autor

Figura 5 – Cargos dos desenvolvedores do survey

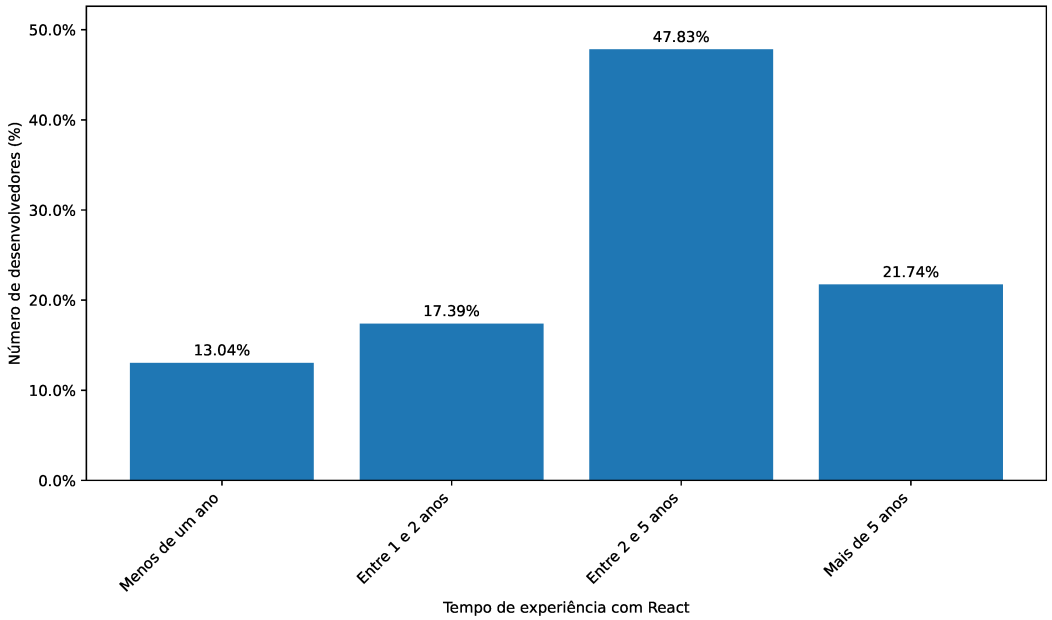


Fonte: Elaborado pelo autor

participantes no contexto deste estudo, uma vez que a maioria (78,2%) trabalha diretamente com desenvolvimento *front-end* ou áreas correlatas, o que está em consonância com a tecnologia abordada neste trabalho. Assim, os resultados obtidos reforçam a pertinência dos *insights* gerados pela pesquisa para a compreensão de práticas de qualidade de software aplicadas ao React.

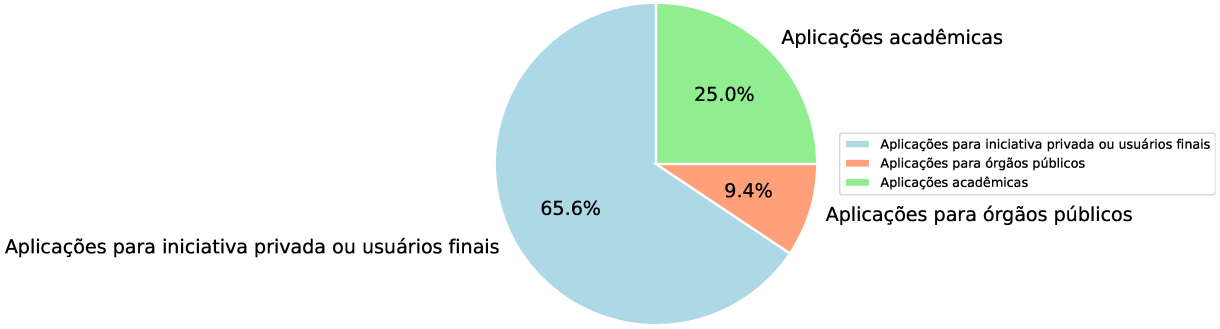
A Figura 6 ilustra a distribuição dos participantes em relação ao tempo de experiência no desenvolvimento de aplicações React. Observa-se que 21,74% dos respondentes possuem mais de 5 anos de experiência com a tecnologia, enquanto a maioria (47,83%) situa-se na faixa de 2 a 5 anos de experiência. Além disso, 17,39% possuem entre 1 e 2 anos de experiência, e 13,04% apresentam menos de 1 ano de atuação com React. Esses dados evidenciam um equilíbrio entre profissionais em início de carreira e aqueles com experiência consolidada, proporcionando uma base diversificada para a análise dos diferentes níveis de adoção de práticas de qualidade de

Figura 6 – Experiência em desenvolvimento de aplicações com React



Fonte: Elaborado pelo autor

Figura 7 – Tipos de aplicações React desenvolvidas



Fonte: Elaborado pelo autor

software.

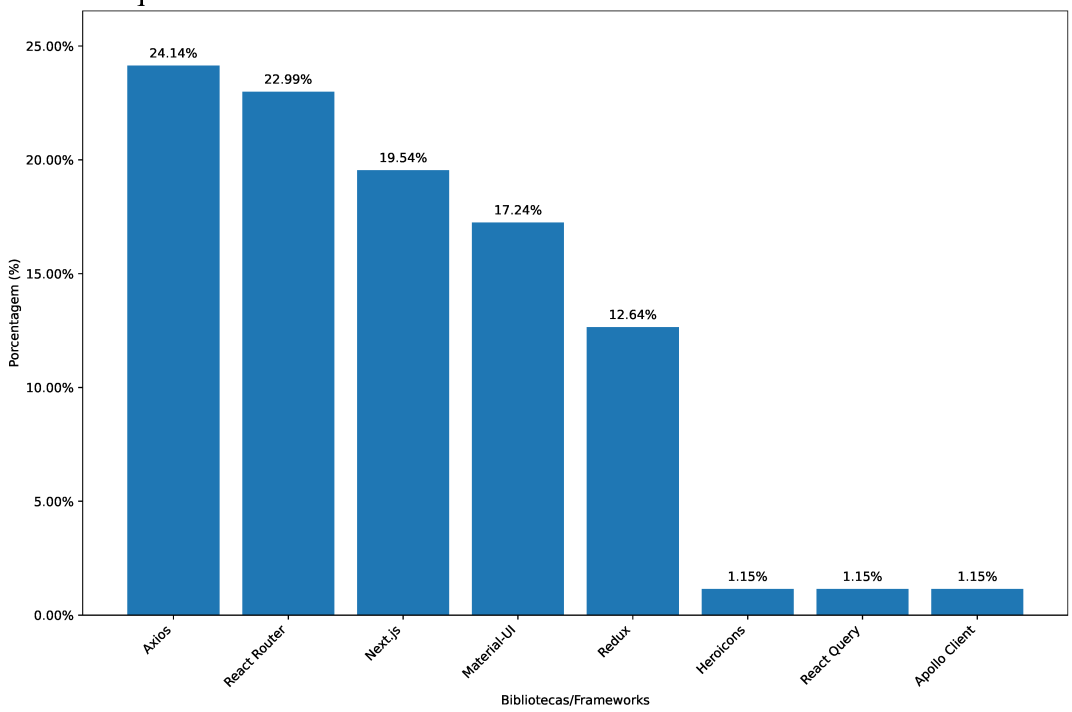
4.1.2 Resultados do survey

Resultados gerais sobre desenvolvimento com React: A Figura 7 apresenta a distribuição dos tipos de aplicações React desenvolvidas pelos participantes. A maioria (65,6%) utiliza o React para criar aplicações voltadas à iniciativa privada ou para usuários finais, seguida por aplicações acadêmicas (25%) e para órgãos públicos (9,4%). Esses resultados refletem a predominância do React em contextos comerciais, enquanto sua presença em setores especializados, como o acadêmico e público, destaca sua flexibilidade para atender a demandas diversas.

A Figura 8 ilustra as bibliotecas e frameworks mais adotados em projetos React, destacando a predominância do AXIOS (24%) e do React Router (23%). Essa preferência

evidencia a valorização de princípios como a separação de responsabilidades e a modularidade, fundamentais para a criação de aplicações escaláveis e de fácil manutenção. Ferramentas como o AXIOS e o React Router promovem uma arquitetura organizada ao isolar preocupações específicas, integrando boas práticas de *clean code* e *design patterns*.

Figura 8 – Frequência de Bibliotecas/Frameworks usados com React

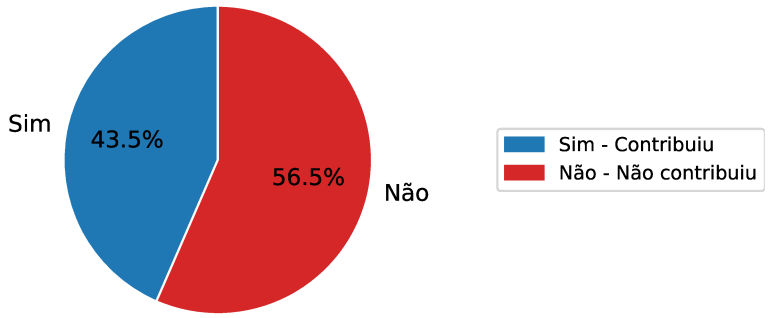


Fonte: Elaborado pelo autor

A Figura 9 demonstra que 43,5% dos participantes possuem experiência no desenvolvimento de plataformas, *frameworks*, APIs, *middlewares* ou bibliotecas relacionadas ao React, enquanto 56,5% não possuem tal vivência. Essa participação ativa evidencia que práticas de qualidade, como o *clean code* e *design patterns*, são essenciais não só na construção de aplicações, mas também no desenvolvimento de ferramentas que impactam diretamente outros profissionais.

