UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

MAYKON GLÊDSON DA SILVA NUNES

DETECÇÃO DE *CODE SMELLS* EM APLICAÇÕES REACT.JS COM TYPESCRIPT

QUIXADÁ

2023

MAYKON GLÊDSON DA SILVA NUNES

DETECÇÃO DE *CODE SMELLS* EM APLICAÇÕES REACT.JS COM TYPESCRIPT

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

Orientadora: Profa. Dra. Carla Ilane Moreira Bezerra.

QUIXADÁ

2023

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)

N926d Nunes, Maykon Glêdson da Silva.
Detecção de code smells em aplicações React.js com TypeScript / Maykon Glêdson da Silva Nunes. – 2023.
78 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Sistemas de Informação, Quixadá, 2023.
Orientação: Profa. Dra. Carla Ilane Moreira Bezerra.

1. React (JavaScript). 2. TypeScript. 3. Code smell. 4. Desenvolvimento web. I. Título.
CDD 005

MAYKON GLÊDSON DA SILVA NUNES

DETECÇÃO DE *CODE SMELLS* EM APLICAÇÕES REACT.JS COM TYPESCRIPT

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

Aprovada em: _____/_____/_____

BANCA EXAMINADORA

Profa. Dra. Carla Ilane Moreira
Bezerra (Orientadora)
Universidade Federal do Ceará (UFC)

Prof. Dr. Marco Túlio Valente
Universidade Federal de Minas Gerais (UFMG)

Prof. Dr. Fábio Ferreira
Instituto Federal Sudeste MG

Prof. Dr. Bruno Góis
Universidade Federal do Ceará (UFC)

Dedico este trabalho à minha família, que esteve comigo durante todos esses anos. Só cheguei até aqui graças ao apoio integral que recebo.

AGRADECIMENTOS

Primeiramente, agradeço à minha família por ter me apoiado durante todos esses anos e por terem feito o necessário para que eu conseguisse concluir mais esta etapa.

Aos meus amigos e amigas que estiveram comigo durante todos esses anos me apoiando, dividindo as preocupações e alegrias. As amizades construídas até aqui me ajudaram à seguir adiante na graduação.

Agradeço à Universidade Federal do Ceará — Campus de Quixadá, pela oportunidade da graduação, pelo conhecimento e pela excelente estrutura fornecida.

À professora Carla Ilane, que me orientou nesta pesquisa, colaborando desde o início com sugestões e correções para que eu conseguisse entregar este trabalho com excelência.

Por fim, agradeço à todas as pessoas que tiraram um tempo para colaborar nas entrevistas e respondendo os formulários das pesquisas realizadas neste trabalho.

"A necessidade por si só não é suficiente para libertar o poder: é preciso que haja conhecimento." (Ursula K. Le Guin)

RESUMO

Code smells são partes do código com potencial para o surgimento de problemas. Eles são gerados por procedimentos problemáticos adotados durante o *design* do software ou más práticas de programação. Ainda que os *code smells* tenham sido explorados em diversos trabalhos, eles são mais frequentemente voltados aos anti-padrões tradicionais da linguagem Java, tendo uma ausência de estudos destinados ao ecossistema JavaScript e as aplicações *web*. Nos últimos anos, o React tornou-se uma biblioteca popular no desenvolvimento de *interfaces web* e, mais recentemente, os desenvolvedores passaram a utilizar o TypeScript, um *superset* que adiciona tipagem estática ao JavaScript, juntamente com o React. O objetivo da presente pesquisa é identificar os *code smells* mais comuns no desenvolvimento de *interfaces* com a biblioteca React e a linguagem TypeScript e analisar como os desenvolvedores avaliam esses *smells* quanto a frequência e impacto negativo no código. Além disso, fornecer uma forma de identificar esses problemas através da ferramenta ReactSniffer. Primeiramente, aplicamos uma revisão da literatura cinzenta, buscando encontrar esses *smells*. Posteriormente, entrevistamos desenvolvedores e realizamos uma *survey* nas comunidades React. Depois estendemos o ReactSniffer para detectar os *smells* catalogados e o avaliamos com o Modelo de Aceitação Tecnológica (*Technology Acceptance Model* — TAM). Como resultado, identificamos 6 *code smells* na revisão da literatura cinzenta. Nas entrevistas e *survey*, identificamos que os *smells Any Type*, *Multiple Booleans for State* e *Non-Null Assertions* são os mais prejudiciais e frequentes. Todos os *smells* foram adicionados ao ReactSniffer e validamos a ferramenta com estudantes da UFC do Campus de Quixadá, onde observamos que a ferramenta é útil e fácil de se utilizar.

Palavras-chave: react (javascript); typescript; *code smell*; desenvolvimento web.

ABSTRACT

Code smells are parts of the code with the potential to lead to problems. They arise from problematic procedures adopted during software design or poor practices on programming. Although code smells have been explored in various studies, they are more frequently focused on the traditional anti-patterns of the Java language, with a lack of studies dedicated to the JavaScript ecosystem and web applications. In recent years, React has become a popular library in web interface development, and more recently, developers have started using TypeScript, a superset that adds static typing to JavaScript, in conjunction with React. This research aims to identify the most common smells in the development of interfaces using the React library and TypeScript language, and to analyze how developers assess these smells in terms of frequency and negative impact on the code. Additionally, we aim to provide a way to identify these issues through the ReactSniffer tool. Initially, we conducted a grey literature review to identify these smells. Subsequently, we interviewed developers and conducted a survey within the React communities. Then, we extended ReactSniffer to detect the cataloged smells and evaluated it using the Technology Acceptance Model (TAM). As a result, we identified 6 code smells in the grey literature review. In interviews and surveys, we found that the Any Type, Multiple Booleans for State, and Non-Null Assertions smells are the most detrimental and frequent. All the smells were added to ReactSniffer, and we validated the tool with students from UFC Campus Quixadá, where we observed that the tool is useful and easy to use.

Keywords: react (javascript); typescript; code smell; web development.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

1 INTRODUÇÃO

Com a ascensão dos *frameworks* JavaScript, a biblioteca React.js tornou-se uma notável e importante ferramenta no desenvolvimento de interfaces (NOVAC *et al.*, 2021). Para a produção de aplicações *front-end*, o JavaScript é uma das linguagens de programação mais difundidas, possuindo uma grande variedade de *frameworks* e bibliotecas. A comunidade ativa é um dos fatores cruciais na escolha dessas ferramentas, pois é ela que dá suporte aos desenvolvedores interessados em utilizá-las (PANO *et al.*, 2018).

Sistemas *web* apresentam soluções de natureza heterogênea, que por conta da demanda, manifestam necessidades de aumento da qualidade (BESSGHAIER *et al.*, 2021). Para fazer a manutenção de um software, os desenvolvedores realizam refatorações, ou seja, modificam o que está implementado sem alterar o seu comportamento (FOWLER, 2018). Antes de fazer uma refatoração, é necessário ler o código a fim de compreender o que está sendo executado. A partir daí, com uma ideia mais clara, os desenvolvedores podem colocar em prática a identificação de pontos de melhoria no código (FOWLER, 2018).

Os *code smells* são partes do algoritmo que têm potencial para o surgimento de um problema (FOWLER, 2018). Eles são gerados por procedimentos problemáticos adotados durante o *design* do software ou más práticas de programação (WALKER *et al.*, 2020). A refatoração de *code smells* é uma forma de melhorar a qualidade do software (MENSHAWY *et al.*, 2021). Ainda que os *code smells* tenham sido explorados em diversos trabalhos, eles são mais frequentemente voltados aos anti-padrões tradicionais da linguagem Java, tendo uma ausência de estudos destinados ao ecossistema JavaScript e as aplicações *web* (JOHANNES *et al.*, 2019; BESSGHAIER *et al.*, 2021). Com o objetivo de fornecer suporte aos desenvolvedores na detecção dos *code smells*, foram desenvolvidas diferentes ferramentas, sendo que a grande maioria dessas soluções foram produzidas para código Java (KAUR; DHIMAN, 2019).

Pela tipagem dinâmica, funções de primeira classe e herança baseada em *prototypes*, o JavaScript trouxe aspectos diferentes das linguagens orientadas a objetos. Por isso, ele apresenta alguns desafios para os softwares de detecção de *code smells* (ALMASHFI; LU, 2020). Fard e Mesbah (2013) e Johannes *et al.* (2019), buscaram detectar anti-padrões para a linguagem JavaScript e, (FERREIRA; VALENTE, 2023) identificaram *code smells* para o React.js e produziram uma ferramenta para identificação dos *smells* propostos. No entanto, nos últimos anos, os projetos passaram a utilizar o TypeScript, um *superset* que adiciona tipagem estática ao JavaScript (BOGNER; MERKEL, 2022). Portanto, nenhum dos trabalhos buscou

identificar *code smells* para a biblioteca React.js junto da linguagem de programação TypeScript.

Para construir sistemas com essa biblioteca, alguns desafios são encontrados, como a dificuldade para dar manutenção e organizar todos os módulos das aplicações, visto que os sistemas *web* estão se tornando cada vez mais complexos (FERREIRA; VALENTE, 2023). Por consequência, o TypeScript passou a ser incorporado e recomendado[1] para novos projetos que usam o React. Além disso, *frameworks* como o Angular, usam o TypeScript como linguagem principal[2]. Por conta disso, se faz necessária uma análise dos *code smells* que podem ser encontrados na construção das interfaces *web*, pois quanto mais anti-padrões a aplicação tiver, maiores serão os impedimentos (PALOMBA *et al.*, 2018a).

Nesse sentido, o objetivo da presente pesquisa é identificar e analisar os *code smells* mais comuns no desenvolvimento de interfaces com a biblioteca React e a linguagem TypeScript. Para isso, foi aplicada uma revisão da literatura cinzenta, buscando encontrar esses *smells*. Com isso, foram realizadas entrevistas estruturas com desenvolvedores e, a partir disso, os *smells* identificados foram descritos. Assim, a ferramenta ReactSniffer foi estendida para identificar os *code smells* listados neste trabalho. Por fim, a ferramenta foi validada com desenvolvedores.

Com isso, este trabalho é o primeiro que procura propor um catálogo de *code smells* para o React e TypeScript, que são tecnologias populares e que são importantes para o desenvolvimento de SPAs (*Single-Page Applications*). Essencialmente, os *smells* afetam princípios do *design* que podem ser vistos em qualquer linguagem ou *framework*. Portanto, a identificação desses problemas pode ajudar os desenvolvedores a detectar e refatorar esses *smells* em projetos *front-end*. Isso trará benefícios aos programadores, pois a remoção dessas anomalias ajuda na manutenção do código.

1.1 Questões de Pesquisa

Para a realização do trabalho, foram elaboradas questões de pesquisa que facilitam a coleta e análise dos dados:

QP1: Quais são os *code smells* mais comuns no desenvolvimento com React e TypeScript?

Espera-se fornecer aos desenvolvedores que utilizam o React com TypeScript um catálogo de *code smells* mais comuns no desenvolvimento com essas tecnologias. Para isso,

[1] https://react.dev/learn/typescript
[2] https://www.typescriptlang.org/docs/handbook/angular.html

realizamos uma revisão na literatura cinzenta e validamos a lista com as visões de programadores. Então, os desenvolvedores conseguirão identificar mais facilmente estas más práticas e, consequentemente, melhorarem a qualidade do *software*.

QP2: Como os desenvolvedores avaliam a frequência e o impacto dos *code smells* do React e TypeScript?

Com a lista de *code smells* identificados na literatura cinzenta, queremos avaliar, segundo o ponto de vista de desenvolvedores, este catálogo produzido. Com esse objetivo, efetuamos entrevistas com 5 desenvolvedores experientes no desenvolvimento de *interfaces* com React e uma *survey* com 30 respondentes.

QP3: Qual a percepção dos desenvolvedores sobre a utilidade, facilidade e intenção de uso do ReactSniffer?

Depois de registrados os *code smells*, estendemos a ferramenta ReactSniffer para que ela consiga detectar os anti-padrões deste trabalho. Com isso, foi possível utilizá-lo nas disciplinas de Manutenção de *Software* e Qualidade de *Software* do Campus da UFC em Quixadá. E então, avaliamos seguindo o Modelo de Aceitação Tecnológica (*Technology Acceptance Model* — TAM) para avaliar a utilidade percebida, facilidade de uso e intenção de uso futuro do ReactSniffer.

1.2 Objetivos

Esta seção é referente ao objetivo geral e específicos do presente trabalho.

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é identificar e analisar os *code smells* presentes no desenvolvimento de interfaces com a biblioteca React.js e a linguagem Typescript.

1.2.2 Objetivos Específicos

a) Catalogar os *code smells* da biblioteca React.js com a linguagem TypeScript.
b) Identificar o *smell* mais comum no desenvolvimento com React e TypeScript.
c) Estender a ferramenta ReactSniffer para detectar os *code smells* apresentados neste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, serão apresentadas as definições e os termos que norteiam este trabalho. A Seção 2.1 apresenta os principais conceitos da biblioteca React. Na Seção 2.2 estão alguns dos conceitos da linguagem TypeScript. A Seção 2.3 apresenta fundamentos relacionados aos *code smells* e a caracterização dos anti-padrões presentes na biblioteca React utilizando JavaScript. Na Seção 2.4 são apresentadas ferramentas utilizadas na identificação de *code smells*. Por fim, a Seção 2.5 introduz os principais conceitos relacionados às refatorações.

2.1 React

O React é uma biblioteca JavaScript de código aberto utilizada no desenvolvimento de interfaces *web*. O projeto é desenvolvido e mantido pelo Facebook e pela comunidade de desenvolvedores que colaboram com a biblioteca. Inicialmente, ele foi projetado para lidar com os problemas de desempenho das aplicações *web* (JAVEED, 2019). O *virtual DOM* desempenha um papel importante para a resolução desses problemas, pois, ele é uma representação em memória da *DOM (Document Object Model)* do navegador (JAVEED, 2019; PSAILA, 2008).

O React evita este recarregamento ao comparar a *virtual DOM* com a *DOM* do navegador (XING *et al.*, 2019). Dessa forma, ele identifica as alterações que devem ser aplicadas sem precisar recarregar toda a página (JAVEED, 2019).