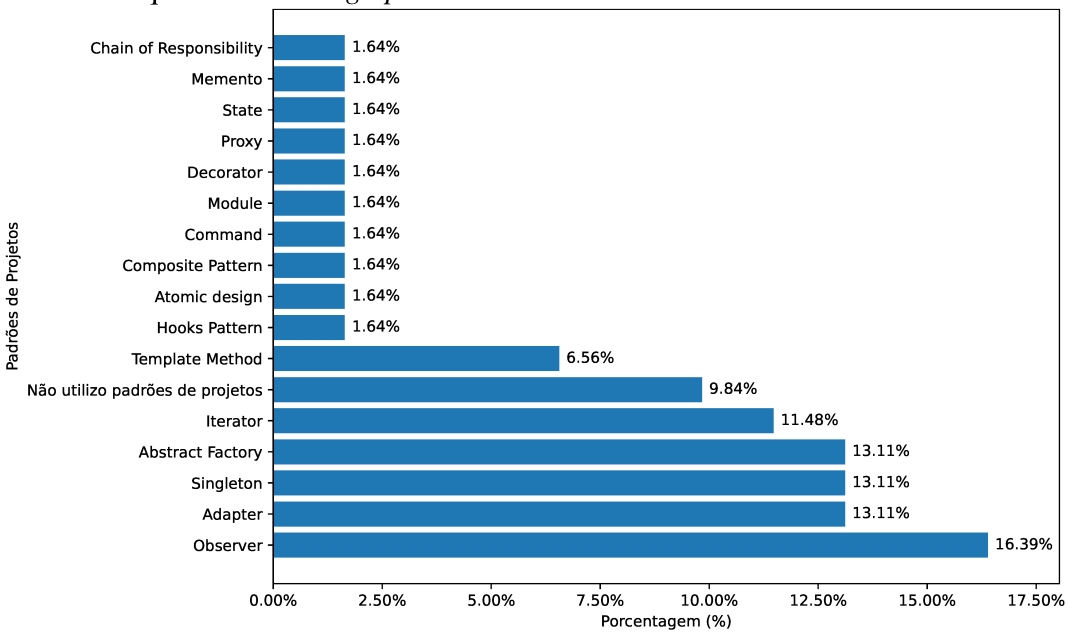
Resultados sobre *Design Patterns*: A Figura 10 apresenta a frequência de adoção dos *design patterns* pelos participantes do *survey*. Nota-se que os padrões *Abstract Factory* (13,11%), *Singleton* (13,11%), *Adapter* (13,11%) e *Observer* (16,39%) destacam-se como os mais utilizados, evidenciando uma preferência por soluções que promovem a modularidade e a reutilização de componentes. Em contrapartida, a baixa adoção de padrões como *Command* (1,64%) pode indicar uma possível limitação de conhecimento técnico ou menor aplicabilidade prática em projetos desenvolvidos com React.

Figura 9 – Contribuição para desenvolvimento de plataformas/frameworks/APIs/middlewares/-bibliotecas React



Fonte: Elaborado pelo autor

Figura 10 – Frequência de *Design patterns* utilizados



Fonte: Elaborado pelo autor

A Pergunta 2.2.2 (APÊNDICE A) investiga o uso de *design patterns* no desenvolvimento de alguma plataforma, *framework*, API, *middlewares* ou bibliotecas de *software* específicas para aplicações com React. Os participantes relataram a adoção de padrões como *Factory Method*, *Singleton*, *Adapter*, *Decorator*, *Strategy*, *Composite* e *Abstract Factory*, além do *Hooks Pattern* específico do React. Isso evidencia uma preocupação com a modularidade e a reutilização de código.

A Pergunta 2.3 (APÊNDICE A) questiona as dificuldades e benefícios de usar *design patterns* no React. As respostas indicam que, embora os *design patterns* ofereçam benefícios como manutenibilidade, reutilização de código e escalabilidade, sua complexidade adicional pode representar desafios, sobretudo em projetos pequenos ou para desenvolvedores menos

experientes. Alguns participantes mencionaram a curva de aprendizado e a necessidade de maior familiaridade com ferramentas, como *hooks* e contextos. Entre os benefícios, destaca-se a melhoria na organização e legibilidade do código, que facilita a manutenção e a colaboração; contudo, o uso excessivo de *design patterns* pode gerar uma complexidade desnecessária e a adaptação de conceitos tradicionais de *back-end* para o *front-end* é apontada como um entrave.

Resultados sobre *Clean Code:* Os resultados das Perguntas 3.1.1, 3.1.2 e 3.1.3 (APÊNDICE A) abordam o uso de TypeScript — uma linguagem de programação de código aberto desenvolvida pela Microsoft que adiciona tipagem ao JavaScript — como uma influência na manutenibilidade. Os resultados indicam que os desenvolvedores percebem sim um impacto positivo em projetos React. Todos afirmam que o uso dessa tecnologia melhora significativamente a manutenibilidade do código, ressaltando a importância da tipagem estática para a identificação de erros e a consistência da base de código.

A Pergunta 3.2 (APÊNDICE A) analisa a aplicação de boas práticas na escrita de código com React, tais como o uso de princípios como o SOLID. Alguns desenvolvedores destacam que sua implementação melhora a legibilidade, escalabilidade e modularização do código, e enfatizaram também que usam princípios como somente o princípio da responsabilidade única, KISS ou DRY. Enquanto outros consideram que a adaptação integral de SOLID pode introduzir um nível de abstração desnecessário no *front-end*, e 26% dos desenvolvedores enfatizaram que não usam nenhum princípio. Essa diversidade de opiniões demonstra que, para alguns, a adoção completa dos princípios é essencial, e para outros, apenas aspectos específicos são empregados para separar responsabilidades e facilitar a manutenção.

A Pergunta 3.3 (APÊNDICE A) destaca as boas práticas que devem ser adotadas em projetos com React. A componentização é considerada fundamental, reduzindo a redundância e melhorando a manutenção. O gerenciamento de estado, por meio de ferramentas como Redux, Context API e Zustand, é apontado como uma prática válida para garantir previsibilidade, embora sua adoção deva ser compatível com a complexidade do projeto.

A adoção de TypeScript ou *Prop Types* para a tipagem das propriedades, bem como a organização do código por meio de uma estrutura clara de pastas e arquivos, também foram enfatizadas. Entretanto, a inconsistência na aplicação dessas práticas pode resultar em projetos desorganizados, dificultando a navegação e a manutenção, especialmente em equipes de maior porte.

Além disso, o uso de *hooks* e componentes funcionais são boas práticas evidenciadas,

assim como a melhor nomeação das entidades (constantes, métodos, funções, entre outras), diminuição de comentários, diminuição de depuração (consoles no código), o uso de uma boa indentação de código e documentação são boas práticas abordadas do *survey*.

4.2 Mineração de repositórios de software

A mineração de repositórios de software foi conduzida com o intuito de identificar as dificuldades e práticas associadas à adoção de *clean code* e *design patterns* em projetos React. Esse processo possibilitou a extração e análise de dados significativos sobre os problemas enfrentados pelos desenvolvedores, bem como as estratégias adotadas para solucioná-los. Entre os resultados obtidos, oito casos foram destacados e são apresentados nas Tabelas 3 e 4, com o objetivo de oferecer uma visão mais detalhada, facilitando a interpretação das principais dificuldades e soluções encontradas.

Os casos destacados abrangem aspectos como clareza de código, duplicação de código, reuso, complexidade de componentes, otimização, testes, múltiplas responsabilidades, além de padrões como *presentational* e *singleton*, e princípios fundamentais como o DRY.

4.2.1 Resultados sobre clean code

A Tabela 3 sumariza as principais *threads* sobre *clean code*. O Item 1 aborda a simplicidade e clareza de código por meio da utilização de uma propriedade de entrada no componente, o que simplifica a lógica interna e torna o comportamento do componente mais previsível e compreensível. Por sua vez, o Item 2, aborda o compartilhamento e reuso de *media queries* do CSS entre componentes do tipo cartão, prática que favorece a consistência e a simplicidade no estilo dos componentes.

O Item 3 enfatiza a necessidade de reorganizar seções e controles para reduzir a complexidade e melhorar a legibilidade do código, promovendo, assim, uma maior manutenibilidade. Por fim, o Item 4 trata diretamente da complexidade dos componentes, ressaltando que códigos mais complexos dificultam a compreensão e os testes dos mesmos. A solução implementada consistiu na decomposição do componente em dois componentes menores, o que facilitou sua manutenção e testabilidade.