2.1.1 JSX

O React usa os arquivos de extensão JSX para escrever o código JavaScript com HTML (SHARMA *et al.*, 2023). Em suma, o JSX adiciona XML ao JavaScript, em virtude disso, podem ser usadas *tags* e atributos para esses elementos (IVANOVA; GEORGIEV, 2019). O React não obriga o uso de JSX. No entanto, os desenvolvedores optam por utilizá-lo, pois torna o entendimento do código mais simples.

O React com JSX acopla a renderização com a lógica da interface, ao invés de separá-los em diferentes arquivos, permitindo que o código seja escrito de maneira declarativa. Para separar esses conceitos, o React usa componentes. Ao construir interfaces com TypeScript, os arquivos utilizam a extensão TSX, que é a extensão específica do TypeScript para um arquivo que contém JSX.

2.1.2 Componentes

Os componentes do React são blocos de código criados para construir partes da interface. Eles aceitam propriedades, nomeadas na biblioteca como *props*, para receber dados de outros componentes (KAUSHALYA; PERERA, 2021). A biblioteca renderiza os componentes e exibe o conteúdo do site na DOM usando o JavaScript (XING *et al.*, 2019). Um grande número de componentes pode resultar em um aumento da complexidade (BOERSMA; LUNGU, 2021). A reutilização ajuda a modularizar e aumenta a capacidade de manutenção da aplicação, além de reduzir a repetição de código (BAJAMMAL *et al.*, 2018).

No React, um componente pode ser uma classe ou uma função. Os componentes funcionais podem ser declarados como funções do JavaScript, enquanto os componentes de classe usam as classes da linguagem (FERREIRA; VALENTE, 2023). Em um componente do React, existe ainda um objeto que armazena dados mutáveis importantes para o componente, chamado de estado, o que permite que os componentes atualizem a renderização a partir dos dados alterados. Abaixo é apresentado um exemplo de componente React com TypeScript:

Código-fonte 1 – Exemplo de componente funcional React com TypeScript

```
1  interface UserProps {
2      name: string;
3  }
4
5  function User(props: UserProps) {
6      return (
7          <div><p>Nome: {props.name}</p></div>
8      );
9  }
10
11 class User extends Component<UserProps> {
12     render() {
13         return (
14             <div><p>Nome: {this.props.name}</p></div>
15         );
16     }
17 }
```

Fonte: Elaborado pelo autor.

No código-fonte 1, temos um exemplo simples de um componente funcional e de classe que renderiza em tela o nome do usuário recebido pelas *props* do componente. Foi definida uma interface para as *props* do componente usuário, determinando o tipo *string* para o nome.

2.2 TypeScript

TypeScript é uma linguagem de programação que estende o JavaScript, adicionando tipos estáticos à linguagem (BIERMAN *et al.*, 2014). A vantagem de utilizar tipos é identificar erros com mais rapidez e corrigi-los em uma IDE. Para que se possa estabelecer uma integração entre as duas linguagens, o TypeScript é convertido para JavaScript em um processo de compilação. (FELDTHAUS; MØLLER, 2014).

O JavaScript surgiu como uma linguagem para executar *scripts* que melhoravam a interação do usuário e tornava as interfaces *web* mais dinâmicas. Atualmente, o JavaScript deixou de ser apenas uma tecnologia exclusivamente da *web* e tornou-se multiplataforma, utilizado para casos muito mais complexos e em grandes sistemas. No entanto, as limitações da linguagem foram surgindo ao utilizá-la em aplicações de grande porte. O TypeScript veio para corrigir esses problemas (RASTOGI *et al.*, 2015). Um ponto relevante é que ele apresenta alguns recursos que atendem aos sistemas mais robustos, como a tipagem estática (BIERMAN *et al.*, 2014).

2.2.1 Tipagem estática

A tipagem estática, presente no TypeScript, quer especificar as características e controlar os valores e operações que podem ser executados com as partes do código. O Typescript garante a verificação dos tipos durante a compilação, o que significa que erros em tempo de execução podem ser evitados (BOGNER; MERKEL, 2022).

Código-fonte 2: Exemplo de tipagem estática

```
1 let letters: string = "abc";
2 typeof letters; // Retorna o valor string
3
4 letters = 123; // Erro de tipagem
```

Fonte: Elaborado pelo autor.

No código 2, foi criada a variável *letters*, atribuído o valor “abc” e definido o tipo *string* para ela, indicando que a variável só aceita valores desse tipo. Na linha seguinte, foi usado

o operador *typeof*, que retornou o tipo da variável *letters*. Como foi determinado o tipo *string* para a variável, será retornado o valor *string*. Após isso, foi feita a atribuição de um número para a variável *letters*, o que resulta em um erro no código, pois a variável recebe apenas valores do tipo *string*.

2.3 *Code Smells*

Os *code smells* são indícios de problemas que podem estar presentes em um código-fonte do sistema (FOWLER, 2018). O termo foi inicialmente introduzido por Fowler (2018), e são tratados como uma oportunidade de refatoração para a melhoria da qualidade do sistema. Eles são gerados a partir de más escolhas durante o *design* do *software* e na implementação, aumentando a complexidade e dificultando a manutenção. A ocorrência de anti-padrões costuma aumentar a probabilidade de falhas. Como também, a implantação de *smells* em um código dificultam a compreensão dos desenvolvedores, impedindo a evolução do sistema (PALOMBA *et al.*, 2018b).

Esses problemas no código representam violações de padrões de projeto e impactam negativamente a qualidade, afetando qualquer tipo de *software* (OIZUMI *et al.*, 2016). Uma das consequências dos *code smells* é a degradação da arquitetura de software, que causa o débito técnico e está ligado à pressa em cumprir os prazos do processo de desenvolvimento de *software*, onde os desenvolvedores ignoram o *design* e a arquitetura e, consequentemente, adicionam *smells* ao projeto (DAS *et al.*, 2022). Em geral, os desenvolvedores estão interessados em corrigir o código, pois são querem evoluir os sistemas (YAMASHITA; MOONEN, 2013).

Fowler (2018) propôs 22 *code smells* para linguagens de programação orientada a objetos, assim como orientações de formas para aplicar a refatoração destes anti-padrões. Com isso, outros trabalhos descreveram *smells* para uma variedade de linguagens. Fard e Mesbah (2013) definiu 13 *smells* para a linguagem JavaScript, como *Excessive global variables, Lazy object, Large object, Coupling JS/HTML/CSS, Close smell* e etc. No entanto, um *smell* como *Coupling JS/HTML/CSS* não pode ser aplicado em códigos da biblioteca React.js, pois acoplar código HTML em JavaScript é o principal fator da lógica de renderização do React. Ele utiliza funções para a separação dos conceitos e funcionalidades, chamadas de componentes.

Ferreira e Valente (2023) identificou 12 *code smells* para o React.js. A seguir, estão esses *smells* e suas definições.

– ***Direct DOM Manipulation:*** ocorre quando é feita uma operação direta na *DOM*, que é a

representação da árvore de elementos de uma interface *web*. O React.js possui a *virtual DOM*, a versão em memória da *DOM* real. Assim, realizar manipulações diretamente na *DOM* interfere no funcionamento da *virtual DOM*.

- ***Duplicated Component:*** quando os componentes apresentam a mesma estrutura de código ou representam o mesmo na interface. A repetição de código vai contra padrões de design já estabelecidos.
- ***Too Many Props***: as *props* são propriedades passadas para um componente. Esse anti-padrão ocorre quando um componente recebe e utiliza muitas *props*. Pode consumir recursos desnecessários, pois o componente é atualizado a cada mudança nas *props*.
- ***Prop Drilling***: manifesta-se sempre que um componente é utilizado como ponte para *props*. Ou seja, um componente mais alto da árvore de elementos é empregado para servir *props* para um componente mais baixo. Traz acoplamento desnecessário para o código, dificultando a manutenção.
- ***Uncontrolled Components***: esse *smell* ocorre quando o código permite que o estado do componente seja armazenado na *DOM* ao invés do React. Com isso, a biblioteca estará sincronizada com o componente.
- ***Force Update***: forçar através do código um recarregamento é um problema. A biblioteca possui um mecanismo para atualização dos componentes sem a necessidade de carregar novamente a página. Ao inserir isso no código, o desenvolvedor estará perdendo uma grande capacidade do React.
- ***Props in Initial State***: iniciar os estados com as *props* ocorre quando o componente recebe esses valores e cria um estado com eles. O efeito colateral disso é que as atualizações dos estados não serão renderizadas no componente em tela.
- ***JSX outside the render method***: ocorre quando o código JSX está fora do método de renderização do componente. Isso é um indicativo de que o componente tem muitas responsabilidades e isso o tornará difícil de ser reutilizado.
- ***Inheritance instead of Composition***: usar herança em componentes pode aumentar o acoplamento e dificultar a reutilização. Ao invés disso, a composição pode tornar o código mais flexível e o componente reutilizável em várias partes da interface.
- ***Large Component***: componentes que dificultam a legibilidade e a compreensão dos desenvolvedores devido ao grande número de linhas de código.
- ***Low Cohesion***: quando o componente possui muita responsabilidades. Isso dificulta a

manutenção, a compreensão e a reutilização do componente.

– ***Large File***: um arquivo que possui várias linhas de implementação e muitos componentes. Isso dificulta a manutenção e a modularidade, pois um componente pode estar localizado em um arquivo com muitos outros componentes.

O código 3 apresenta um exemplo do *code smell too many props*, que foi descrito acima:

Código-fonte 3: Exemplo de Too Many Props

```
1  function Component(props) {
2      return (
3          <div>
4              <div>{props.prop1}</div>
5              <div>{props.prop2}</div>
6              <div>{props.prop3}</div>
7              <div>{props.prop4}</div>
8              // Outras props
9          </div>
10     );
11 }
```

Fonte: Elaborado pelo autor.

Existem diversas maneiras de identificar esses *smells*, como: revisão de código pelos desenvolvedores, métricas e o uso de aplicações de detecção. As ferramentas são uma forma automatizada de identificar os *code smells* para assegurar a qualidade de software (HOZANO *et al.*, 2018). Muitas ferramentas foram produzidas para atender essas necessidades, algumas utilizando diferentes técnicas e métricas. A próxima seção descreve algumas das ferramentas construídas para a detecção de *code smells*.

2.4 Ferramentas para *Code Smells*

Muitos trabalhos construíram aplicações para a detecção de *code smells*, a maioria produzida para uma única linguagem, sendo o Java a mais explorada (KAUR; DHIMAN, 2019). A detecção desses *smells* pode ser realizada através da análise estática do código, na qual a ferramenta verifica componentes como objetos, classes, funções e outros elementos para encontrá-los (WALKER *et al.*, 2020). Existem diversas ferramentas criadas por empresas e

pesquisadores para diferentes linguagens atualmente. A seguir, estão algumas das ferramentas que detectam *code smells* do React.js, JavaScript ou TypeScript.

- **SonarQube**[1]**:** ferramenta de análise estática, com interface para visualização e configuração, identifica *smells* e vulnerabilidades no código em linguagens como Java, Python, JavaScript, TypeScript e outras. Além disso, apresenta sugestões de como refatorar as anomalias que ela identifica.
- **ReactSniffer**[2]**:** Ferreira e Valente (2023) propôs essa ferramenta para identificar os *code smells* catalogados do React.js. Ela identifica os *smells* presentes em arquivos JSX de projetos que usam JavaScript como linguagem. A ferramenta disponibiliza uma interface de linha de comando para execução e visualização.
- **JSNose**[3]**:** Fard e Mesbah (2013) desenvolveram uma ferramenta para detectar os 13 *smells* do JavaScript classificados no trabalho. Ela faz a análise estática e dinâmica para identificação das anomalias.
- **Embold**[4]**:** é outra ferramenta de análise estática que atende uma variedade de linguagens. Analisa possíveis problemas no código, vulnerabilidades e *code smells*. Pode ser instalado como *plugin* em uma IDE.

Ferramentas como o JSNose e o ReactSniffer, que realizam análise estática do código, geram a árvore sintática abstrata. Com ela, o detector pode passar pela árvore e identificar os *code smells* do código (ALMASHFI; LU, 2020). É possível gerar a árvore através do Babel[5], uma biblioteca que compila o JavaScript e que possui um analisador que percorre todo o código. (SARAFIM *et al.*, 2022). A ferramenta ReactSniffer gerou a árvore do código React.js usando o Babel.

2.5 Refatoração

Desenvolvedores podem aplicar técnicas para melhorar a estrutura do *software* e garantir a qualidade. As refatorações podem remover *code smells* do código (FOWLER, 2018). *Code smells* têm relação com refatorações, pois os *smells* trazem problemas de qualidade e a excelência é um ponto crucial da engenharia de *software*. O principal método para reduzir a ocorrência de *code smells* é a refatoração (LACERDA *et al.*, 2020). Antes de refatorar *code*

[1] https://www.sonarsource.com/
[2] https://www.npmjs.com/package/reactsniffer
[3] https://github.com/saltlab/JSNose
[4] https://embold.io/
[5] https://babeljs.io/

smells, é necessário que os desenvolvedores possuam experiência e treinamento (SANTOS *et al.*, 2019).

Para Lacerda *et al.* (2020) as refatorações em comparação com os *code smells* têm um maior impacto na facilidade de manutenção, modularidade, funcionalidades e na compreensibilidade do sistema. Sendo assim, é mais relevante aplicar a refatoração do que apenas identificar *smells*, com o objetivo de preservar o *software*. As etapas do processo de execução da refatoração são apresentadas por Mens e Tourwe (2004):

1. Identificar que parte do código é preciso refatorar.
2. Determinar qual a refatoração pode ser aplicada no local identificado.
3. Garantir que a refatoração preserve o comportamento.
4. Executar a refatoração.
5. Verificar o efeito da refatoração nos atributos de qualidade ou no processo de *software*.
6. Manter a consistência entre o programa refatorado e os artefatos de *software*.

Muitos estudos tem como foco principal a detecção de *code smells* e oportunidades de refatoração em projetos de código aberto da linguagem Java. Fowler (2018) definiu 22 técnicas para refatorar código Java, que são utilizadas por muitos trabalhos que estudam as refatorações (Al Dallal, 2015). Dessa forma, a identificação de *code smells*, aliada à aplicação da refatoração, podem auxiliar na garantia da qualidade, além de diminuir a complexidade dos projetos (LACERDA *et al.*, 2020).