Tabela 3 – Principais resultados sobre *clean code* da mineração de repositório de software

Identificador	Thread	Dificuldades/Problemas Identificados
1	feat: dataset add modal	O uso da *prop formMode*, conforme discutido no *Pull Request*, contribuiu para uma maior simplicidade e clareza no código, alinhando-se aos princípios do *Clean Code*.
2	feat: home screen mvp	O compartilhamento e reuso de *media queries* nos cartões promoveriam uma maior simplicidade, reduziriam a duplicação de código e estabeleceriam um padrão consistente de *media queries* entre os cartões, beneficiando todos os componentes envolvidos.
3	[TODO] Clean up controlPanels/sections and controls in superset-front-end	A *Issue* trata da limpeza e organização dos arquivos relacionados aos *control panels* e *controls* no front-end do *Apache Superset*, que são componentes da interface de usuário utilizados na exploração de gráficos e relatórios. Dessa forma, busca-se reduzir a duplicação de código e melhorar a reusabilidade.
4	test: TableCollection.tsx (front-end) - Dividing into smaller components and creating tests	Este *Pull Request* trata da complexidade dos componentes, afirmando que um código mais complexo torna o componente mais difícil de ser compreendido e testado. Além disso, são abordadas soluções por meio da decomposição do componente complexo em dois componentes menores, que juntos formam a solução. Posteriormente, discute-se o uso de *hooks*, como o *useMemo*, para otimização, testes e *clean code*.

Fonte: Próprio autor

4.2.2 Resultados sobre design patterns

A Tabela 4 sumariza os principais resultados sobre princípios de desenvolvimento de *software* e *design patterns*. O Item 1 apresenta que a aplicação do princípio da responsabilidade única se mostrou eficaz, promovendo a separação lógica das responsabilidades e garantindo uma arquitetura mais limpa e compreensível. Além disso, foram introduzidos testes para assegurar uma maior confiabilidade e robustez no comportamento dos componentes.

No Item 2 mostra a recomendação da adoção do padrão *presentational*, que separa a lógica de apresentação da lógica de negócios. Essa separação torna o componente mais simples e reutilizável, facilitando a manutenção e promovendo maior consistência na interface do usuário.

O Item 3, por sua vez, reforça a aplicação do princípio DRY, eliminando redundâncias no código e favorecendo a reutilização. Essa abordagem não só simplifica a manutenção como também reduz a possibilidade de inconsistências no decorrer do projeto. Finalmente, o Item 4 implementa uma instância *singleton* do *Switchboard* para compartilhar estado entre o painel incorporado e o elemento pai que carrega o *iframe*, facilitando a comunicação e o gerenciamento de estado.

Tabela 4 – Resultados sobre princípios de desenvolvimento de *software* e *design patterns* da mineração de repositório de software

Identificador	Thread	Dificuldades/Problemas Identificados
1	test: DatabaseSelector | Splitting the file and creating tests	O componente está assumindo múltiplas responsabilidades, o que compromete sua manutenção e testabilidade. Dessa forma, aplica-se o uso do princípio de responsabilidade única no React, que busca dividir o componente em partes menores e mais especializadas.
2	Explore control panel - Chart control, TimeFilter, GroupBy, Filters	O *Pull Request* incentiva a adoção do padrão de componentes *presentational*, que se concentram exclusivamente em como os dados são exibidos ao usuário, promovendo uma separação clara de responsabilidades.
3	fix(chart): legacyRegularTime and charts are being rendered, and solves issue	O usuário sugere a criação de um controle compartilhado que inclua a *time section*, tornando essa funcionalidade reutilizável entre diferentes gráficos, como o *t-test* e *event flow*. Isso aplica o princípio *DRY*, centralizando a lógica e simplificando a manutenção do código.
4	feat(embedded-dashboard): Share Switchboard State for Sending Events from Plugins	Uma nova funcionalidade adiciona uma instância *singleton* do *Switchboard* para compartilhar estado entre o painel incorporado e o elemento pai que carrega o *iframe*. Isso permite enviar interações e dados, como filtros selecionados, ao elemento pai. O *singleton* é inicializado sob demanda, eliminando a necessidade de múltiplas instâncias no painel.

Fonte: Próprio autor

4.2.3 Considerações finais da mineração de repositórios de software

Além dos casos apresentados nas Tabelas 3 e 4, a mineração de repositórios revelou uma série de problemas e soluções adicionais de grande relevância. Entre os casos observados, o uso de validação estática no código, particularmente com React e ESLint, foi eficaz na identificação precoce de erros, o que contribuiu para a melhoria da qualidade do código. Outro aspecto notável foi a adoção de componentização, uma estratégia amplamente empregada para promover maior modularidade e reutilização de código. Além disso, observou-se a ênfase no uso de componentes funcionais com *hooks*, uma abordagem que facilita a manutenção do código e torna as aplicações mais escaláveis.

No contexto da otimização da performance, duas práticas se destacaram: a utilização de padrões de *front-end* para renderizações *client-side* e a renderização no servidor (*SSR*). Ambas as abordagens contribuíram de forma significativa para melhorar a experiência do usuário e a performance geral das aplicações. A discussão sobre renderização e ordenação no *front-end* também levou à solução do padrão *lazy loading*, que permite o carregamento sob demanda de componentes, contribuindo para a eficiência do sistema.

Adicionalmente, foi analisado o uso de técnicas de *Higher-Order Components* (HOC) para otimizar a interação entre elementos da interface. A solução, que envolveu o uso de *forwardRef* e utilitários especializados para manipulação de árvores de componentes, se mostrou eficiente ao permitir o aninhamento de gatilhos e a propagação de propriedades de maneira modular e extensível.

Entretanto, um desafio técnico significativo foi identificado na renderização no servidor, onde alguns componentes apresentaram problemas de compatibilidade, evidenciando a complexidade associada à integração entre renderização no servidor e bibliotecas, o que impactou o desempenho em alguns projetos.

Além disso, nas descrições de alguns componentes, foi utilizado o termo *Composition Pattern*, o qual facilita a criação de interfaces modulares no React, promovendo a divisão da UI em componentes menores que podem ser combinados de forma flexível para atender a diferentes necessidades.

Por fim, as palavras-chave destacadas no Quadro 1, utilizadas na mineração de repositórios descrita na Seção 3.4, foram encontradas de forma recorrente em centenas de casos. Estas palavras apareceram como nomes de arquivos, pastas, componentes e scripts no *front-end*, evidenciando sua aplicação e relevância no desenvolvimento de software analisado.

4.3 Discussões

Nessa seção são discutidas as repostas para as QP com base nos resultados do *survey* e da mineração de repositórios de *software*.

4.3.1 QP1: Os desenvolvedores conseguem identificar boas práticas de clean code e e o uso de design patterns nas suas implementações em React?

Os resultados do *survey* indicam que os desenvolvedores possuem um entendimento variado sobre *clean code* e *design patterns* no contexto do React. Enquanto práticas como a componentização e o uso de tipagem estática (TypeScript) são amplamente adotadas, observa-se que a aplicação de princípios mais avançados, como SOLID, gera divergências entre os profissionais. Parte dos desenvolvedores defende que esses princípios promovem modularidade e manutenibilidade, enquanto outros questionam sua aplicabilidade em um ambiente predominantemente funcional como o React.

A mineração de repositórios reforça essa percepção ao identificar o uso de padrões como *Singleton*, *Adapter*, *Factory Method* e outros, mas também evidencia dificuldades na adoção de princípios de responsabilidade única e modularização. Assim, o estudo revela que, embora haja um conhecimento disseminado sobre boas práticas, a aplicação efetiva desses conceitos, mesmo que pequeno, ainda enfrenta desafios, sobretudo na adoção de padrões tradicionais da POO para o paradigma funcional do React.

4.3.2 QP2: Quais as dificuldades e os benefícios de seguir design patterns em projetos com React?

Os resultados expuseram benefícios quanto desafios na adoção de *design patterns* em projetos React. Entre as vantagens, observa-se a melhoria na organização do código, a reutilização de componentes e a maior previsibilidade do comportamento da aplicação. A utilização de padrões como *Presentational* e *Singleton* promove a separação de responsabilidades, reduzindo o acoplamento e tornando o código mais sustentável a longo prazo.

No entanto, as dificuldades também são significativas. A curva de aprendizado é apontada como um obstáculo, especialmente para desenvolvedores menos experientes. Além disso, a introdução excessiva de padrões pode resultar em complexidade desnecessária, tornando a manutenção do código mais trabalhosa do que seria em uma abordagem mais pragmática.

Vale destacar também que por volta de 10% das respostas do *survey* informaram que não conhece ou não aplica *design patterns*. Isso pode estar relacionado à mudança do paradigma de desenvolvimento com React, que abandonou o uso de classes em favor de componentes funcionais e *hooks*, reduzindo a necessidade de certos padrões tradicionais de POO.

Além disso, algumas respostas do *survey* evidenciaram um ceticismo sobre a aplicação de *design patterns* no contexto do React. Alguns desenvolvedores argumentam que *"design patterns são soluções de POO para problemas de POO"* e que *"React já mostrou não ser um contexto bom para aplicação desses princípios"*. Isso sugere que parte da comunidade não concorda ou não se alinha ao uso de *design patterns* junto com React.

Contudo, a pesquisa também evidencia que parte significativa dos desenvolvedores adotam ativamente *design patterns* no desenvolvimento com React. O uso de padrões como *Factory Method*, *Singleton*, *Adapter* e *Decorator* foi relatado nas respostas do *survey*, indicando que, apesar das resistências, esses padrões continuam sendo empregados. A mineração de repositórios também reforça esse cenário, com exemplos concretos de implementações do

princípio da responsabilidade única, *Presentational Pattern*, reutilização de código por meio do DRY e dentre outros padrões.

Dessa forma, há uma clara dualidade na adoção de *design patterns* no React: enquanto alguns desenvolvedores os consideram úteis e aplicáveis, outros os veem como desnecessários ou mesmo inadequados para o paradigma atual do React. Essa divergência aponta para a necessidade de uma análise contextualizada na adoção dessas práticas, garantindo que sua implementação realmente agregue valor ao desenvolvimento.

4.3.3 QP3: Como os desenvolvedores utilizam as técnicas de clean code e design patterns ao desenvolver seus sistemas com React?

Os resultados apontam que os desenvolvedores adotam *clean code* e *design patterns* de maneira seletiva, priorizando práticas que impactam diretamente a legibilidade e a manutenibilidade do código. A componentização surge como uma das técnicas mais empregada, garantindo modularidade e reutilização de código. Além disso, a tipagem com TypeScript e a utilização de ferramentas como ESLint auxiliam na identificação precoce de problemas.

No entanto, a adoção de *design patterns* ainda é limitada. De acordo com os resultados do *survey*, os padrões mais utilizados incluem *Observer*, *Adapter*, *Singleton*, *Abstract Factory* e *Iterator*. Ainda assim, uma parte considerável dos desenvolvedores relatou não utilizar *design patterns* em seus sistemas, o que sugere uma lacuna no conhecimento ou na necessidade percebida dessas práticas no desenvolvimento com React.

A mineração de repositórios de software revelou que padrões como *Singleton*, *Presentational Pattern*, DRY, *Factory Method*, *Composition Pattern* e outros que são amplamente utilizados. O *Singleton*, por exemplo, foi aplicado para gerenciar estados globais, enquanto o *Presentational Pattern* ajudou a separar a lógica de negócios da camada de apresentação. Além disso, a adoção do princípio DRY foi notável, com desenvolvedores buscando minimizar repetições de código por meio de abstrações reutilizáveis.

4.3.4 QP4: Quais são os artefatos de mais reuso no React?

A análise dos dados obtidos por meio do *survey* e da mineração de repositórios evidencia que os artefatos de reuso em projetos React são multifacetados e refletem uma preocupação contínua com a modularidade, a manutenibilidade e a escalabilidade do código. De um lado, os *design patterns* que foram apontados pelos desenvolvedores como estratégias im-

portantes para estruturar a aplicação, promovendo a criação de componentes reutilizáveis e a separação clara de responsabilidades. Essa preferência ressalta o valor desses padrões na redução da duplicação de código e na facilitação da testabilidade, embora a sua aplicação direta ao contexto React nem sempre seja a solução mais adequada para todos os cenários.

Entretanto, a mineração dos repositórios revelou que o maior artefato de reuso está centrado nos *hooks*. Esse recurso, intrínseco à filosofia funcional do React, aparece com destaque em nomes de arquivos, funções, componentes e pastas, indicando sua onipresença e relevância na extração e compartilhamento de lógica. Os *hooks* possibilitam a separação e a reutilização de comportamentos, como o gerenciamento de estado e os efeitos colaterais, de maneira elegante e modular. Assim, enquanto os padrões tradicionais oferecem uma estrutura conceitual sólida, os *hooks* representam a evolução prática dessa ideia, permitindo uma composição mais fluida e adaptada às necessidades específicas de cada projeto.

Além dos artefatos apresentados, os resultados apontam que bibliotecas e *frameworks* complementares — como o AXIOS e o React Router — também desempenham um papel crucial no reuso de funcionalidades, isolando preocupações específicas e promovendo uma arquitetura mais organizada. A componentização, evidenciada na decomposição de componentes complexos em unidades menores, reforça os princípios de *Clean Code* e o conceito DRY (*Don't Repeat Yourself*), contribuindo para a manutenção e escalabilidade das aplicações. Ademais, a tipagem com TypeScript e o uso de ferramentas como ESLint auxiliam na identificação precoce de problemas, refletindo uma preocupação contínua com a qualidade do código.

Contudo, é importante adotar esses artefatos com discernimento. A centralidade dos *hooks* no reuso de código ressalta sua eficácia, mas também impõe a necessidade de uma implementação criteriosa para evitar complexidades desnecessárias ou o acoplamento excessivo. Dessa forma, os benefícios dos *hooks* e demais práticas devem ser balanceados com a simplicidade e clareza do *design*, garantindo que as soluções adotadas estejam em sintonia com as características e desafios de cada projeto.

Em síntese, os artefatos de mais reuso no React com base no resultados do *survey* e mineração de repositórios englobam, primordialmente, a utilização intensiva de *hooks*, que se destacam na mineração de repositórios, aliada aos *design patterns* e ao uso estratégico de bibliotecas e *frameworks* complementares. Essa combinação de práticas, contribui para um desenvolvimento mais alinhado às demandas contemporâneas do ecossistema React.

5 AMEAÇAS À VALIDADE

Neste capítulo, discutem-se as ameaças à validade identificadas no presente estudo, categorizadas em quatro dimensões: construção, conclusão, internas e externas. A definição dessas ameaças baseia-se no trabalho de Lima *et al.* (2014), que apresenta uma classificação consolidada sobre o tema.

A Seção 5.1 aborda as ameaças de construção do presente estudo, isto é, pontos que estão relacionados a fatores humanos ou a má definição da base teórica. Ademais, a Seção 5.2 fala sobre as ameaças de conclusão, que envolve uma análise correta dos resultados, confiabilidade dos resultados, e das implementações usadas para tratar e filtrar os resultados (LIMA *et al.*, 2014).

Por fim, tem-se as ameças internas presente na Seção 5.3 e as ameaças externas presente na Seção 5.4. A primeira são pontos do trabalho que determina se a relação entre o tratamento e o resultado é de causa e efeito, sem a interferência de outros fatores não avaliados (WOHLIN *et al.*, 2024). A segunda avalia se os resultados do estudo podem ser aplicados a outros contextos, verificando se a relação de causa e efeito é realmente válida fora do nosso estudo (WOHLIN *et al.*, 2024).

5.1 Ameaças de Construção

Uma ameaça de construção diz respeito à possível falta de compreensão dos desenvolvedores ao responderem o *survey*. Perguntas abertas e de múltipla escolha foram utilizadas para minimizar ambiguidades. Ainda assim, o risco de interpretações equivocadas dos termos técnicos – como os nomes dos *design patterns*, à *clean code* e ao React – pode ter afetado a qualidade dos dados, pois os participantes podem basear seus comportamentos nas opções apresentadas nas questões fechadas presentes no formulário.

5.2 Ameaças de Conclusão

O processo de análise e extração dos dados provenientes do *survey* e da mineração dos repositórios pode estar sujeito à interpretação subjetiva para chegar a uma conclusão correta, o que constitui outra ameaça à validade de conclusão. A aplicação de técnicas como as questões de pesquisa, e a triangulação dos resultados pode reduzir esse viés.

5.3 Ameaças Internas

Uma ameaça interna está relacionada à organização das perguntas do *survey*. A disposição dos itens pode incentivar os participantes a selecionarem apenas as alternativas apresentadas, sem refletir adequadamente sobre suas respostas. Para mitigar esse viés, foi inserido um campo de resposta aberta em diversas questões, permitindo que os desenvolvedores expressem suas opiniões de forma mais livre.

5.4 Ameaças Externas

Em relação às ameaças externas, a amostra de entrevistados pode ter sido limitada. Inicialmente, os formulários foram distribuídos para comunidades acadêmicas e grupos de desenvolvedores no Brasil, e, adicionalmente, para profissionais internacionais cujos e-mails foram extraídos dos repositórios utilizados na mineração de repositórios. Essa estratégia pode restringir a generalização dos resultados para o universo global de desenvolvedores, considerando as diferenças culturais e contextuais na utilização do React. Porém, os resultados do *survey* foram complementados com a mineração.

Outra ameaça externa refere-se à seleção dos repositórios de código usados na mineração. Embora tenha sido optado por analisar os maiores repositórios React de três grandes organizações (Microsoft, Google e Apache), essa escolha pode não abranger a totalidade dos projetos existentes, introduzindo um viés de seleção que limita a validade externa dos resultados.

Por fim, a metodologia de coleta dos dados, que incluiu técnicas de mineração de repositórios e a análise de *pull requests* e *issues* (conforme detalhado nos Capítulos 3 e 4), está sujeita a riscos relacionados à precisão e integridade das informações extraídas. A dependência de ferramentas automatizadas e de algoritmos de filtragem pode ocasionar erros sistemáticos. Para contornar essa limitação, realizou-se uma validação manual dos dados, embora o risco não seja completamente eliminado.

6 TRABALHOS RELACIONADOS

A literatura voltada para a investigação de *design patterns* e *clean code* em aplicações React revela uma escassez notável de trabalhos que abordam diretamente esse tema. Embora existam estudos que tangenciam aspectos relevantes, identificar pesquisas que se concentram especificamente na aplicação dessas práticas por desenvolvedores *front-end* é desafiador. Essa dificuldade reforça a necessidade de ampliar a discussão e promover investigações mais direcionadas.