As refatorações podem apresentar riscos que devem ser observados pelos desenvolvedores. Elas estão sujeitas à ocorrência de problemas que podem ter um impacto negativo no código e nas funcionalidades do *software*. Problemas como instabilidade do código, conflitos de *merge*, dificuldade para gerenciar o tempo gasto na refatoração e na revisão do código e introduzir mais mudanças do que o necessário (KIM *et al.*, 2014).

Além de identificar *smells*, os desenvolvedores podem utilizar métricas para identificar oportunidades de refatoração em bases de código. As métricas que avaliam a qualidade do código-fonte, como a facilidade de leitura e compreensão, têm um papel mais relevante em comparação com as métricas que avaliam as relações entre os componentes do código (PANTIUCHINA *et al.*, 2020). Dessa forma, há diferentes objetivos ao se realizar uma refatoração. Os desenvolvedores podem buscar realizar refatorações de *code smells*, assim como também refatorar para atender aos requisitos (SILVA *et al.*, 2016).

Durante o desenvolvimento de *software*, os desenvolvedores utilizam muitas fer-

ramentas. Elas podem automatizar o processo de refatoração ou simplificar o processo de manutenção, impedindo a introdução de problemas (PERUMA *et al.*, 2021). Os usuários dessas ferramentas desejam modificar o código para facilitar a implementação de novas funcionalidades ou melhorar o código existente, e não apenas para corrigir os erros que ocorrem (EILERTSEN; MURPHY, 2021).

Fontana *et al.* (2015) apresenta algumas ferramentas que auxiliam os desenvolvedores a reescreverem partes do código, mantendo o comportamento do *software*. Dentre elas, o JDeodorant[6], um *plugin* da IDE Eclipse que identifica *smells* e sugere técnicas de refatorações para serem aplicadas no local identificado para removê-los. Szoke *et al.* (2015) fornece um *plugin* para o Eclipse, NetBeans e Intellij, o FaultBuster, uma ferramenta que permite identificar partes problemáticas do código através da análise estática, realizar a refatoração automática da parte identificada e corrigir *code smells*.

Para Peruma *et al.* (2021) os problemas de usabilidade e a falta de confiabilidade nas ferramentas impedem que os desenvolvedores as utilizem adequadamente. Por conta disso, os programadores costumam buscar soluções em sites de perguntas e respostas como o Stack Overflow[7]. Outro problema está na dificuldade das ferramentas em se comunicar com os usuários, resultando em problemas para comunicar aos usuários as sugestões de forma clara e compreensíveis (EILERTSEN; MURPHY, 2021).

[6] https://marketplace.eclipse.org/content/jdeodorant
[7] https://stackoverflow.com/

3 TRABALHOS RELACIONADOS

Neste capítulo, serão apresentados os principais trabalhos identificados na literatura que se relacionam com esta pesquisa. A Seção 3.1 apresenta um estudo que identificou *code smells* para o React.js. Na Seção 3.2 um trabalho que identifica *code smells* para o JavaScript. A Seção 3.3 descreve um estudo da ocorrência dos *smells* em arquivos JavaScript. A Seção 3.4 detalha um trabalho que compara a qualidade de *software* de projetos que usam JavaScript e TypeScript. Por fim, a Seção 3.5 faz uma análise comparativa dos trabalhos relacionados com o presente trabalho.

3.1 *Detecting code smells in React-based Web apps*

Ferreira e Valente (2023) estudam os *code smells* introduzidos por más escolhas de *design* de *software* no React.js. Os autores foram motivados pelo baixo número de trabalhos produzidos para identificar *smells* dos *frameworks* JavaScript e foram os primeiros a propor *code smells* para a biblioteca. Para auxiliar os desenvolvedores, foram catalogados 12 anti-padrões do React.js e produziram uma ferramenta para identificar esses *smells*.

Inicialmente, realizaram uma revisão da literatura cinzenta sobre *code smells* relacionados ao React. Para assegurar a qualidade dos documentos selecionados, usaram as diretrizes propostas Garousi *et al.* (2019) para avaliar da qualidade das fontes. Depois disso, realizou-se uma análise temática para a extração e análise dos dados. Como resultado, foram identificados 12 *smells* para o React, divididos em 3 categorias.

Quadro 1: Code *smells* identificados para o React

Categorias	***Code smells***
Novos *smells*	*Force Update*
	Direct DOM Manipulation
	Props in Initial State
	Uncontrolled Components
	JSX Outside the Render Method
Smells tradicionais	*Inheritance Instead of Composition*
	Duplicated Component
	Too Many Props
	Large Component
	Large Files
	Low Cohesion
Smells parcialmente novos	*Prop Drilling*

Fonte: Ferreira e Valente (2023)

Além disso, foram realizadas entrevistas semi-estruturadas com os desenvolvedores, para identificar as más práticas observadas enquanto trabalham com React. As entrevistas ajudaram a identificar o *smell Large File* para o catálogo apresentado no Quadro 1. Posteriormente, os autores investigaram os *smells* mais frequentes nos projetos do React. Foi desenvolvida a ferramenta ReactSniffer, que identificou 2.565 *code smells* nos 10 projetos mais populares do GitHub que usam a biblioteca. A ferramenta foi disponibilizada como uma interface de linha de comando que auxilia os desenvolvedores a encontrarem os *smells* classificados no trabalho em projetos React.js.

Por fim, os autores buscaram encontrar a frequência de remoção dessas anomalias em projetos React com o ReactSniffer, que permitiu analisar os repositórios selecionados. Inicialmente, foram definidos três intervalos de seis meses cada, possibilitando a análise das mudanças ao longo do tempo. Após isso, os repositórios dos projetos foram baixados de acordo com cada intervalo de tempo e usaram a ferramenta para encontrar os *smells*. Em seguida, foram contabilizadas as anomalias identificadas na data de início do intervalo que foram removidas até a data de término.

3.2 *JSNose: Detecting JavaScript Code Smells*

Fard e Mesbah (2013) estudaram os *code smells* da linguagem JavaScript, onde foram listados 13 *smells*, sendo 7 deles adaptados dos anti-padrões tradicionais do Java e 6 identificados de outras fontes, como livros e documentações que apresentam padrões para o JavaScript. Além disso, apresentam um método para identificar os *smells* através de métricas, e adaptaram essa abordagem para uma ferramenta que realiza a análise estática e dinâmica para identificar os *smells* do trabalho.

Alguns *smells*, como *Empty catch*, *Large object*, *Lazy object*, *Long functions*, *Long parameter list*, *Switch statements* e *Unused/dead code*, foram retirados dos *smells* tradicionais e adaptados para o JavaScript. Além disso, os autores apresentam *smells* específicos para JavaScript, como *Closure smells*, *Coupling JavaScript/HTML/CSS*, *Excessive global variables*, *Long message chain*, *Nested callback* e *Refused Bequest*.

Os autores buscaram saber a efetividade do JSNose em detectar os *smells*. Para isso, compararam os resultados obtidos em uma inspeção manual com os produzidos pela ferramenta em 9 aplicações. Eles também procuraram encontrar o *smell* mais frequente nas aplicações *web*. Nesse sentido, foram contabilizados os *smells* identificados em 11 aplicações. Por fim, avaliaram

se existe uma correlação entre os *code smells* encontrados e as métricas definidas.

Os autores descrevem um mecanismo para identificar os *code smells* expostos no trabalho. Esse mecanismo usa métricas determinadas através da extração dos objetos, funções e relacionamentos. Após isso, é realizada uma análise estática e dinâmica, com o objetivo de compreender o comportamento do código durante a execução. Primeiramente, é necessário definir as métricas e os limites. Essas definições são utilizadas para identificar a presença dos *smells*.

Em seguida, é capturado o código-fonte para ser analisado e instrumentado, identificando partes do código que não estão sendo usadas e convertendo-o em uma árvore sintática abstrata no formato JSON, para que o analisador estático possa percorrê-la. Além disso, é feito o rastreamento da execução para a análise dinâmica. Por fim, com base nos dados coletados, os *code smells* são identificados.

3.3 *An Empirical Study of Code Smells in JavaScript Projects*

Saboury *et al.* (2017) estudam os *code smells* do JavaScript e os efeitos da ocorrência dessas anomalias. Foram detectados 12 *smells* e feita uma análise da sobrevivência deles nas bases de código de 5 projetos populares do JavaScript. Para isso, eles compararam o tempo decorrido até a ocorrência de uma falha em arquivos que contém *smells* e arquivos sem *smells*. Alguns *smells* como *Chained Methods, Long Parameter List* e *Long Method* foram retirados de Fard e Mesbah (2013). O restante dos *code smells* foram encontrados em guias de estilo do código JavaScript.

Os autores investigaram se o risco de falhas é maior em arquivos JavaScript com *smells* em comparação com arquivos sem *smells*. Posteriormente, estudaram se os arquivos JavaScript com *code smells* são igualmente propensos a falhas. Para investigar essas questões, foi realizada a identificação dos *smells* de cada arquivo dos *commits* dos cinco projetos selecionados. Para analisar esses *commits*, foi necessário verificar se eles corrigiam ou adicionavam *bugs* ou se corrigiam ou adicionavam *smells*.

O processo realizado para a extração dos dados inicia pelo clone local do repositório dos projetos selecionados. Posteriormente, com a API do GitHub, foi obtida uma lista de todos os *bugs* resolvidos nos repositórios e os arquivos que foram modificados. Para identificar os *code smells*, foi gerada a árvore sintática abstrata do código com o ESLint[1]. Com isso, também foi

[1] https://eslint.org/

possível utilizar o conjunto de métricas estabelecidas no trabalho.

Saboury *et al.* (2017) também realizaram uma pesquisa qualitativa com 1.484 desenvolvedores, que responderam 15 perguntas sobre os *code smells* do trabalho. A maioria deles tinha mais de 3 anos de experiência com o JavaScript. Ainda nessa pesquisa, os participantes especificaram, em uma escala Likert, de 1 a 10, cada um dos 12 *smells* do trabalho conforme o impacto deles em compreensibilidade, depuração e esforços de manutenção. Como resultado, *Nested Callback* foi considerado o *smell* mais prejudicial, seguido de *Variable Re-assign* e *Long Parameter List*.

3.4 *To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub*

Bogner e Merkel (2022) realizaram um estudo para avaliar se as aplicações que usam TypeScript apresentam uma melhor qualidade de *software* em relação às que usam JavaScript. Os autores apontam para uma discussão na comunidade de desenvolvimento de que as linguagens de tipagem dinâmica têm uma melhor qualidade de *software* em comparação com as de tipagem estática. Pela falta de estudos que suportem essa afirmação, eles produziram uma mineração de muitos repositórios de projetos de código aberto no GitHub.

Os autores querem saber se projetos que usam o TypeScript têm uma melhor qualidade de *software* do que os que usam JavaScript. Além disso, investigaram se os projetos TypeScript que usam menos o tipo *any* possuem uma melhor qualidade de *software*. Para responder ambas as questões, é necessário realizar a mineração dos repositórios no GitHub.

Os autores selecionaram algumas características, para analisar a qualidade de *software* e responder às questões. A qualidade do código foi uma das selecionadas, sendo os *code smells* um indicador da baixa qualidade do código. Outro ponto considerado foi a compreensibilidade do código, para entender grau de compreensão dos desenvolvedores sobre o código. Os autores usaram a ferramenta SonarQube para identificar esses problemas. Eles ainda olharam para a frequência de *bugs*, através da *API* do GitHub, obtendo o histórico de *commits* dos projetos. Posteriormente, foi analisado o tempo necessário para a remoção de um *bug* nesses projetos, utilizando os dados obtidos através das *issues* do GitHub.

Por fim, os autores definiram 8 hipóteses para cada questão de pesquisa e testaram cada uma desas hipóteses. As hipóteses da primeira questão foram fundamentadas na discussão de que a qualidade do software desenvolvido com TypeScript é superior à do JavaScript. Para

as outras hipóteses, assumiram que utilizar menos o tipo *any* melhora a qualidade de *software*. Como resultado, eles validaram as hipóteses de que aplicações TypeScript possuem menos *smells* e são mais compreensíveis do que as JavaScript. Como também, validaram a hipótese de que o tipo *any* afeta negativamente o tempo de correção de problemas, qualidade e compreensibilidade do código.

3.5 Análise comparativa

Este trabalho, assim como Ferreira e Valente (2023), Fard e Mesbah (2013) e Saboury *et al.* (2017) tem como objetivo identificar *code smells*. Entretanto, diferente dos estudos apresentados, o presente trabalho busca identificar *smells* em código React com a linguagem TypeScript. Saboury *et al.* (2017) buscou encontrar novos *smells* apenas em guias de estilo do JavaScript, enquanto este trabalho irá fazer uma revisão da literatura cinzenta para identificar *smells*, assim como foi feito em Ferreira e Valente (2023).

Bogner e Merkel (2022) mostra questões relacionadas à qualidade dos sistemas que utilizam o TypeScript, e apresenta problemas em utilizar o tipo *any*. No presente trabalho, o intuito é apresentar outros problemas e anti-padrões que podem ser encontrados em projetos com TypeScript e React.

Fard e Mesbah (2013) apresenta *code smells* para o JavaScript na versão do ECMAScript 5, lançada em 2009. Ferreira e Valente (2023) identifica *code smells* para projetos React que utilizam o JavaScript. Dessa forma, é possível notar que este estudo difere dos citados, uma vez que utiliza uma linguagem de programação diferente para identificar novos *smells* do React. O Quadro 2 compara este trabalho com os outros trabalhos apresentados.