Os estudos relacionados abordam tópicos que apresentam tanto semelhanças quanto diferenças em relação ao presente trabalho. Por exemplo, há similaridades no uso de *survey* em pesquisas como as de Ferreira *et al.* (2023), Silva *et al.* (2022), Moreira *et al.* (2023), Ljung e Gonzalez-Huerta (2022) e Bandi (2016), que exploram questões relativas a *clean code* ou *design patterns*—mas nunca ambos simultaneamente. Ademais, esses trabalhos não se dedicaram a uma análise focada no React nem empregaram técnicas auxiliares, como a mineração de repositórios de *software*.

Em contrapartida, estudos como os de Ferreira *et al.* (2023) e Farias (2022) investigaram bibliotecas e *frameworks* (incluindo o React) e temas como *clean code* e *design patterns*. Contudo, tais pesquisas não realizaram validações cruzadas que combinassem *survey* com mineração de repositórios exclusivamente no contexto do React.

Adicionalmente, os trabalhos de Gomes *et al.* (2023), Ljung e Gonzalez-Huerta (2022) e Laksri *et al.* (2022) discutem impactos, princípios e práticas relacionados a *design patterns* ou *clean code*, mas sem abordar esses tópicos especificamente no ambiente do React. Por fim, destaca-se que Laksri *et al.* (2022) adota uma estratégia de mineração de *design patterns* que se assemelha à abordagem de mineração de repositórios de *software* utilizada neste trabalho.

O estudo de Silva *et al.* (2022) utiliza questionários para investigar a legibilidade, manutenibilidade e compatibilidade de navegadores com novas funcionalidades do JavaScript. Já Moreira *et al.* (2023) combina um *survey* com revisão sistemática para analisar ferramentas de detecção de *design patterns*.

Trabalhos como os de Ljung e Gonzalez-Huerta (2022) e Bandi (2016) realizam pesquisas com questionários para avaliar a percepção dos desenvolvedores sobre práticas de *clean code* e violações arquiteturais. O estudo experimental de Gomes *et al.* (2023) evidencia o impacto dos *design patterns* na performance de aplicações web.

Por fim, Farias (2022) apresenta uma investigação exploratória que combina revisão

bibliográfica e experimentos com frameworks como React, ressaltando a aplicabilidade dos *design patterns* e a necessidade de aprimorar as práticas de desenvolvimento. Essa análise reforça a lacuna existente quanto ao foco específico nas práticas de desenvolvedores *front-end*, justificando a relevância do presente estudo. As Tabelas 5 e 6 resumem e comparam os trabalhos.

Tabela 5 – Comparação dos trabalhos relacionados

Trabalho	Abordagem	Objetos de pesquisa	Resultados
Presente estudo (2025)	Mineração de repositórios de *software* no Github e pesquisa por questionário (*survey*).	Investigar o uso de *Design Patterns* e *Clean Code* por desenvolvedores front-end que utilizam React.	O estudo concluiu que a aplicação de *design patterns* e práticas de *clean code* em projetos React melhora a clareza, modularidade e manutenção do código, apesar de desafios na adaptação de conceitos tradicionais de POO ao paradigma funcional.
Laksri *et al.* (2022)	Mineração de dados (*empirical analysis*) em posts do Stack Overflow.	Design patterns e seus contextos de design em repositórios do Stack Overflow.	Desenvolvimento de uma taxonomia de contextos de design e um método automatizado (DPC Miner) para minerar conhecimento sobre design patterns e contextos de design.
Ferreira *et al.* (2023)	Pesquisa com questionários e desenvolvimento de ferramentas (como o ReactSniffer) para identificar *code smells*.	Identificar fatores que motivam a adoção de *frameworks* Front-end, criar um catálogo de *code smells* associados a problemas de design em aplicações React e entender as operações de refatoração realizadas por desenvolvedores.	Foram identificados 12 *code smells* específicos para React, e a ferramenta *ReactSniffer* detectou um total de 2.565 ocorrências desses smellys em projetos populares do GitHub que usam React. Além disso, foram catalogadas 69 operações distintas de refatoração, com 25 sendo específicas para React.
Silva *et al.* (2022)	Pesquisa por questionário (*survey*).	Legibilidade e manutenibilidade do código, compatibilidade de navegadores com novas funcionalidades do JavaScript, e a importância de ferramentas como o Babel para transpilação de código.	Foram respondidos questionários por 54 desenvolvedores, alcançando uma taxa de resposta de 72%. Os principais motivos para adoção de novas funcionalidades foram relacionados à qualidade do código, como legibilidade e velocidade de desenvolvimento. Foi detectada a presença significativa do Babel em projetos JavaScript, destacando sua importância na superação da incompatibilidade entre novas funcionalidades e navegadores antigos.
Moreira *et al.* (2023)	*Survey* e Revisão Sistemática da Literatura.	Ferramentas de Detecção de Padrões de Projeto (DPP), características, desempenho e usabilidade.	Identificaram 42 ferramentas de DPP, com destaque para a precisão e cobertura em detecção de padrões; também apontaram baixa concordância entre os resultados de diferentes ferramentas e perceberam preferências dos desenvolvedores por ferramentas específicas baseadas em contextos e estruturas internas.

Fonte: Próprio autor

Tabela 6 – Comparação dos trabalhos relacionados

Trabalho	Abordagem	Objetos de pesquisa	Resultados
Presente estudo (2025)	Mineração de repositórios de *software* no Github e pesquisa por questionário (*survey*).	Investigar o uso de *Design Patterns* e *Clean Code* por desenvolvedores front-end que utilizam React.	O estudo concluiu que a aplicação de *design patterns* e práticas de *clean code* em projetos React melhora a clareza, modularidade e manutenção do código, apesar de desafios na adaptação de conceitos tradicionais de POO ao paradigma funcional.
Ljung e Gonzalez-Huerta (2022)	Foi realizado um *survey* (questionário) junto a desenvolvedores e uma revisão sistemática da literatura.	Os princípios e práticas de Clean Code.	A maioria dos participantes concorda com os princípios de Clean Code, afirmando que eles facilitam a leitura, compreensão, reutilização e manutenção do código, embora haja controvérsias em algumas práticas específicas.
Bandi (2016)	*Survey* (pesquisa por questionário)	Violações de arquitetura e anti-design patterns em software.	25% dos desenvolvedores não conheciam violações arquiteturais, 36% já tinham ouvido falar, mas não entendiam bem, e apenas 4% tinham conhecimento profundo e aplicavam essas práticas frequentemente. Além disso, foram identificadas cinco categorias de perspectivas sobre violações arquiteturais: não aderência à arquitetura original, falta de qualidade do software, decisões de design ruins, falta de habilidades dos desenvolvedores e considerações de custo-benefício.
Gomes *et al.* (2023)	Estudo experimental (avaliação comparativa de implementações com e sem design patterns).	Impacto do uso de design patterns no desempenho computacional de aplicações web.	Implementações com design patterns mostraram melhor desempenho em termos de renderização e resposta, com redução significativa no consumo de memória RAM e CPU.
Farias (2022)	Estudo exploratório que inclui uma revisão bibliográfica e experimentos com frameworks (React e Vue.js).	Engenharia de software, desenvolvimento web, padrões de projeto e frameworks.	Identificação da aplicabilidade de padrões de projeto em diferentes frameworks e a necessidade de melhorias na qualidade do código e nas práticas de desenvolvimento.

Fonte: Próprio autor

7 CONCLUSÕES E TRABALHOS FUTUROS

7.1 Conclusões

Este trabalho investigou a aplicação de *design patterns* e *clean code* no desenvolvimento de aplicações React, combinando os dados quantitativos e qualitativos obtidos por meio de um *survey* aplicado a desenvolvedores e da mineração de repositórios de software (ver Capítulos 3 e 4).

A motivação para este estudo reside na crescente demanda por interfaces dinâmicas e na necessidade de manter a clareza, a manutenibilidade e a escalabilidade dos projetos, sobretudo em um cenário marcado pela pluralidade de frameworks e ferramentas. O objetivo principal foi avaliar o nível de conhecimento dos profissionais e identificar, de forma crítica, os desafios e benefícios decorrentes da aplicação dessas práticas no ambiente React.

Os dados quantitativos do *survey* demonstram que a base de participantes possui uma formação sólida, com 43,48% dos respondentes apresentando Ensino Superior Completo, 47,83% com experiência entre 2 e 5 anos em React e 21,74% com mais de 5 anos de atuação na tecnologia. Em relação à utilização de *design patterns*, os resultados indicam que, embora aproximadamente 26% dos desenvolvedores tenham declarado não conhecer ou não aplicar tais padrões, os demais relataram a adoção de soluções como *Abstract Factory*, *Singleton*, *Adapter* e *Observer*.

Quanto a *clean code*, os desenvolvedores enfatizaram a importância de evitar problemas como *Long Method* e *Duplicate Code*, práticas essenciais para garantir a qualidade e a manutenibilidade do código. Assim como usar TypeScript para impactar positivamente na legibilidade e manutenibilidade do código para React.