Quadro 2: Quadro comparativo entre os trabalhos relacionados e este trabalho

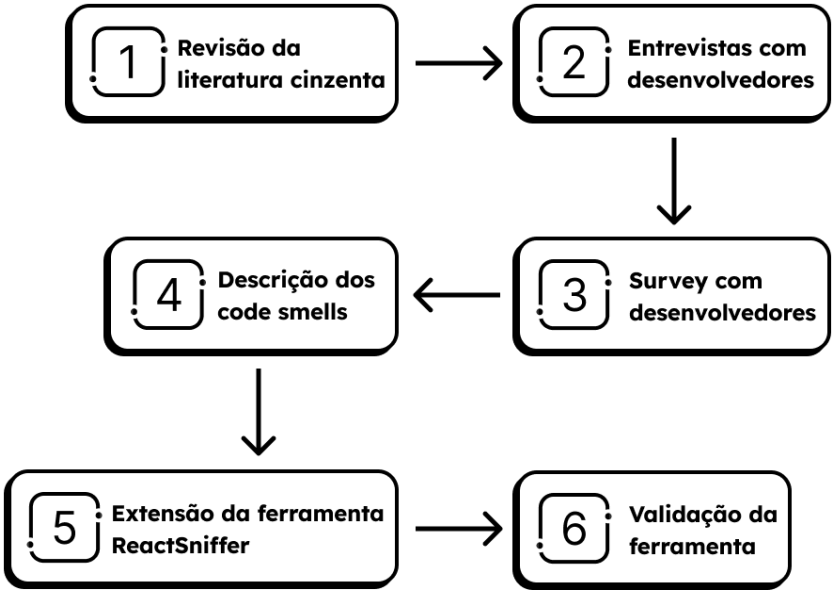
	Ferreira e Valente (2023)	Fard e Mesbah (2013)	Saboury *et al.* (2017)	Bogner e Merkel (2022)	Este trabalho
Identifica *code smells*	X	X	X		X
Utiliza TypeScript				X	X
Automatiza a detecção de *code smells*	X	X			X

Fonte: Elaborado pelo autor.

4 METODOLOGIA

Este capítulo apresenta os procedimentos metodológicos usados neste trabalho. Para alcançar os objetivos propostos foi aplicada uma metodologia inspirada no trabalho de Ferreira e Valente (2023). Como diferencial, serão adicionados novos *smells* à ferramenta ReactSniffer e a mesma será validada aplicando um método de avaliação. As próximas seções descrevem as etapas que serão seguidas e a Figura 1 ilustra o sequenciamento entre essas etapas.

Figura 1: Metodologia da pesquisa



Fonte: Elaborada pelo autor.

4.1 Revisão da literatura cinzenta

A literatura cinzenta oferece uma visão prática e baseada em experiências, usando como fonte blogs, vídeos, documentações, fóruns, etc. Ela pode fornecer informações importantes sobre tópicos de pesquisa emergentes na Engenharia de Software, pois ela inclui conhecimentos que não estão presentes na literatura acadêmica e que podem ser encontrados no mercado de software (GAROUSI *et al.*, 2019).

Para encontrar os *code smells* do React com TypeScript, é necessário realizar uma revisão da literatura cinzenta devido à falta de trabalhos que utilizam essas tecnologias. Pela comunidade ativa e a quantidade de desenvolvedores que usam essas tecnologias, a literatura cinzenta pode capturar contribuições valiosas desses desenvolvedores e dessas comunidades. Para isso, essa revisão será executada de acordo com as diretrizes propostas no trabalho de

Garousi *et al.* (2019).

Primeiramente, será feita uma pesquisa em um mecanismo de busca, neste trabalho será o Google. Após isso, serão selecionados, de acordo com a ordem retornada pelo algoritmo de classificação de páginas do Google, os resultados que tratam de *smells* do React com TypeScript. Por fim, os *smells* serão extraídos e validados.

4.1.1 Pesquisa no Google

O mecanismo de busca do Google será utilizado para encontrar alguns materiais de literatura cinzenta, como blogs, documentos e discussões em fóruns. Para isso, será necessário criar uma *query* de busca com alguns termos relacionados à *code smells* e que consiga listar os materiais que estejam relacionados aos *smells* do React com TypeScript.

Para garantir uma grande quantidade de resultados, foi definida a *query* com os termos em inglês. Nas duas primeiras *strings* foram adicionadas as duas tecnologias principais da pesquisa, o React e o TypeScript, seguidos de variações do termo *code smells*. As seguintes *strings* foram selecionadas: *"code smell", "bad smell", "bad pratice"* e *"antipattern"*. Dessa forma, foi definida a seguinte *query*:

> *("react" OR "reactjs") AND ("typescript") AND ("code smell" OR "bad smell" OR "antipattern" OR "bad practice")*

Foram retornados 138.000 resultados pelo mecanismo de busca. Garousi *et al.* (2019) define alguns critérios de parada, dentre eles, incluir apenas os N principais resultados. Neste trabalho, os resultados serão analisados até a 10ª página retornada e continuará apenas se os resultados da última página ainda revelarem resultados relevantes. Na seção seguinte, serão descritos os parâmetros para a seleção dos resultados que podem indicar *smells* candidatos.

4.1.2 Seleção dos resultados

Pelas características da literatura cinzenta, é necessário realizar uma seleção com base em critérios de qualidade. Após obter os resultados do Google, será possível avaliar quais desses resultados podem ser relevantes. Neste trabalho, serão selecionados aqueles que podem apresentar *code smells* do React e TypeScript. Inicialmente, devem ser definidos critérios de seleção para as fontes resultantes e, posteriormente, realiza-se o processo de seleção.

Garousi *et al.* (2019) aborda a avaliação da qualidade das fontes, definindo uma lista de verificação para os resultados obtidos na pesquisa do Google. Nesta etapa, é necessário adequar os itens da lista para que eles se adaptem a esta pesquisa. Para cada resultado obtido, será aplicada a verificação de acordo com os critérios definidos no Quadro 3 e serão selecionadas as fontes que atendem pelo menos duas questões desses critérios. As questões tratam da experiência dos autores e a objetividade do texto.

Quadro 3: Lista de verificação da qualidade

Critério	Questões
Domínio do autor	Publicou outros trabalhos na área?
	Possui experiência na área?
Objetividade	O texto é objetivo e com informações verificáveis? Ou é uma opinião subjetiva?
	Possui exemplos para os *smells*?

Fonte: Elaborado pelo autor

4.1.3 Extração e síntese dos dados

Após selecionar os resultados que podem estar relacionados com os *code smells* do React com TypeScript, será feita a leitura e análise de cada uma das fontes, para a extração e síntese dos *smells*. De acordo com Garousi *et al.* (2019) é comum que na literatura cinzenta, para manter as informações sucintas, uma maior explicação seja ignorada pelos autores, enquanto os dados originais estão em outras fontes.

Primeiramente, é importante desenvolver uma planilha de rastreabilidade, para extrair os dados de forma eficiente. Nela devem estar os *links* rastreáveis para cada fonte selecionada, assim como os critérios para a seleção da fonte conforme a lista de verificação da qualidade aplicada na etapa anterior. Durante a extração dos dados, é necessária uma revisão com foco em encontrar e registrar cada um desses *smells*.

Será aplicada a análise temática, que auxiliará na identificação de padrões, ou seja, a apresentação de um determinado *code smell* nos resultados obtidos. Nesta pequisa, é imprescindível registrar dados qualitativos dos resultados obtidos pelo mecanismo de busca. Eles serão importantes na síntese dos dados, pois, após extrair os dados, será possível verificar a ocorrência dos *code smells* em outras fontes. Ao final dessa etapa, será necessário validar esses *smells*.

4.1.4 Validação dos dados

Por fim, após realizar a extração e síntese dos dados, fazemos a validação dos *smells* encontrados. Nesta etapa buscamos definir quais dos *code smells* candidatos podem ser aplicados para React e TypeScript. Para isso, analisaremos estes *code smells* e excluiremos aqueles que não se configuram como anti-padrões.

O uso de bibliotecas externas, guias de estilo e código limpo não se configuram como *code smells*. A compreensão e a interpretação são importantes nessa etapa, com o conhecimento das tecnologias React e TypeScript, podemos evitar falsos positivos.

4.2 Entrevistas com desenvolvedores

Depois de identificar os *code smells*, serão feitas entrevistas semi-estruturadas com desenvolvedores que atuam no mercado de desenvolvimento de *software* e que utilizam as tecnologias React e TypeScript. As entrevistas serão aplicadas para entender a percepção e as opiniões dos desenvolvedores em relação aos *smells* encontrados neste trabalho e validá-los. Como também, será possível saber se os desenvolvedores conseguem encontrar outros *smells* nos projetos em que atuam.

Dessa forma, as entrevistas semi-estruturadas não só auxiliarão na descrição dos *smells*, como também poderão identificar novos *code smells* para compor o catálogo proposto neste trabalho. Para isso, perguntamos aos participantes quais eram as principais más práticas e causadores de problemas nos projetos React que eles trabalham, a fim de coletar as informações sobre a ocorrência, as observações e outros *code smells*. Após isso, apresentamos exemplos dos anti-padrões identificados na revisão da literatura cinzenta e perguntamos aos desenvolvedores se os *code smells* catalogados impactam negativamente a manutenção, legibilidade e evolução do código. Como também, questionamos a ocorrência desses *smells* nos projetos que eles desenvolvem. O Quadro 4 lista as perguntas feitas para os participantes.

Quadro 4: Perguntas da entrevista

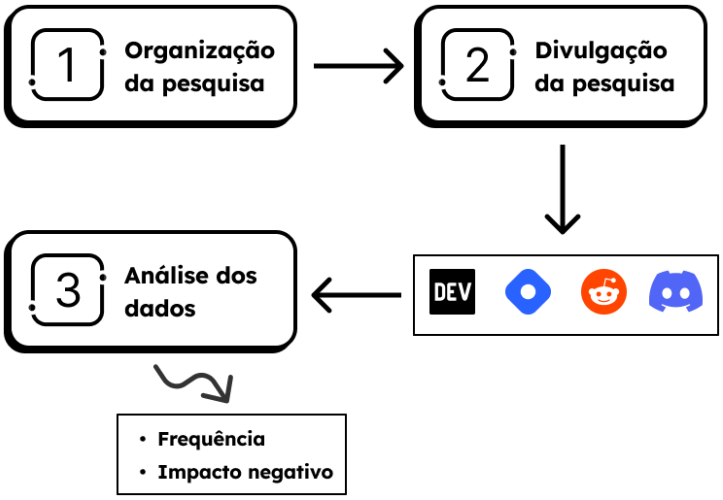
Perguntas
Quanto tempo de experiência você tem com o React?
Quantos projetos em que você participou ou contribuiu utilizavam React com TypeScript?
Quais são as principais más práticas que você observou ao trabalhar com React?
Esses *smells* impactam negativamente na manutenção, legibilidade e evolução do código?
Com que frequência você encontrou esse tipo de problema nos projetos em que trabalhou?

Fonte: Elaborado pelo autor

4.3 *Survey* com desenvolvedores

Além das entrevistas, realizamos um *survey* com profissionais que trabalham com o React e TypeScript para complementar a análise do impacto e frequência dos anti-padrões. A Figura 2 ilustra as etapas de elaboração, divulgação e análise dos resultados.

Figura 2: Visão geral do *survey*



Fonte: Elaborada pelo autor.

4.3.1 Organização do survey

Inicialmente, estruturamos o formulário explicando quais são os objetivos deste trabalho, esclarecendo para o público-alvo o processo, proteção das informações e participação voluntária. As questões do questionário estão disponíveis no apêndice B. Optamos por dispor as questões da seguinte maneira:

- Perfil: Nas três primeiras perguntas, questionamos a localização geográfica do participante, os anos de experiência com React e o número de projetos em que ele trabalhou/contribuiu.
- Frequência dos *smells*: Adicionamos um *link* de um Gist[1] do GitHub, com o catálogo de *code smells* e um detalhamento para cada um deles. Com isso, perguntamos aos participantes qual o grau de frequência com que cada um dos *smells* ocorrem. Utilizamos uma escala *Likert* de 5 pontos que vai de muito raro até sempre.
- Impacto negativo dos *smells*: Depois de avaliar a frequência, os participantes avaliaram o impacto negativo de cada *code smell*. Como na seção anteriormente, utilizamos uma escala Likert de 5 pontos que vai de muito baixo até muito alto.

[1] https://gist.github.com/maykongsn/5bf2f206fbd40b087dc8d33d603d4be0

4.3.2 Divulgação do survey

O formulário foi transmitido através de comunidades de tecnologia, algumas específicas do React e redes sociais do autor deste trabalho. De início, convidamos os desenvolvedores React em canais relacionados ao React listados no site da biblioteca[2]. Então, compartilhamos a pesquisa no Reddit, DEV, Hashnode, comunidades no Discord e Reddit.

4.4 Descrição dos *code smells* identificados

A etapa seguinte, após validar e ter à disposição os *code smells* identificados, é descrever cada um deles e como ocorrem de forma detalhada e apresentar os exemplos de código. Como também, serão acrescentados os comentários dos participantes da entrevista, com a frequência observada por eles. Dessa forma, ficará estabelecido o catálogo com os *code smells* deste trabalho.

Para descrever de forma mais aprofundada os *smells*, é necessário compreender como eles ocorrem, o que será possível a partir das fontes selecionadas, pois algumas delas podem fornecer uma definição para cada um dos anti-padrões apresentados. Além disso, o conhecimento e experiência dos participantes da entrevista permitirão descrever os *code smells* e esclarecer possíveis dúvidas do autor desta pesquisa. Adicionalmente, a documentação do React e do TypeScript são fontes de informação sobre essas tecnologias e podem auxiliar no processo de elaboração.

4.5 Extensão da ferramenta ReactSniffer

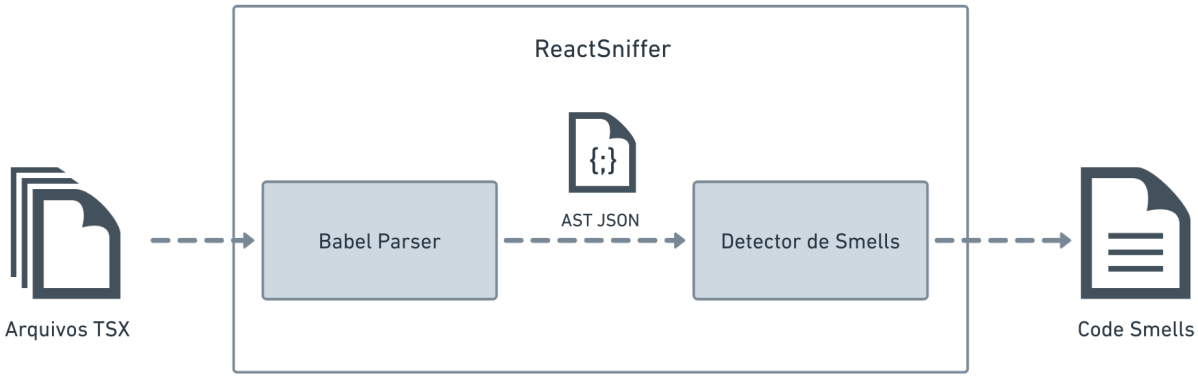
A ferramenta ReactSniffer será estendida para detectar de forma automática os *code smells* do React com TypeScript. Na versão atual, o ReactSniffer consegue detectar os anti-padrões propostos por Ferreira e Valente (2023). O catálogo proposto no presente trabalho oferece uma oportunidade de ampliação da ferramenta, para auxiliar os desenvolvedores a removerem outros *smells* durante o desenvolvimento de interfaces.

[2] https://react.dev/community

4.5.1 Arquitetura

Assim como Ferreira e Valente (2023), este trabalho precisará implementar o analisador de código com o *parser* do Babel. Como também, deverá implementar o módulo de detecção dos *smells*. O *parser* gera a árvore sintática abstrata do código TSX e o módulo de detecção identifica os *code smells*. A análise do código é feita após a geração da árvore, pois, com ela, o analisador irá identificar os *code smells*. A Figura 3 ilustra a arquitetura e o funcionamento da ferramenta.

Figura 3: Arquitetura do ReactSniffer



Fonte: Elaborada pelo autor.

Dessa forma, serão adicionados os *smells* deste trabalho e adaptaremos o ReactSniffer para que o *parser* gere a árvore do código em um JSON e o detector de *smells* recursivamente percorra a árvore e execute as estruturas condicionais para identificar os *code smells* nos componentes do React.

4.6 Validação da ferramenta

Após adicionar os *smells* à ferramenta, é possível aplicar um método de avaliação do ReactSniffer com desenvolvedores. Para este trabalho, foi selecionado o Modelo de Aceitação Tecnológica (*Technology Acceptance Model* — TAM) (DAVIS, 1989), que avalia a aceitação de uma tecnologia com base na utilidade percebida e a facilidade de uso.

Desenvolvemos sentenças para que os participantes avaliem a utilidade percebida, facilidade de uso e intenção de uso futuro do ReactSniffer. Os avaliadores terão que indicar em uma escala *Likert* o nível de concordância com as sentenças. Para isso, os desenvolvedores poderão fazer a utilização da ferramenta durante a aplicação desse método de avaliação.

5 REVISÃO DA LITERATURA CINZA SOBRE *CODE SMELLS* PARA *REACT TYPESCRIPT*

Nesta seção, serão apresentados os resultados obtidos a partir da revisão da literatura cinzenta, seguindo as etapas descritas na metodologia, pesquisa sobre a relevância dos *smells*, descrição dos anti-padrões, extensão da ferramenta e validação da ferramenta. Inicialmente, foi feita uma pesquisa no Google, que resultou em fontes para encontrar os *smells* candidatos listados nessa etapa.

5.1 Seleção dos resultados

A pesquisa com a *query* de busca no Google gerou 138.000 resultados. Devido ao grande número de dados, Garousi *et al.* (2019) aponta que a extração de resultados é influenciada pela queda na qualidade das evidências. Por isso, é necessário definir uma regra de parada. Durante a pesquisa, a qualidade dos resultados foi diminuindo após avançar pelas páginas retornadas pelo buscador.

Avaliamos cada um dos resultados obtidos com a busca. Com isso, foi possível verificar se os resultados se adequavam aos objetivos deste trabalho. Selecionamos para a próxima etapa as fontes que atendessem a pelo menos duas das questões da lista de verificação. Como anteriormente mencionado, seguiríamos a seleção caso os resultados da 10ª página retornada revelasse evidências relevantes. No entanto, nessa página, as fontes que recebemos não atendiam aos critérios de qualidade estabelecidos.

Nessa etapa, adicionamos na planilha de rastreabilidade[1] os *links* das fontes selecionadas, assim como os critérios para a seleção da fonte. Dessa forma, foram obtidas 41 fontes para a extração dos dados na próxima etapa e estão listadas na Tabela 2.

5.2 Extração e síntese dos dados

Após selecionar as fontes de evidências, foi feita a leitura e análise de cada uma delas, para a extração e síntese dos *smells* que podem estar relacionados ao React e TypeScript. Aplicando a análise temática, verificamos então se o conteúdo da evidência selecionada tratava de *smells* das tecnologias deste trabalho. Verificando se continham uma explicação para o *code smell* apresentado ou um exemplo.

[1] https://github.com/maykongsn/react-typescript-code-smells/blob/main/grey-literature/revisao-da-literatura-cinzenta.md

Das fontes selecionadas anteriormente, 26 delas apresentavam algum *code smell* candidato. O restante abordavam a introdução de anti-padrões na biblioteca mas sem citar como eles ocorrem. Outros discutiam temas que não estavam relacionados com *smells*. Como por exemplo, artigos de blog fazendo comparativos do React com outros *frameworks*.

Ao final dessa etapa, obtivemos os 28 *smells* candidatos, listados no Quadro 5 e que foram validados posteriormente para formar a lista final de *smells*. Alguns destes ocorreram em diversas fontes.

Quadro 5: *Code smells* candidatos identificados na literatura cinzenta

	Code Smells	Fontes
1	Children prop without explicitly type	F1
2	Many Non-Null Assertions	F3, F7, F20
3	Not use any type in TypeScript	F3, F10, F22, F26, F34
4	Type aliases should be used	F3, F11
5	Magic Strings and Numbers	F22, F14
6	Enum implicit values	F12
7	Prop-types in TSX	F21, F25
8	Value as SomeType	F34, F19
9	Rely on Generated Types	F35
10	Multiple booleans for state	F27
11	Props Plowing	F4
12	Component Nesting	F4
13	Heavy Work	F4
14	Messy Events	F4
15	Coupled State	F4, F5
16	Non-stable Identities in React Context Provider	F8
17	Rendering Non-Boolean Condition Values	F9
18	Destroy and Recreate	F23
19	Index as key	F24
20	Props Spreading	F25, F38
21	Too Many useState	F27
22	Large useEffect	F27
23	Unnecessary useMemo/useCallback on props	F30
25	Optional Properties	F34
25	One-letter Generics	F34
26	Modify State Directly	F37
27	Navigate to Go Back	F37
28	Barrel Files	F41

Fonte: Elaborado pelo autor.

5.3 Validação dos dados

Nessa etapa, analisamos como ocorrem cada um dos *smells* candidatos, seja por trechos de código presentes nas evidências ou a explicação em texto. No geral, as evidências resultantes da etapa anterior estavam mais relacionadas com nomenclatura de variáveis, guias

de estilo, uso de bibliotecas, etc. Alguns *smells* estavam relacionados somente ao React e ao desenvolvimento de interfaces com essa biblioteca, que já foram catalogados no trabalho de Ferreira e Valente (2023) e, portanto, não foram selecionados.

Alguns *smells*, foram removidos da lista final, como *Magic Strings and Numbers*, que é um *smell* tradicional e já está presente em outros trabalhos. Outro *smell*, *Value as SomeType*, também não foi selecionado, por conta da semelhança com o *code smell Any Type*. A abordagem problemática seria aplicar o *type assertion as any* e para isso, selecionamos o *code smell Any Type*. A nomenclatura de *Children prop without explicitly type* e *Type aliases should be used* foram alteradas para nomes mais descritivos: *Props with Children Pitfall* e *Missing Union Type Abstraction*, respectivamente.

Depois de identificar os *code smells*, eles foram analisados e classificados em duas categorias. Os quatro primeiros *smells* são específicos da linguagem TypeScript. Esses anti-padrões estão presentes na linguagem e afetam o código de projetos React, como a utilização de tipagem *any*, *enums* com valores implícitos e outros. A segunda categoria é composta por dois *code smells* e está relacionada às implementações de elementos do React com código TypeScript. Ou seja, os *smells* envolvem conceitos de ambas as tecnologias. Assim, os anti-padrões identificados estão ligados a problemas de excesso de *states* do tipo *boolean* e *children props* sem a tipagem correta.

Com isso, identificamos 6 *smells* do React com TypeScript. O Quadro abaixo apresenta os *smells* identificados.

Quadro 6: *Code smells* identificados

Categoria	***Code smell***	**Ocorrências**
TypeScript	*Any Type*	5
	Non-Null Assertions	3
	Missing Union Type Abstraction	2
	Enum Implicit Values	1
React e TypeScript	*Multiple Booleans for State*	3
	Props With Children Pitfall	1

Fonte: Elaborado pelo autor

5.4 Descrição dos *Code Smells* identificados

Nesta seção apresentamos o catálogo de *code smells* identificados neste estudo. Discutiremos sobre os anti-padrões comumente relatados em conteúdos sobre React e TypeScript.

Estes resultados foram suportados pela etapa de revisão da literatura cinzenta, onde obtivemos exemplos e descrições para o entendimento dos *smells*. Portanto, nessa seção, para cada *smell* daremos uma descrição detalhada e um exemplo de código.

5.4.1 Any Type

O TypeScript permite a verificação e inferência de tipos para variáveis, objetos, funções, etc. Recomenda-se fortemente na literatura cinzenta que os desenvolvedores aproveitem dos benefícios que o TypeScript oferece. Ao definir o tipo *any* para algum dos elementos do código, o desenvolvedor está desativando a checagem de tipos, perdendo a vantagem do *type safety* que a linguagem provê, possibilitando a manipulação dessas entidades sem qualquer verificação. Isso pode ser uma forma de manter o código JavaScript existente. No entanto, é importante que o desenvolvedor adicione os tipos corretos de forma gradual.

Por exemplo, considere o código-fonte 4, onde as *props* de um componente foram definidas como *any*. Nesse caso, as informações sobre os tipos das *props* que estão sendo recebidas serão perdidas, uma vez que o TypeScript aceitará qualquer tipo para as *props*. Consequentemente, sem a verificação, é possível que erros em tempo de execução aconteçam.

Código-fonte 4: Exemplo do *smell Any Type*

```
1 function Component(props: any) {
2     return(
3         <div>
4             <AnotherComponent>{...props}</AnotherComponent>
5         </div>
6     )
7 }
```

Fonte: Elaborado pelo autor.

O *type safety* ajuda o desenvolvedor React a detectar erros em tempo de desenvolvimento, dado que o TypeScript realizará uma verificação de tipos durante a compilação do código, trazendo um pouco mais de confiabilidade contra erros na execução e auxiliando na manutenção do código. Adicionalmente, quando especificamos os tipos corretos, estamos adicionamos uma documentação em um componente React.

No código 5, ilustramos a vantagem de se utilizar os tipos corretos, adicionando uma *interface* com as *props* do componente e definindo o tipo das *props* como *ComponentProps*.

Dessa forma, o TypeScript verificará se estão sendo passadas para *Component* as propriedades esperadas e do tipo correto. Além disso, o fornecimento de informações dos tipos de dados para o TypeScript facilita a manutenção e refatoração do código.

Código-fonte 5: Exemplo de código *type safety*

```
1  interface ComponentProps {
2      text: string;
3  }
4
5  function Component(props: ComponentProps) {
6      const { text } = props;
7      return(
8          <div>
9              <AnotherComponent text={props.text} />
10         </div>
11     )
12 }
```

Fonte: Elaborado pelo autor.

5.4.2 Non-Null Assertions

O operador de *non-null assertion* informa que uma propriedade não será *null* ou *undefined* para o sistema de tipos da linguagem. A utilização desse operador pode ser um indício da falta de tratamento para valores nulos no código, comprometendo a segurança de tipos e causando erros em tempo de execução.

No exemplo de código 6, o componente *Profile* possui um estado que é atualizado por um *useEffect*, carregando os dados desse componente. No entanto, ao desenvolvermos a renderização do componente, o TypeScript irá gerar um erro, informando que *user.email* pode ser *undefined*. Por esse motivo, os desenvolvedores podem optar por uma solução rápida, que seria utilizar o operador *non-null assertion* na linha 10, informando para o TypeScript que *user* não será *undefined*. Isso introduz o risco para erros de *runtime* se *user* for *undefined* quando acessado. Por essa razão a documentação do TypeScript recomenda usá-lo com cuidado:

Similarmente a outros *type assertions*, isso não altera o comportamento em tempo de execução do seu código, então é importante usar o ! apenas quando vocês sabe que o valor não pode ser *null* ou *undefined*.

Código-fonte 6: Exemplo do *smell Non-Null Assertions*

```
const Profile = () => {
    const [user, setUser] = useState<User>();

    useEffect(() => {
        // Capturando dados
    }, []);

    return (
        <div>
            <p>{user!.email}</p>
        </div>
    );
};
```

Fonte: Elaborado pelo autor.

5.4.3 Missing Union Type Abstraction

Os *union types* são uma maneira de combinar dois ou mais tipos, com eles podemos criar elementos que podem ser de um desses tipos. Em um código React podemos ter a repetição do mesmo *union type* em vários lugares. Por conta disso, podemos utilizar dos *type aliases* do TypeScript para abstrair os *union types* de um componente como no código 7.

Código-fonte 7: Exemplo de *type aliases*

```
type NullableString = string | null;
```

Fonte: Elaborado pelo autor.

Os *type alises* do TypeScript são similares às interfaces e podem ser uma maneira de aumentar a compreensão do código. Um dos ganhos em utilizá-los é a facilidade em reutilizar *union types* em diferentes lugares, sem a necessidade de indicar os tipos várias vezes.

No código 8, temos um componente com os estados *data* e *status* que podem ser do tipo *string* ou *null*. A repetição dos *union types* pode requerer alterações em diversos pontos, consumindo tempo e aumentando a probabilidade de inconsistência. Portanto, é recomendável utilizar *type aliases* para centralizar a definição de tipos em um local e evitar a repetição destes *union types*.

Código-fonte 8: Exemplo do *smell Missing Union Type Abstraction*

```
1 const Info = () => {
2     const [userName, setUserName] = useState<string | null>(null);
3     const [status, setStatus] = useState<string | null>(null);
4     // Outros estados com esse union type
5 };
```

Fonte: Elaborado pelo autor.