A mineração de repositórios complementou os achados do *survey* ao identificar casos práticos que ilustram a aplicação dos conceitos teóricos. Foram destacados exemplos que evidenciam a melhoria na clareza e modularidade do código – como o uso do conceito DRY para a manutenção e escalabilidade das aplicações – bem como a implementação de *design patterns*, como o padrão *Singleton* para o compartilhamento de estado e o padrão *Presentational* para a separação entre a lógica de apresentação e de negócios.

A análise integrada dos resultados revela uma dualidade importante entre uma pequena parte (26%) dos participantes do *survey*: embora os desenvolvedores valorizem práticas que aprimoram a legibilidade e a manutenção do código, há resistência na aplicação direta de

conceitos tradicionais oriundos da programação orientada a objetos, como os princípios SOLID, quando confrontados com o paradigma funcional predominante no React.

Essa realidade indica a necessidade de adaptar as boas práticas ao ambiente específico do desenvolvimento *front-end*, de forma a equilibrar inovação e aderência a padrões consolidados. As limitações apontadas no Capítulo 5 – relativas à interpretação subjetiva dos participantes e à seleção dos repositórios analisados – reforçam que os resultados devem ser interpretados com cautela e abrem caminho para futuras investigações com amostras ampliadas e análises qualitativas aprofundadas, conforme sugerido pelos Trabalhos Relacionados (Capítulo 6).

Os benefícios decorrentes deste estudo são amplos e atingem diversos perfis de profissionais. Desenvolvedores iniciantes podem utilizar os *insights* aqui apresentados para compreender melhor as práticas recomendadas e evitar erros comuns, enquanto profissionais experientes encontrarão subsídios para aprimorar a qualidade e a manutenção de seus projetos com React.

Ademais, estudantes e pesquisadores terão à disposição dados quantitativos e qualitativos que contribuem para o avanço do conhecimento no desenvolvimento *front-end*, possibilitando a construção de diretrizes práticas que promovam a sustentabilidade e a eficiência das aplicações web.

Em síntese, os achados deste estudo fornecem uma compreensão crítica dos desafios e benefícios envolvidos na aplicação de *design patterns* e *clean code* em projetos React, contribuindo para o aprimoramento das práticas de desenvolvimento e para a evolução contínua do ambiente *front-end*.

7.2 Trabalhos Futuros

Como trabalhos futuros que permitam expandir a pesquisa deste trabalho, propõe-se:

- **Expansão para outros frameworks:** Investigar a aplicação de práticas de *clean code* e *design patterns* em outros *frameworks* e bibliotecas de desenvolvimento de interfaces web, como o Angular [1] e o Vue [2]. Essa comparação poderá revelar diferenças na adoção de boas práticas e na manutenção do código, ampliando o entendimento sobre a aplicabilidade dos conceitos estudados.
- **Coleta de dados em redes sociais e fóruns:** Realizar estudos que envolvam a coleta

[1] Angular disponível em: <https://angular.dev/>
[2] Vue disponível em: <https://vuejs.org/>

e análise de dados oriundos de redes sociais e fóruns de perguntas e respostas, como o *Stack Overflow* [3], onde discussões sobre *design patterns* e *clean code* são frequentes. Essa abordagem poderá complementar os dados obtidos pelo *survey* e pela mineração de repositórios, fornecendo uma visão mais abrangente dos desafios enfrentados pela comunidade de desenvolvedores.

- **Integração de outras dimensões da qualidade de software:** Explorar a intersecção entre as práticas de *clean code* e *design patterns* com outras áreas da qualidade de software, como segurança, desempenho e testes. Tal integração pode contribuir para o desenvolvimento de aplicações mais robustas e eficientes, considerando uma visão holística dos fatores que influenciam a manutenção e evolução dos sistemas.
- **Expansão da mineração de repositórios:** Ampliar a mineração de repositórios realizada no Github para incluir repositórios de outras organizações, abrangendo diferentes tipos, como organizações de menor porte, para analisar se as práticas de desenvolvimento e manutenção variam conforme o contexto organizacional.

Em síntese, os resultados deste estudo reforçam a importância de se adotar boas práticas de desenvolvimento para garantir a qualidade e a manutenibilidade do código em projetos React. As direções apontadas para trabalhos futuros visam não apenas validar e expandir os achados aqui apresentados, mas também contribuir para o avanço contínuo da engenharia de software no contexto do desenvolvimento de aplicações web.

[3] *Stack Overflow* disponível em: <https://stackoverflow.com/>

REFERÊNCIAS

AMAZON. **O que é uma CLI (Interface de linha de comando)?** 2025. Acessado em 25 de fevereiro de 2025: <https://aws.amazon.com/pt/what-is/cli/>.

AXIOS. **AXIOS doc**. 2023. Acessado em 17 de Novembro de 2023: <https://axios-http.com/ptbr/docs/intro>.

BANDI, A. Developers' perspectives on architecture violations: A survey. In: **25th International Conference on Software Engineering and Data Engineering, SEDE 2016**. [S.l.: s.n.], 2016. p. 91–96.

BERTOLI, M. **React Design Patterns and Best Practices**. [S.l.]: "Packt", 2017.

BUGL, D. **Learn React Hooks: Build and refactor modern React. js applications using Hooks**. [S.l.]: Packt Publishing Ltd, 2019.

COYIER, C. **What Are The Benefits of Using a CSS Framework?** 2008. Acessado em 26 de fevereiro de 2025: <https://css-tricks.com/what-are-the-benefits-of-using-a-css-framework/>.

CRAMER, H. **Ask the Expert: A Recap on Patient-Reported Outcomes**. [S.l.]: Mary Ann Liebert, Inc., publishers 140 Huguenot Street, 3rd Floor New . . . , 2024. 207–208 p.

DODDS, K. C. **How to use React Context effectively**. 2025. Acessado em 26 de fevereiro de 2025: <https://kentcdodds.com/blog/how-to-use-react-context-effectively>.

FARIAS, L. H. C. R. Estudo comparativo da utilização de design patterns no desenvolvimento de aplicação web utilizando frameworks front-end. 2022.

FEATHERS, M. C. **Working Effectively with Legacy Code**. [S.l.]: Prentice Hall, 2004.

FERREIRA, F. da S. *et al.* Assisting javascript front-end developers in maintaining and evolving react-based applications: Code smells and refactoring operations. Universidade Federal de Minas Gerais, 2023.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. [S.l.]: Addison-Wesley, 1994.

GOMES, F.; SOARES, P.; ARAÚJO, A. A. Examining the performance implications of design patterns in front-end web development: A preliminar comparative study with react and vue. js. In: **Proceedings of the 29th Brazilian Symposium on Multimedia and the Web**. [S.l.: s.n.], 2023. p. 255–259.

HUNT, A.; THOMAS, D. The trip-packing dilemma [agile software development]. **IEEE Software**, IEEE, v. 20, n. 3, p. 106–107, 2003.

JARTARGHAR, H. A.; SALANKE, G. R.; AR, A. K.; SHARVANI, G.; DALALI, S. React apps with server-side rendering: Next. js. **Journal of Telecommunication, Electronic and Computer Engineering (JTEC)**, v. 14, n. 4, p. 25–29, 2022.

KHOLMATOV, A. WIDELY USED LIBRARIES IN THE JAVASCRIPT PROGRAMMING LANGUAGE AND THEIR CAPABILITIES. **Intent Research Scientific Journal**, 2023.

KOPPALA, J. **ERP Solution with ReactJS**. [S.l.]: TBD Publishing, 2021. Chapter 3: The evolution from XHP to ReactJS.

LAKSRI, W.; ALDEIDA, A.; TINGTING, B.; TANG, A. Mining and relating design contexts and design patterns from stack overflow. **Empirical Software Engineering**, Springer Nature BV, v. 27, n. 1, 2022.

LAZUARDY, M. F. S.; ANGGRAINI, D. Modern front end web architectures with react. js and next. js. **Research Journal of Advanced Engineering and Science**, v. 7, n. 1, p. 132–141, 2022.

LIMA, V. C. M.; NETO, A. G. S. S.; EMER, M. C. F. P. Investigação experimental e práticas ágeis: ameaças à validade de experimentos envolvendo a prática ágil programação em par. **Revista Eletrônica de Sistemas de Informação**, v. 13, n. 1, 2014.

LJUNG, K.; GONZALEZ-HUERTA, J. "to clean code or not to clean code" a survey among practitioners. In: SPRINGER. **International Conference on Product-Focused Software Process Improvement**. [S.l.], 2022. p. 298–315.

LUBURIĆ, N.; VIDAKOVIĆ, D.; SLIVKA, J.; PROKIĆ, S.; GRUJIĆ, K.-G.; KOVAČEVIĆ, A.; SLADIĆ, G. Clean code tutoring: Makings of a foundation. In: **Proceedings of the 14th International Conference on Computer Supported Education**. [S.l.: s.n.], 2022. v. 1, p. 137–148.

MADURAPPERUMA, I.; SHAFANA, M.; SABANI, M. State-of-art frameworks for front-end and back-end web development. Faculty of Technology, South Eastern University of Sri Lanka, Sri Lanka, 2022.

MARTIN, R. C. **Design Principles and Design Patterns**. [S.l.]: Object Mentor, 2000.