5.4.4 Enum Implicit Values

Os *enums* são uma forma de definir constantes numéricas e de texto. Eles são úteis para representar valores relacionados, facilitando a manutenção e melhorando a legibilidade de um código. No entanto, no TypeScript, é comum que os desenvolvedores usem *enums* sem definir um valor explícitos para eles. Sendo assim, o TypeScript define valores numéricos para eles seguindo a ordem.

Código-fonte 9: Exemplo de *Enum*

```
1 enum Status {
2   Pending,
3   Processing,
4   Completed,
5 }
```

Fonte: Elaborado pelo autor.

No exemplo de código 9, as constantes *Pending*, *Processing* e *Completed* estão sem valores constantes. Com isso, o TypeScript define valores numéricos seguindo a sequência dos itens começando de 0.

Loading = 0, Success = 1, Error = 2, seguindo a sequência.

Por conta disso, caso o desenvolvedor adicione novos itens ao *enum*, alterando a ordem dos itens, os valores numéricos serão alterados pelo TypeScript, podendo provocando erros em tempo de execução caso algo dependa desses valores. Portanto, é importante atribuir constantes aos membros do *enum*. Dessa forma, evitamos estes problemas causados pela mudança na ordem dos itens.

Código-fonte 10: Exemplo do *smell Enum Implicit Values*

```
1  enum Status {
2    Loading,
3    Success,
4    Error,
5  }
6
7  const Component = () => {
8      const [status, setStatus] = useState<Status>(Status.Loading);
9
10     // Restante do componente
11 }
```

Fonte: Elaborado pelo autor.

5.4.5 Multiple Booleans for State

Os *boolean states* podem ser utilizadas na renderização condicional, exibição de componentes e muitos outros casos. Nesses casos, esses estados são utilizados para determinar qual o estado atual de um componente baseado em um valor *boolean*. O código abaixo exemplifica o uso de um *useState* para renderização condicional, onde uma mensagem é exibida dependendo do valor *true* ou *false* do estado *showMessage*.

Código-fonte 11: Exemplo do *smell Multiple Booleans for State*

```
1  const Component = () => {
2      const [isVisible, setIsVisible] = useState(false);
3
4      return (
```

```
5           <div>
6               <button onClick={() => setIsVisible(!isVisible)}>Exibir/
                    Ocultar</button>
7               {isVisible && <p>Hello World</p>}
8           </div>
9       );
10  }
```

Fonte: Elaborado pelo autor.

Portanto, é comum que os desenvolvedores React usem *states* do tipo *boolean* para diferentes aspectos do componente. E por isso, o componente acaba dependendo de muitas variáveis de estado. No exemplo a seguir, o componente possui alguns *useState* do tipo *boolean*. Um aumento do número desses estados pode tornar a manutenção e organização desse código desafiadora.

Código-fonte 12: Exemplo do *smell Multiple Booleans for State*

```
1  function Component() {
2      const [isActive, setIsActive] = useState(false);
3      const [isLoading, setIsLoading] = useState(false);
4      const [isVisible, setIsVisible] = useState(false);
5      // Outros estados booleanos
6  }
```

Fonte: Elaborado pelo autor.

5.4.6 Props With Children Pitfall

Escrevendo uma expressão JSX com *tags* de abertura e fechamento, o conteúdo passado entre elas é chamado de *children*. Ela pode ser de qualquer tipo, ou seja, outro elemento JSX, um componente, *string*, etc. Neste exemplo, temos um componente que recebe uma *children* como *prop*. O componente *AnotherComponent* passa a *children* que *Component* receberá.

Código-fonte 13: Exemplo de um componente com *children*

```
1  function Component({ children }) {
2      return (
3          <div>{children}</div>
4      );
```

```
5  }
6
7  function AnotherComponent() {
8      return (
9          <Component>
10             <p>Content</p>
11         </Component>
12     );
13 }
```

Fonte: Elaborado pelo autor.

Se o componente tem *children props*, não é uma boa prática definir *any* ou tipos que podem restringir os elementos JSX como: *undefined*, *null*, *unknown* e *never*. A falta de segurança de tipos é o problema de se utilizá-los. Como explicado anteriormente, ao definir o tipo da *children* como *any*, o desenvolvedor estará desativando a verificação de tipos e aceitando qualquer tipo de dado. Neste exemplo, a utilização de *never* representa um *children* que nunca ocorre.

Código-fonte 14: Exemplo do *smell Children Props Pitfall*

```
1  type ComponentProps = {
2    children: never;
3  };
4
5  function Component({ children }: ComponentProps) {
6      return (
7          <div>
8              {children}
9          </div>
10     );
11 }
```

Fonte: Elaborado pelo autor.

6 ENTREVISTA E *SURVEY* COM DESENVOLVEDORES

Neste capítulo, apresentamos os resultados obtidos a partir das entrevistas e pesquisa com desenvolvedores. Na Seção 6.2 analisamos os resultados obtidos após realizar as entrevistas. Por fim, a Seção 6.3 apresenta os resultados de um *survey* com desenvolvedores.

6.1 Introdução

Para validar o catálogo de *code smells* para o React e TypeScript apresentado na Seção 5.4, conduzimos entrevistas com desenvolvedores experientes e realizamos um *survey* nas comunidades do React e de desenvolvimento *web*.

Visamos aprofundar o conhecimento dos *code smells* e seus efeitos no React. Por isso, desenvolvemos a seguinte questão de pesquisa: ***QP2.*** *Como os desenvolvedores avaliam a frequência e o impacto dos code smells do React e TypeScript?* Com essa questão, procuramos compreender a opinião dos desenvolvedores em relação aos *code smells* listados, pois, até este momento, só temos o ponto de vista dos resultados da revisão da literatura cinzenta.

6.2 Entrevista com desenvolvedores

Entrevistamos desenvolvedores *front-end* com experiência e que trabalham atualmente com React. O objetivo desta etapa de entrevistas é aprofundar o conhecimento sobre os *code smells* catalogados e identificar outras práticas que prejudicam a qualidade do código dos projetos TypeScript com a biblioteca React.

6.2.1 Seleção dos participantes da entrevista

Convidamos 5 programadores com experiência de mercado e que são da rede de contatos do autor deste trabalho. As informações sobre cada desenvolvedor podem ser encontradas no Quadro 7. O participante 1 tem 4 anos de experiência com a tecnologia, tendo participado de grandes projetos que usam React e TypeScript. O participante 2 está a 3 anos desenvolvendo sistemas profissionalmente com React em uma organização com soluções utilizadas por grandes empresas internacionais. O terceiro participante tem 4 anos de experiência com desenvolvimento *web*, participando de diversos projetos, de diferentes mercados com a biblioteca React. O participante 4 está a 2 anos trabalhando com React.

Quadro 7: Participantes das entrevistas

Participante	Setor	Experiência
P1	Serviços de seguros	4 anos
P2	Alimentos	3 anos
P3	Serviços financeiros	4 anos
P4	Serviços financeiros	2 anos
P5	Consultoria de tecnologia	3 anos

Fonte: Elaborado pelo autor

6.2.2 Análise dos resultados das entrevistas

Como descrito na Seção 4.2, questionamos os desenvolvedores sobre as más práticas que eles identificaram ao trabalhar com React. Em seguida, apresentamos os *code smells* identificados na revisão da literatura cinzenta e discutimos sobre a frequência e os impactos negativos desses *smells* na manutenção, legibilidade e evolução do código.

Todos os participantes expressaram a opinião de que os desenvolvedores não utilizam adequadamente o TypeScript quando recorrem ao *any*, prejudicando a manutenção do código:

> *Um código com tipos fracos pode dificultar a compreensão daqueles que realizarem a manutenção e introduzir erros.*

Os participantes P2 e P3 mencionaram que algo que acontece com frequência é a utilização de *any* quando estão trabalhando com bibliotecas de terceiros, o entrevistado P3 apontou o seguinte:

> *Para lidar com algumas bibliotecas que possuem um sistema de tipos fraco, precisei utilizar o any. Algumas bibliotecas não são desenvolvidas usando TypeScript ou utilizam muitos tipos any na implementação. E nem sempre consigo ter tempo de criar uma tipagem para os dados dessas bibliotecas.*

Do mesmo modo, P2 considera a falta de abstração de dados como uma má prática e citou uma situação que vivenciou em um projeto:

> *Já me deparei com o uso de Record de string para any para representar um objeto de usuário, em vez de optar por uma abstração por meio de uma interface ou classe.*

No geral, os desenvolvedores entrevistados consideram os *code smells* catalogados neste trabalho como prejudiciais à qualidade de um sistema. P2, P3 e P4 pensam que a utilização do operador *non-null assertion* deve ser evitado ao extremo, por conta do risco de erros em tempo de execução. Além disso, P3 vê a necessidade de refatorar *Missing Union Type Abstraction* e deu uma sugestão de refatoração:

> *Se você vai usar esse union type em vários lugares, separe uma interface global para que esses valores fiquem acessíveis para diferentes lugares.*

Ele também reconhece a relevância de refatorar o *Multiple Booleans for State*, conforme o que foi citado na entrevista:

> *Podemos criar um único estado que represente os três possíveis estados do componente. Isso diminui o número de re-renderizações.*

Referente a frequência, segundo o relato de todos os entrevistados, o *smell Any Type* é significativamente o mais comum. Depois dele vem o anti-padrão de *Multiple Booleans for State*, citado pelos desenvolvedores P1, P2 e P3 como muito incidente. *Non-Null Assertions* é comum apenas para o P3 e todos os entrevistados preferem não recorrer ao operador.

Enum Implicit Values e *Props with Children Pitfall* são considerados por todos os desenvolvedores como os que menos ocorrem. Segundo eles, não é comum utilizar *enums* e definir os tipos listados como inadequados para a *children* nos projetos em que trabalham. O participante P3 comentou sobre como ele lida com enumerações:

> *Utilizo objetos constantes em vez dos enums. Com isso eu consigo ter valores imutáveis.*

Após as entrevistas, não retiramos nem adicionamos novos *smells* à lista deste trabalho. Os entrevistados consideraram os *smells* catalogados como adequados para as más práticas do React com TypeScript e deram mais detalhes sobre o motivo pelo qual eles são problemáticos.

6.3 *Survey* com desenvolvedores

Conduzimos uma *survey* com desenvolvedores React. Nosso objetivo é utilizar desta abordagem para compreender ainda mais o impacto negativo e a frequência dos *code smells* catalogados. Além disso, identificar os *smells* que são mais recorrentes e prejudiciais ao código TypeScript do React. As questões do formulário estão no Apêndice B.

6.3.1 Perfil dos participantes do survey

Depois de compartilhada com desenvolvedores, concluímos a pesquisa com 30 respostas. No que diz respeito a experiência, 43,3% dos respondentes possuem entre 1 e 3 anos de experiência, 26,7% têm menos de 1 ano de experiência, 26,7% possuem entre 3 e 5 anos, enquanto apenas 3,4% acumulam mais de 5 anos de experiência com o React.

Em termos de quantidade de projetos que os respondentes participaram, aproximadamente 47% dos desenvolvedores trabalharam em 6 ou mais projetos, 30% contribuíram entre 1 a 2 projetos, enquanto 23% colaboraram entre 3 e 5 projetos React.

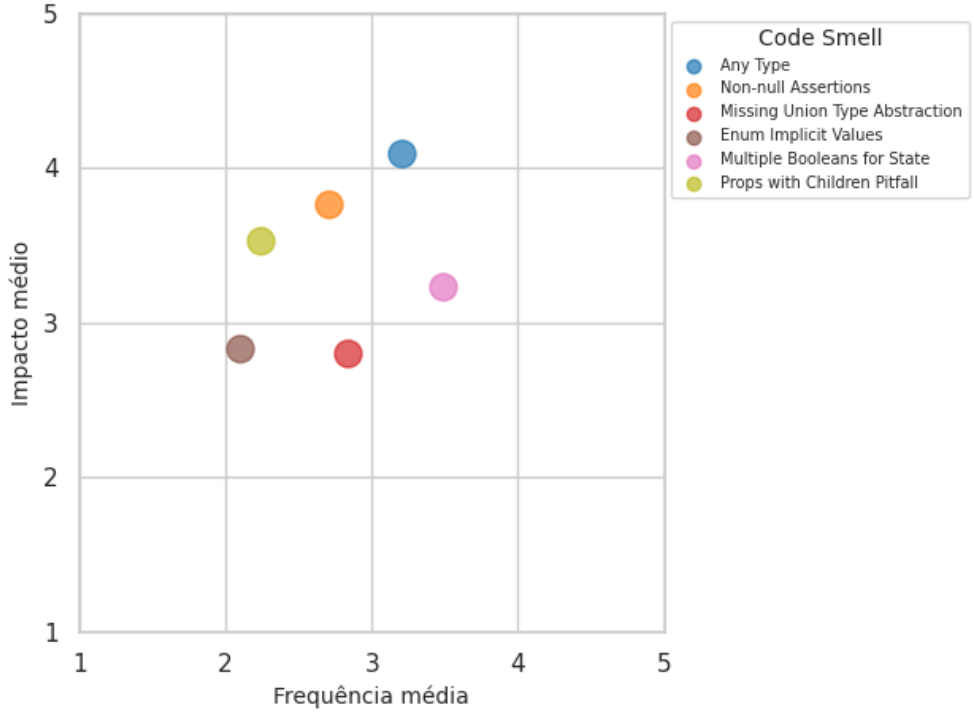
6.3.2 Análise dos resultados da pesquisa

Utilizando um gráfico de dispersão, como vemos na Figura 4, apresentamos uma representação da avaliação realizada por desenvolvedores utilizando a escala Likert de 5 pontos para a frequência e impacto negativo dos *code smells* identificados. Na variável do eixo y do gráfico temos a média do impacto negativo dos *smells*, enquanto no eixo x temos a média da frequência.

Os pontos do gráfico representam a média da frequência e impacto para os *code smells* e estão divididos em cores diferentes. Percebemos que alguns dos anti-padrões estão mais acima do eixo vertical, indicando que os participantes avaliaram esse *smell* como mais frequente nos projetos React. Como também, constatamos que alguns dos pontos mais avançados do eixo horizontal sugerem uma frequência maior, segundo os respondentes.

Analisando o gráfico, podemos ver que os pontos estão concentrados na faixa entre 2.8 e 4.1 do eixo do impacto negativo. Esse agrupamento indica que os *smells* apresentam impactos moderados e altos, segundo a avaliação dos desenvolvedores. Percebemos que o *code smell Any Type* é considerado mais prejudicial, seguido de *Non-Null Assertions* e *Props with Children Pitfall*.

Figura 4: Avaliação dos desenvolvedores sobre os *code smells*



Fonte: Elaborada pelo autor.

Quanto à frequência, vemos que ela variou de 2.1 até 3.4, indicando uma distribuição de ocorrência entre moderada e ocasional. Isso sugere que os *code smells* não são extremamente comuns para os desenvolvedores, mas eventualmente ocorrem. *Multiple Booleans for State* é considerado o mais comum, seguido do *Any Type* e *Missing Union Type Abstraction*.

Com frequência e impacto equilibrados, nossa análise baseada nos intervalos das médias sugere uma tendência central para os *code smells* que estão, em geral, concentrados em valores medianos, sem serem altamente frequentes, mas também não são raros, e o impacto negativo não é extremamente baixo nem muito alto.

Para identificar aqueles com maior frequência e impacto negativo, podemos calcular a média aritmética de cada um dos *smells*. Essa abordagem ajuda a encontrar um ponto de equilíbrio entre a frequência e o impacto. Com isso, os anti-padrões *Any Type* (3.65), *Multiple Booleans for State* (3.35) e *Non-Null Assertions* (3.23) apresentam as médias mais altas do conjunto de *code smells*. Isso indica que esses anti-padrões podem ser os mais críticos e prioritários para serem corrigidos, necessitando de uma maior atenção da parte dos desenvolvedores, já que ocorrem com frequência e afetam negativamente o código.

As respostas de cada participante da pesquisa e o Jupyter Notebook com o gráfico de distribuição, impacto, frequência e média de cada *smell* estão disponíveis no GitHub[1].

[1] https://github.com/maykongsn/react-typescript-code-smells/tree/main/survey

7 EXTENSÃO E VALIDAÇÃO DA FERRAMENTA REACTSNIFFER

Neste capítulo, descrevemos os passos realizados para estender e validar a ferramenta ReactSniffer. Na Seção 7.1 detalhamos as alterações feitas no ReactSniffer para detecta os novos *code smells*. Na Seção 7.2.1 apresentamos as sentenças que foram elaboradas para o questionário de validação. A Seção 7.2.2 apresenta o perfil dos avaliadores da ferramenta. Por fim, na Seção 7.2.3 apresentamos os resultados da validação.

7.1 Extensão da ferramenta ReactSniffer

A ferramenta está disponível publicamente no GitHub[1] e pode ser instalada com um gerenciador de pacotes como o NPM. O *fork* com as alterações aplicadas está disponível também no GitHub[2]. Neste trabalho, seguimos a mesma arquitetura do trabalho de Ferreira e Valente (2023), inserindo novas verificações para identificar os *smells*.

Conforme mencionado na Seção 4.5.1, o ReactSniffer possui um módulo de detecção dos *smells*. Adicionamos condicionais para cada um dos anti-padrões catalogados neste trabalho. O pseudocódigo abaixo descreve como foi implementada a identificação na ferramenta.

Código-fonte 15: Algoritmo de detecção dos *smells*

```
1  detectSmells(node)
2      smells = []
3      if(node->type == "TSAnyKeyword") then
4          smells.insert(ANY)
5      if(node->type == "TSNonNullExpression") then
6          smells.insert(NNA)
7      if(node->type == "TSUnionType" AND component[MUT] > N_Unions)
           then
8          smells.insert(MUT)
9      if(node->type == "TSEnumMember" AND node->initializer ==
           undefined) then
10         smells.insert(EIV)
11     if(node->callee == "useState" AND node->arguments ==
12     "BooleanLiteral") then
13         smells.insert(MB)
```

[1] https://github.com/fabiosferreira/reactsniffer
[2] https://github.com/maykongsn/reactsniffer

```
14      if(node->type == "TSAnyKeyword" OR node->type == "TSNeverKeyword"
            OR node->type == "TSUndefinedKeyword" OR node->type == "
        TSNullKeyword" OR node->type == "TSUnknownKeyword") then
15          smells.insert(PCP)
16      return smells
```

Fonte: Elaborado pelo autor.

A seguir tratamos dos *code smells* que foram adicionados à ferramenta e o que foi incrementado no módulo de detecção.

Utilizando o *parser* para gerar os nós da árvore abstrata sintática, adicionamos uma condicional para nós *TSAnyKeyword* da árvore para o *code smell Any Type*. Dessa forma, o tipo *any* pode ser identificado em variáveis, funções ou outros contextos em que este tipo pode ser usado. Quando encontrado, pegamos a linha onde ocorre o *smell* e a acrescentamos na saída.

Código-fonte 16: Verificação do smell *Any Type*

```
1 if (value.type == "TSAnyKeyword") {
2     // Captura da linha do smell
3 }
```

Fonte: Elaborado pelo autor.

Para *Non-Null Assertions* a estratégia é a mesma, buscando por nós do operador de *non-null assertion*. Para isso, adicionamos uma condicional para procurar pelo nó *TSNonNullExpression*. Com isso, quando o código analisado utiliza o operador, é possível encontrá-lo e retornar a linha de código correspondente.

Código-fonte 17: Verificação do smell *Non-Null Assertions*

```
1 if(value.type == "TSNonNullExpression") {
2     // Captura da linha do smell
3 }
```

Fonte: Elaborado pelo autor.

Missing Union Type Abstraction é detectado a partir da busca por anotações de tipo com *TSTypeAnnotation* que tenham associado à chave *type* o valor *TSUnionType*. Nós contamos o número de *union types* encontrados no componente. Se o valor for maior do que o limite estabelecido, retornamos as linhas onde estão esses elementos.

Código-fonte 18: Verificação do smell *Missing Union Type Abstraction*

```
if(
    value.type == "TSTypeAnnotation" &&
    value.typeAnnotation.type == "TSUnionType"
) {}
```

Fonte: Elaborado pelo autor.

O *code smell* de *Enum Implicit Values* utiliza uma condição para encontrar nós com *TSEnumMember*, um nó específico para encontrar membros de um *enum*. Em seguida, verificamos no objeto JSON se o membro possui a chave *initializer* como sendo *undefined*. Isso indica que o item não possui uma constante atribuída, que se encaixa com o *code smell* catalogado. Assim, quando algum *enum* desse tipo é identificado, a ferramenta retorna a linha de código onde ele está declarado.

Código-fonte 19: Verificação do smell *Enum Implicit Values*

```
if(value.type == 'TSEnumMember' && value.initializer == undefined) {}
```

Fonte: Elaborado pelo autor.

Para *Multiple Booleans for State* foi feita uma verificação buscando primeiramente por variáveis declaradas. Em seguidas, checamos se a variável possui uma chamada para a função *useState*. Se os argumentos desta função que está sendo chamada forem do tipo *boolean*, que no contexto do *parser* seria *BooleanLiteral*, nós contabilizamos essa ocorrência. Se este número for maior do que o limite estabelecido, retornamos as linhas dessas declarações.

Código-fonte 20: Verificação do smell *Multiple Booleans for State*

```
if(
    value.type === 'VariableDeclarator' &&
    value.init &&
    value.init.callee &&
    value.init.callee.name === 'useState' &&
    value.init.arguments.length > 0 &&
    value.init.arguments[0].type === 'BooleanLiteral'
) {}
```

Fonte: Elaborado pelo autor.

Para detectar *Props With Children Pitfall*, primeiramente checamos se os nós incluem *TSPropertySignature*, um tipo de nó gerado pelo *parser* para declarações de propriedades dentro de uma *interface* ou um *type alias*. Depois conferimos no objeto JSON se *children* é o valor associado à chave *name*. Por fim, a condicional verifica se a chave *type* da *type annotation* é do tipo *TSUndefinedKeyword*, *TSAnyKeyword*, *TSNullKeyword*, *TSNeverKeyword* ou *TSUnknownKeyword*. Quando algum destes tipos são identificados, retornamos na *interface* de saída.

Código-fonte 21: Verificação do smell *Multiple Booleans for State*

```
1 if(
2     value.type == "TSPropertySignature" &&
3     value.key.name == 'children'
4     value.typeAnnotation.typeAnnotation.type == "TSUnknownKeyword" ||
5     value.typeAnnotation.typeAnnotation.type == "TSNeverKeyword" ||
6     value.typeAnnotation.typeAnnotation.type == "TSAnyKeyword" ||
7     value.typeAnnotation.typeAnnotation.type == "TSNullKeyword" ||
8     value.typeAnnotation.typeAnnotation.type == "TSUndefinedKeyword"
9 ) {}
```

Fonte: Elaborado pelo autor.

Os limites para *Missing Union Type Abstraction* e *Multiple Booleans for State* foram obtidos com base na opinião dos desenvolvedores entrevistados anteriormente e nos resultados da revisão da literatura cinzenta, como é o caso *Missing Union Type Abstraction* que na ferramenta SonarQube tem um limite de 3 *union types*[3].

7.2 Validação da ferramenta

A ferramenta ReactSniffer foi estendida conforme descrito. Nesta Seção apresentaremos os resultados obtidos na avaliação da ferramenta com o Modelo TAM. O objetivo desta etapa é compreender o nível de aceitação da ferramenta pelos usuários e identificar possíveis melhorias a serem feitas.

[3] https://rules.sonarsource.com/typescript/RSPEC-4323/

7.2.1 Questionário

Formulamos algumas sentenças a serem avaliadas pelos participantes desta etapa que usaram o ReactSniffer. Na Tabela 1 temos estas sentenças para a utilidade percebida, facilidade uso e intenção de uso futuro.

Tabela 1: Sentenças para UP, FUP e IUF

Utilidade Percebida (UP)
UP1. Utilizar a ferramenta proposta me ajudará a ter mais controle sobre a qualidade do meu código, identificando possíveis problemas.
UP2. Utilizar a ferramenta proposta me ajudará a identificar antecipadamente possíveis problemas.
UP3. Utilizar a ferramenta proposta aumentará a minha produtividade, ao permitir uma identificação ágil das áreas problemáticas do código.
UP4. A ferramenta me fornece *insights* sobre áreas do código React que podem precisar de melhorias para evitar problemas futuros.
UP5. Usar essa ferramenta permitirá que eu economize tempo nos meus projetos, identificando rapidamente áreas problemáticas no código do React.
UP6. A ferramenta proposta será útil no meu dia-a-dia, ajudando a compreender e aprimorar o entendimento sobre o assunto.
UP7. A ferramenta é valiosa devido à sua habilidade de gerar uma saída com detalhes sobre a localização dos *code smells*.
UP8. A enumeração da quantidade de cada tipo de *code smell* identificado é valiosa.
UP9. No geral, acho a ferramenta proposta útil para o meu trabalho/projetos.
Facilidade de Uso Percebida (FUP)
FUP1. Compreender e interagir com a ferramenta proposta é uma experiência intuitiva.
FUP2. A ferramenta proposta mostra informações claras sobre os *code smells*.
FUP3. Com a ferramenta proposta, consigo informações claras sobre a localização dos *smells* nas linhas de código.
FUP4. No geral, achei a ferramenta proposta fácil de se utilizar.
Intenção de Uso Futuro (IUF)
IUF1. Utilizaria a ferramenta proposta em meu trabalho/projetos.
IUF2. Acho a visão fornecida pela ferramenta sobre os *code smells* mais interessante do que o que outras ferramentas ou *plugins* oferecem.

Fonte: Elaborado pelo autor.

Com essas sentenças e a escala com graus de concordância com as mesmas, formulamos um questionário para os participantes da validação. As questões estão disponibilizadas no Apêndice C.

7.2.2 Participantes da validação

Após estendermos a ferramenta, a disponibilizamos para os estudantes das disciplinas de Manutenção de *Software* e Qualidade de *Software* do Campus de Quixadá, que realizarem a identificação, refatoração de *code smells* e contribuição em projetos React de código aberto. Realizamos algumas aulas e gravações de instruções para explicar aos estudantes como instalar a ferramenta, mostrar as saídas que ela gera e como refatorar alguns dos *code smells*. Por fim, depois de usarem a ferramenta, encaminhamos o formulário para os participantes ao final das disciplinas. Como resultado, 30 alunos utilizaram a ferramenta e 16 responderam o questionário.

A maioria dos participantes, 50% deles, tem experiência entre 1 a 3 anos com o

React, 44% tem menos de 1 ano de experiência e 6% possui entre 3 e 5 anos de experiência. Quanto ao curso dos respondentes, 69% são estudantes de Engenharia de *Software*, 25% de Ciência da Computação e 6% de Sistemas de Informação.

7.2.3 Resultados da avaliação

Apresentamos os resultados da pesquisa sobre utilidade percebida, facilidade de uso e intenção de uso futuro da ferramenta ReactSniffer a seguir.

7.2.3.1 Utilidade percebida

Na Figura 5 temos a representação da distribuição das respostas obtidas para cada sentença do questionário. A altura da barra indica a quantidade de respostas para diferentes níveis de concordância. Adicionamos também rótulos nas barras que indicam a porcentagem de cada nível de concordância em relação ao total de respostas.