MDN. **Manipulating documents**. 2024. Acessado em 1 de maio de 2024: <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents>.

MDN. **Rendering engine**. 2024. Acessado em 1 de maio de 2024: <https://developer.mozilla.org/en-US/docs/Glossary/Rendering_engine>.

META. **Legacy React Documentation**. Meta, 2023. Disponível em: <https://legacy.reactjs.org/>.

META. **Component State**. 2025. Acessado em 26 de fevereiro de 2025: <https://legacy.reactjs.org/docs/faq-state.html>.

META. **Composition vs Inheritance**. 2025. Acessado em 26 de fevereiro de 2025: <https://legacy.reactjs.org/docs/composition-vs-inheritance.html>.

META. **Conditional Rendering**. 2025. Acessado em 26 de fevereiro de 2025: <https://react.dev/learn/conditional-rendering>.

META. **Getting Started Create React App**. 2025. Acessado em 25 de fevereiro de 2025: <https://create-react-app.dev/docs/getting-started/>.

META. **Hook useRef**. 2025. Acessado em 26 de fevereiro de 2025: <https://react.dev/reference/react/useRef>.

META. **Passing Props to a Component**. 2025. Acessado em 26 de fevereiro de 2025: <https://react.dev/learn/passing-props-to-a-component>.

META. **State: A Component's Memory**. 2025. Acessado em 26 de fevereiro de 2025: <https://react.dev/learn/state-a-components-memory>.

META. **Virtual DOM and Internals**. 2025. Acessado em 26 de fevereiro de 2025: <https://legacy.reactjs.org/docs/faq-internals.html>.

META. **Writing Markup with JSX**. 2025. Acessado em 26 de fevereiro de 2025: <https://react.dev/learn/writing-markup-with-jsx>.

META. **Your First Component**. 2025. Acessado em 26 de fevereiro de 2025: <https://react.dev/learn/your-first-component>.

MOREIRA, R. A. F. *et al.* Design pattern detection tools: review-based comparison and survey studies. Universidade Federal de Minas Gerais, 2023.

MOZILLA. **AJAX**. 2023. Acessado em 11 de Novembro de 2023: <https://developer.mozilla.org/pt-BR/docs/conflicting/Web/Guide/AJAX>.

MOZILLA. **CSS**. 2025. Acessado em 25 de fevereiro de 2025: <https://developer.mozilla.org/pt-BR/docs/Web/CSS>.

MOZILLA. **Document Object Model (DOM)**. 2025. Acessado em 26 de fevereiro de 2025: <https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model>.

MOZILLA. **HTML: Linguagem de Marcação de Hipertexto**. 2025. Acessado em 25 de fevereiro de 2025: <https://developer.mozilla.org/pt-BR/docs/Web/HTML>.

MOZILLA. **import**. 2025. Acessado em 26 de fevereiro de 2025: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/export>.

MOZILLA. **import**. 2025. Acessado em 26 de fevereiro de 2025: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import>.

MOZILLA. **Javascript**. 2025. Acessado em 25 de fevereiro de 2025: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>.

MOZILLA. **MVC**. 2025. Acessado em 25 de fevereiro de 2025: <https://developer.mozilla.org/en-US/docs/Glossary/MVC>.

MOZILLA. **Package management basics**. 2025. Acessado em 25 de fevereiro de 2025: <https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Client-side_tools/Package_management>.

MOZILLA. **SPA (Single-page application)**. 2025. Acessado em 26 de fevereiro de 2025: <https://developer.mozilla.org/en-US/docs/Glossary/SPA>.

NODE. **The V8 JavaScript Engine**. 2024. Acessado em 1 de maio de 2024: <https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine>.

NPMJS. **About npm**. 2025. Acessado em 25 de fevereiro de 2025: <https://docs.npmjs.com/about-npm>.

OSMANI, A. **Learning JavaScript Design Patterns: A JavaScript and React Developer's Guide**. [S.l.]: "O'Reilly Media, Inc.", 2023.

PADOLSEY, J. **Clean Code in JavaScript: Develop reliable, maintainable, and robust JavaScript**. [S.l.]: Packt Publishing Ltd, 2020.

REACT. **React Documentation**. 2025. Acessado em 19 de Março de 2025: <https://react.dev/learn>.

REDHAT. **O que é API REST?** 2023. Acessado em 11 de Novembro de 2023: <https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api>.

REDUX. **Getting Started with Redux**. 2025. Acessado em 26 de fevereiro de 2025: <https://redux.js.org/introduction/getting-started>.

REFACTORING.GURU. **Design Patterns**. 2025. Acessado em 26 de fevereiro de 2025: <https://refactoring.guru/design-patterns>.

ROBBESTAD, S. A. **ReactJS blueprints**. [S.l.]: Packt Publishing, 2016.

Robert C. Martin. **Clean code: a handbook of agile software craftsmanship**. [S.l.]: Upper Saddle River, NJ etc., 2009.

Robert C. Martin. **Arquitetura Limpa**. [S.l.]: Alta Books, 2019.

ROUTER, R. **React Router**. 2023. Acessado em 26 de fevereiro de 2025: <https://reactrouter.com/home>.

SILVA, N.; RODRIGUES, E.; CONTE, T. A catalog of micro frontends anti-patterns. **arXiv preprint arXiv:2411.19472**, 2024.

SILVA, T. A. N. *et al.* Assessing the usage of new javascript features: a survey and mining study. Universidade Federal de Minas Gerais, 2022.

TOOLKIT, R. **Comparison with Other Tools**. 2025. Acessado em 26 de fevereiro de 2025: <https://redux-toolkit.js.org/rtk-query/comparison>.

VITE. **Vite Documentation**. 2023. Acessado em 6 de Junho de 2023: <https://vitejs.dev/>.

W3SCHOOLS. **w3schools react hooks**. 2023. Acessado em 8 de setembro de 2023: <https://www.w3schools.com/react/react_hooks.asp>.

W3SCHOOLS. **How TO - Make a Static Website**. 2024. Acessado em 1 de maio de 2024: <https://www.w3schools.com/howto/howto_website_static.asp>.

W3SCHOOLS. **CSS Tutorial**. 2025. Acessado em 25 de fevereiro de 2025: <https://www.w3schools.com/css/>.

W3SCHOOLS. **HTML Tutorial**. 2025. Acessado em 25 de fevereiro de 2025: <https://www.w3schools.com/html/>.

W3SCHOOLS. **JavaScript Modules**. 2025. Acessado em 26 de fevereiro de 2025: <https://www.w3schools.com/js/js_modules.asp>.

W3SCHOOLS. **JavaScript Tutorial**. 2025. Acessado em 25 de fevereiro de 2025: <https://www.w3schools.com/js/>.

W3SCHOOLS. **React Components**. 2025. Acessado em 26 de fevereiro de 2025: <https://www.w3schools.com/react/react_components.asp>.

W3SCHOOLS. **React Conditional Rendering**. 2025. Acessado em 26 de fevereiro de 2025: <https://www.w3schools.com/react/react_conditional_rendering.asp>.

W3SCHOOLS. **React JSX**. 2025. Acessado em 26 de fevereiro de 2025: <https://www.w3schools.com/react/react_jsx.asp>.

W3SCHOOLS. **Styling React Using CSS**. 2025. Acessado em 26 de fevereiro de 2025: <https://www.w3schools.com/react/react_css.asp>.

WIJENDRA, D. R.; HEWAGAMAGE, K. P. Analysis of cognitive complexity with cyclomatic complexity metric of software. **Int. J. Comput. Appl**, v. 174, p. 14–19, 2021.

WILSON, G.; ARULIAH, D. A.; BROWN, C. T.; HONG, N. P. C.; DAVIS, M.; GUY, R. T.; HADDOCK, S. H.; HUFF, K. D.; MITCHELL, I. M.; PLUMBLEY, M. D. *et al.* Best practices for scientific computing. **PLoS biology**, Public Library of Science San Francisco, USA, v. 12, n. 1, p. e1001745, 2014.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *et al.* **Experimentation in Software Engineering**. [S.l.]: Springer; Second Edition 2024, 2024. v. 236.

APÊNDICE A – PERGUNTAS TÉCNICAS SOBRE REACT, *CLEAN CODE* E *DESIGN PATTERNS*

Tabela 7 – Perguntas de entrevista

Tópico	Identificador	Pergunta
React	1.1	Em quais contextos você trabalhou no desenvolvimento de aplicações com React? [ ] Aplicações para iniciativa privada ou para os usuários finais [ ] Aplicações para órgãos públicos [ ] Aplicações acadêmicas [ ] Outros (Campo aberto)
	1.2	Quais outras bibliotecas/frameworks você utiliza em conjunto com React? [ ] Redux [ ] React Router [ ] Next.js [ ] Material-UI [ ] Axios [ ] Nenhuma [ ] Outros (Campo aberto)
Design Patterns	2.1	Quais *Design Patterns* você utiliza no desenvolvimento de aplicações com React? [ ] Singleton [ ] Abstract Factory [ ] Adapter [ ] Command [ ] Observer [ ] Template Method [ ] Iterator [ ] Não utilizo *Design Patterns* [ ] Outros (Campo aberto)
	2.2.1	Você já contribuiu com o desenvolvimento de alguma plataforma, framework, API, middlewares ou bibliotecas específicas para React?
	2.2.2	Se sim, quais? Você consegue identificar *Design Patterns* nelas?
	2.3	Quais as dificuldades e benefícios de usar *Design Patterns* em projetos com React?
Clean Code	3.1.1	Você usa ou já usou TypeScript nos projetos React?
	3.1.2	Se sim, o uso de TypeScript influencia positivamente na manutenibilidade do projeto?
	3.1.3	Se não, por que o TypeScript não influencia positivamente no Clean Code?
	3.2	Você usa boas práticas de desenvolvimento, como o SOLID em projetos React? Quais as lições aprendidas ao usá-las?
	3.3	Na sua opinião, quais boas práticas devem ser adotadas em projetos React ?

Fonte: Próprio autor

APÊNDICE B – CARACTERIZAÇÃO DO PERFIL PROFISSIONAL

Tabela 8 – Caracterização do perfil profissional

Identificador	Pergunta
1	Maior nível de Escolaridade [ ] Nível superior incompleto [ ] Nível Superior completo [ ] Mestrando [ ] Mestre [ ] Doutorando [ ] Doutor
2	Qual é o seu cargo atual ou os que você já atuou na área de programação ? [ ] Front-End [ ] Back-End [ ] Full Stack [ ] DevOps [ ] Mobile [ ] Full Cycle [ ] Outros (Campo aberto para adicionar outros papeis)
3	Tempo de experiência em atividades de desenvolvimento de aplicações com React. [ ] Mais de 5 anos [ ] Entre 2 e 5 anos [ ] Entre 1 e 2 anos [ ] Menos de um ano [ ] Não tenho experiência

Fonte: Próprio autor

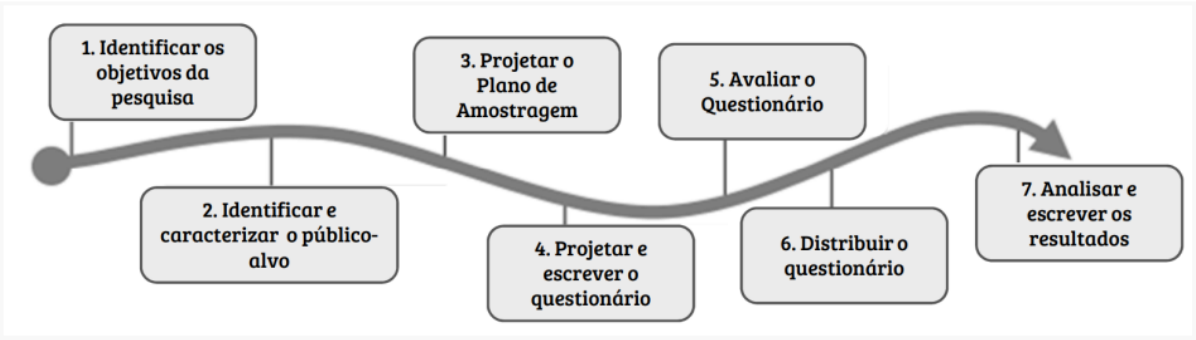
APÊNDICE C – PROTOCOLO GERAL

O protocolo geral do trabalho apresenta de forma concisa os fundamentos metodológicos e operacionais para investigar a aplicação de *design patterns* e *clean code* em projetos React. Ele está estruturado em duas abordagens complementares: um survey, destinado a coletar percepções e dados dos desenvolvedores *front-end*, e uma mineração de repositórios do GitHub, visando extrair e analisar práticas reais de desenvolvimento.

Entre os principais pontos, o protocolo define os objetivos (tanto gerais quanto específicos para cada abordagem), a população-alvo (desenvolvedores e repositórios populares), o plano de amostragem (detalhando critérios e tamanhos mínimos para participantes e repositórios) e as questões que serão aplicadas no survey, abrangendo desde a identificação do perfil profissional até perguntas abertas sobre desafios e benefícios das práticas adotadas.

Introdução

Esse documento contém as informações gerais para criação do survey sobre os *design patterns* e *clean code* usados por desenvolvedores que trabalham com desenvolvimento em React. Ademais, também tem como objetivo fazer uma mineração de padrões de projetos em repositórios do Github que utilizam o React.



1) Objetivos

a) Survey

Coletar percepções e informações diretas de desenvolvedores front-end que atuam com React para mapear o conhecimento, as dificuldades e os benefícios relacionados à aplicação de práticas de clean code e design patterns.

1.1.1 Objetivos específicos

1. Identificar o nível de conhecimento dos desenvolvedores sobre design patterns e boas práticas (por exemplo, o grau de familiaridade com princípios como SOLID, DRY, KISS, entre outros).
2. Investigar quais dificuldades e benefícios os profissionais encontram ao aplicar design patterns em projetos React – isto é, quais desafios são percebidos e quais ganhos em termos de legibilidade, manutenção e escalabilidade são experimentados.
3. Mapear o perfil profissional dos participantes (por meio de questões específicas sobre experiência, cargo e formação) para correlacionar as respostas com a adoção de práticas de qualidade no desenvolvimento.
4. Analisar como as respostas do survey se relacionam com os resultados obtidos na mineração, permitindo identificar convergências (ou divergências) entre o conhecimento teórico e a aplicação prática das técnicas.

1.2 Mineração

Realizar a extração e análise de dados de repositórios no GitHub (através de pull requests e issues) para identificar práticas e desafios na aplicação de clean code e design patterns em projetos React, complementando os dados coletados via survey.

1.2.1 Objetivos específicos

1. Selecionar e filtrar repositórios relevantes (por exemplo, os maiores repositórios de código aberto com a marcação React de grandes organizações) usando critérios como o número de estrelas e palavras-chave definidas no protocolo do estudo.
2. Extrair informações relacionadas a discussões e implementações – isto é, identificar quais pull requests e issues tratam de temas ligados à legibilidade, modularidade e manutenção do código, bem como os padrões de design aplicados.
3. Analisar os dados extraídos para mapear padrões comuns de práticas (como a utilização de componentes reutilizáveis, a aplicação seletiva de design patterns e a resolução de problemas como duplicação de código ou complexidade excessiva) e as dificuldades relatadas.
4. Validar os resultados da mineração com os dados do survey, permitindo uma análise integrada que corrobore ou complemente as percepções dos desenvolvedores.

2) População

a) Survey: Pesquisadores e Profissionais que trabalham ou já trabalharam com desenvolvimento de interfaces usando o React.
b) Mineração: Repositórios populares do github que usam o React.

3) Plano de Amostragem

a) Os participantes serão convidados por email para participar do survey e os repositórios serão procurados pesquisando pela popularidade e por tags no github.
b) O tamanho da amostra pode variar, mas o mínimo de participantes será de 20 participantes para o survey, e 3 repositórios para a mineração.
c) Será feita uma análise por amostragens das questões fechadas e das questões abertas; e individualizada das questões em aberto opinativas. Assim como, para a mineração será feita uma análise no repositório, como questões de commits, issues, stars, e também uma análise no código.

4) Questões para o Survey

a) Identificação do perfil profissional

i) Email (opcional)
ii) Maior nível de Escolaridade
    (1) Nível superior incompleto
    (2) Nível Superior completo
    (3) Mestrando
    (4) Mestre
    (5) Doutorando
    (6) Doutor
iii) Cargo do profissional
iv) Tempo de experiência em atividades de desenvolvimento de aplicações com React.
    (1) mais de 5 anos
    (2) entre 2 e 5 anos
    (3) entre 1 e 2 anos
    (4) menos de um ano
    (5) Não tenho experiência

b) Questões sobre tecnologia e desenvolvimento de aplicações com React

i) Em quais contextos você trabalhou no desenvolvimento de aplicações com React? [Múltipla escolha]
    (1) Aplicações para iniciativa privada ou para os usuários finais dos dispositivos
    (2) Aplicações para órgãos públicos
    (3) Aplicações acadêmicas
    (4) Outro
ii) Quais outras bibliotecas/frameworks você utiliza em conjunto com React? [Múltipla escolha]
    (1) Redux
    (2) React Router
    (3) Next.js
    (4) Material-UI
    (5) Axios
    (6) Outros
iii) Você já trabalhou com alguma plataforma, framework, API, middlewares ou bibliotecas de software específicas para aplicações com React?
    (1) Sim
    (2) Não

c) Questões Abertas sobre o desenvolvimento de aplicações com React

i) Com quais plataformas, frameworks, APIs e/ou bibliotecas de software específicas para aplicações com React você já trabalhou?

ii) Quais padrões de projetos você utiliza no desenvolvimento de aplicações com React?

iii) Quais as dificuldades e benefícios de seguir padrões de projetos em projetos com React?

iv) Quais as dificuldades e lições aprendidas ao desenvolver aplicações com React ligados a requisitos?

v) Quais as dificuldades e lições aprendidas ao desenvolver aplicações com React ligados a codificação?

vi) Quais as dificuldades e lições aprendidas ao desenvolver aplicações com React ligados a testes, validação e avaliação?

vii) Quais as dificuldades e lições aprendidas ao desenvolver aplicações com React ligados a pesquisa?

viii) Quais as vantagens de usar padrões de projetos no desenvolvimento de aplicações com React?