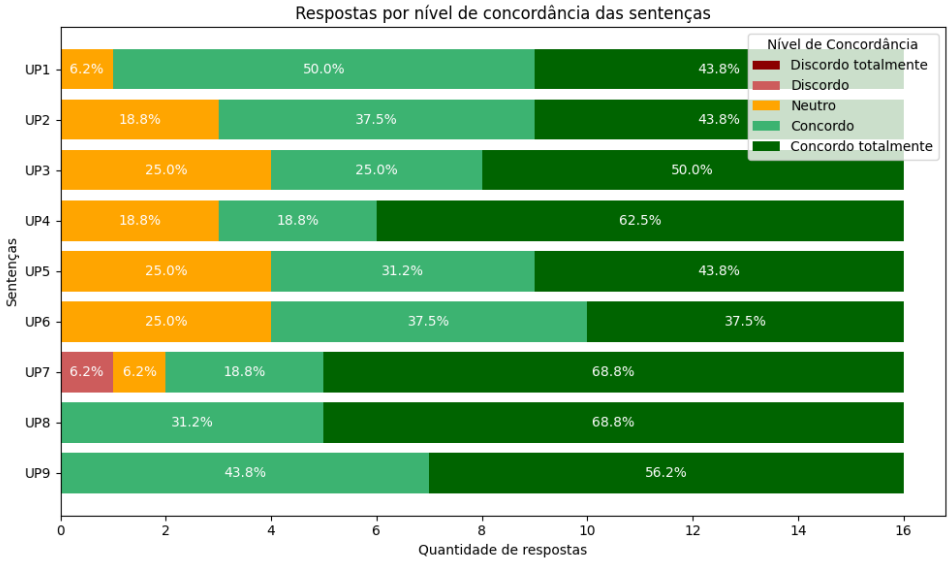
De acordo com a opinião dos participantes, a ferramenta ReactSniffer é proficiente. Todos consideram o ReactSniffer como útil para seus trabalhos/projetos, sendo que aproximadamente 44% concordam com a utilidade da ferramenta e 56% concordam totalmente com esta sentença, como pode ser visto na barra no item **UP9**. Portanto, a ferramenta foi bem programada para atender as necessidades dos desenvolvedores que a utilizaram.

Os dados do item **UP1** sugerem uma tendência de aprovação de 94% em relação à utilidade da ferramenta para se ter controle sobre a qualidade do código e identificar *code smells*, sendo uma pequena proporção de neutros (6%). Logo, predominantemente, as respostas dos estudantes sugerem uma receptividade positiva à ferramenta.

UP2, **UP3**, **UP4** e **UP5** são sentenças que estão relacionadas e que avaliam a capacidade de identificar *smells* antecipadamente, ser produtivo e economizar tempo. Elas apresentam resultados um pouco diferentes. As afirmações possuem resultados positivos, com 81% dos estudantes concordando com antecipação e identificação de áreas problemáticas usando ferramenta, enquanto 75% concordam com a economia de tempo e aumento da produtividade. Isso sugere que os respondentes consideram a realização da tarefa de identificação de *code smells* mais eficiente na ferramenta avaliada. Enquanto a economia de tempo e aumento da produtividade com o reconhecimento de áreas problemáticas não é tão observada em comparação com a antecipação na revelação dos *smells*.

UP6, **UP7** e **UP8** estão associados, já que avaliam a utilidade da ferramenta na com-

Figura 5: Resultados da avaliação da utilidade percebida



Fonte: Elaborada pelo autor.

preensão e melhoria do código. Os níveis de concordância total com a utilidade da enumeração gerada na saída do ReactSniffer em **UP7** e **UP8** revelam que 69% concordam fortemente com essas afirmações, sendo 88% e 100% a concordância somada, respectivamente. Isso mostra que a ferramenta consegue fornecer informações valiosas na saída gerada na linha de comando e no CSV que aponta para as linhas com *smells*.

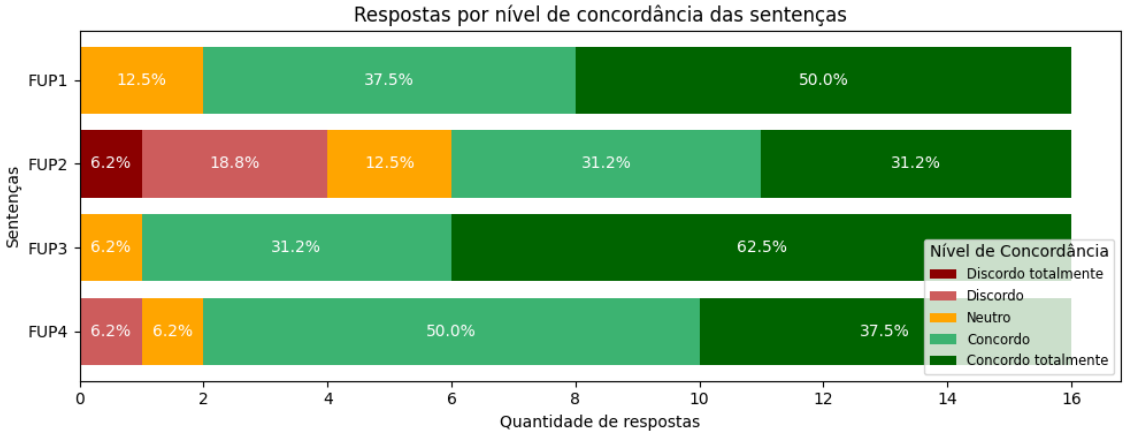
Para **UP6**, temos uma distribuição igual entre os que concordam e os que concordam totalmente, com concordância total menor em comparação com as correlacionadas, de 37.5%. A concordância somada é de 75%, considerada uma concordância positiva e com 25% de respostas neutras. Isso significa que os estudantes concordam com a utilidade prática do ReactSniffer nas atividades diárias, porém, temos uma incerteza de alguns sobre esses benefícios.

7.2.3.2 Facilidade de uso

A Figura 6 ilustra os resultados obtidos para as sentenças de facilidade de uso. A sentença **FUP4** avalia a facilidade de uso da ferramenta. Para aproximadamente 87% dos participantes da validação, o ReactSniffer é fácil de se utilizar. Sendo que 50% destes concordam com a afirmação e 37.5% concordando inteiramente.

As afirmações **FUP2** e **FUP3** se referem à competência da ferramenta em fornecer informações compreensíveis sobre os *code smells*. Para a primeira sentença, temos um equilíbrio entre a concordância (50%) e a discordância (31%). Para **FUP3**, a concordância dos participantes é forte (93.7%). Esses resultados mostram que embora a ferramenta ofereça a localização dos

Figura 6: Resultados da avaliação da facilidade de uso



Fonte: Elaborada pelo autor.

smells, essas informações não são claras suficiente para os avaliadores.

Por fim, **FUP1** avalia se o ReactSniffer é intuitivo, enquanto os estudantes utilizam a ferramenta. Como resultado, 87.5% dos respondentes consideram a ferramenta intuitiva e fácil de se entender e usar. Por tanto, de modo geral, ela é clara e objetiva para a maioria dos usuários.

7.2.3.3 Intenção de uso futuro

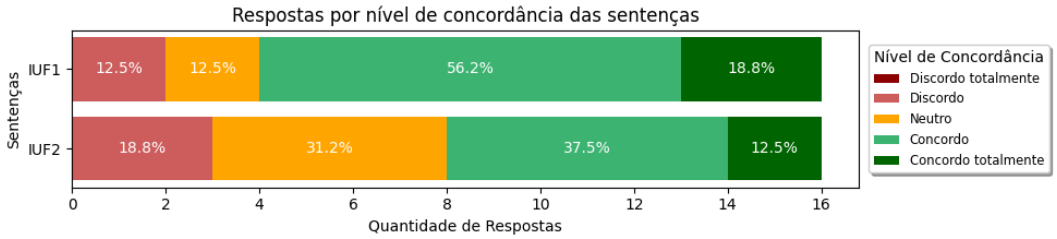
Nas últimas sentenças, avaliamos a intenção dos participantes de usarem a ferramenta futuramente. Para isso, avaliamos em *IUF1* a disposição dos participantes de utilizarem o ReactSniffer em outros projetos. Para a maioria, 75% dos estudantes, temos uma receptividade considerável que é favorável em usar a ferramenta, com apenas 12.5% (2 participantes) discordando.

A afirmação **IUP2** avalia a percepção dos estudantes sobre as ferramentas e *plugins* existentes para identificar *code smells*. Analisando as respostas, percebemos que existe uma divisão na impressão dos avaliadores. 50% concordam com a afirmação, mas temos alguns neutros (31.2%) e discordâncias (18.8%). Isso indica que embora os desenvolvedores tenham interesse em usar a ferramenta posteriormente, eles não veem vantagens comparativas em relação a outras ferramentas.

7.2.4 Impactos da ferramenta

O ReactSniffer fornece vários benefícios e informações valiosas na identificação de *code smells*. Na validação da ferramenta, conseguimos disponibilizá-la para estudantes que a utilizaram para detectar e refatorar *smells* em projetos de código aberto, contribuindo com a

Figura 7: Resultados da avaliação da intenção de uso futuro



Fonte: Elaborada pelo autor.

comunidade de desenvolvimento de *software*.

Portanto, os desenvolvedores podem utilizam a ferramenta para identificar *code smells* e contribuir com a melhoria da qualidade de *software* de projetos React, identificando áreas problemáticas do código que precisam de uma atenção. Em nossa avaliação, todos os estudantes concordam que o ReactSniffer é útil para os projetos React em que eles codificam.

7.2.5 Limitações

Como avaliado na variável de facilidade de uso, alguns desenvolvedores discordaram da sentença **FUP2**, que se refere à facilidade de compreensão das informações geradas pela ferramenta. Portanto, é necessário considerar mudanças na enumeração dos *code smells* na saída da linha de comando e no CSV gerado que aponta para as linhas onde os *smells* se apresentam. Como também, alguns usuários relataram que o *smell JSX outside the render method* estava sendo detectado em partes que não correspondiam ao *smell*.

Por fim, a afirmação **IUP2**, que compara o ReactSniffer com outras ferramentas, ficou dividida entre os avaliadores, o que pode indicar algumas limitações na ferramenta. Portanto, em trabalhos futuros, poderão ser adicionadas novas funcionalidades e visualizações que a diferenciem das outras.

8 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho realizamos a identificação de *code smells* do React com o TypeScript, catalogando *code smells* encontrados na literatura cinzenta, mostrando que existem *smells* específicos dessas tecnologias. Para isso, seguimos uma metodologia de revisão usando como fonte blogs, vídeos, documentações, fóruns, etc. Isso nos retornou uma lista com mais de 28 *smells* candidatos. Após realizar uma validação, finalizamos a lista com os *code smells* mais comuns do desenvolvimento com React e TypeScript, respondendo nossa QP1: *code smells Any Type*, *Non-Null Assertions*, *Missing Union Type Abstraction*, *Enum Implicit Values*, *Props with Children Pitfall* e *Multiple Booleans for State*.

Posteriormente, entrevistamos desenvolvedores que trabalham atualmente com o React. Eles discorreram sobre más práticas observadas durante o desenvolvimento de *interfaces web* com esta biblioteca. Os participantes também avaliaram cada um dos *smells* quanto a frequência e impacto negativo. Confirmamos que os *code smells* identificados estão presentes nos projetos que os desenvolvedores contribuem e prejudicam a qualidade do *software*. Por fim, entendemos melhor como ocorrem esses anti-padrões, e descrevemos e exemplificamos estes *smells*.

Em sequência, realizamos uma *survey* com desenvolvedores React. Divulgamos um formulário em alguns canais da comunidade das destas tecnologias. Avaliamos a frequência e impacto negativo dos *code smells* identificados até a etapa da revisão. A partir disso, respondemos nossa QP2 para cada *code smell* segundo a avaliação dos desenvolvedores. Os *code smells Any Type*, *Multiple Booleans for State* e *Non-Null Assertions* são mais frequentes e impactam negativamente a legibilidade, manutenção e evolução do *software*.

Com o entendimento de como ocorrem estes *code smells*, realizamos a extensão da ferramenta ReactSniffer. Para isso, criamos um *fork* do repositório da ferramenta e adicionamos ao código algumas condicionais que identificam os *smells* através da árvore sintática abstrata gerada pelo Babel *parser*. Todos os 6 *code smells* foram adicionados na ferramenta e podem ser detectados em projetos React e TypeScript.

Por fim, validamos a ferramenta utilizando o Modelo de Aceitação Tecnológica (*Technology Acceptance Model* — TAM). Nesta fase, providenciamos para os estudantes das disciplinas de Manutenção de *Software* e Qualidade de *Software* o ReactSniffer. Os alunos utilizaram ele para contribuir em projetos de código aberto e ao final avaliaram a ferramenta. Para isso, disponibilizamos um questionário com algumas sentenças, para que os participantes

avaliassem o nível de concordância com estas afirmações. Respondendo nossa QP3, observamos que a ferramenta é útil e fácil de usar, necessitando apenas de ajustes na detecção do *smell JSX outside the render method*, na visualização das saídas geradas por ela e de novas funcionalidades.

Como trabalho futuro, podem ser realizadas novas pesquisas que identifiquem outros *code smells* mais específicos para o desenvolvimento de *interfaces web* modulares. Como também, adicionar novas visualizações ao ReactSniffer, principalmente nas saídas que são geradas com os *smells* identificados. Novas pesquisas que identifiquem técnicas de refatoração dos *code smells* deste trabalho e de outros trabalhos também poderão ser realizadas. Com isso, sugestões de refatorações podem ser adicionadas na ferramenta.

REFERÊNCIAS

Al Dallal, J. Identifying refactoring opportunities in object-oriented code: A systematic literature review. **Information and Software Technology**, Elsevier, v. 58, p. 231–249, 2015.

ALMASHFI, N.; LU, L. Code smell detection tool for java script programs. In: **2020 5th International Conference on Computer and Communication Systems (ICCCS)**. [*S. l.*]: IEEE Computer Society, 2020. p. 172–176.

BAJAMMAL, M.; MAZINANIAN, D.; MESBAH, A. Generating reusable web components from mockups. In: **Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering**. [*S. l.*]: Association for Computing Machinery, 2018. p. 601–611.

BESSGHAIER, N.; OUNI, A.; MKAOUER, M. W. A longitudinal exploratory study on code smells in server side web applications. **Software Quality Journal**, Springer International Publishing, p. 901–941, 2021.

BIERMAN, G.; ABADI, M.; TORGERSEN, M. Understanding typescript. In: **ECOOP 2014 - Object-Oriented Programming**. [*S. l.*]: Springer Berlin Heidelberg, 2014. p. 257–281.

BOERSMA, S.; LUNGU, M. React-bratus: Visualising react component hierarchies. In: **2021 Working Conference on Software Visualization (VISSOFT)**. [*S. l.*]: IEEE Computer Society, 2021. p. 130–134.

BOGNER, J.; MERKEL, M. To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github. In: **2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)**. [*S. l.*]: IEEE Computer Society, 2022. p. 658–669.

DAS, D.; MARUF, A. A.; ISLAM, R.; LAMBARIA, N.; KIM, S.; ABDELFATTAH, A. S.; CERNY, T.; FRAJTAK, K.; BURES, M.; TISNOVSKY, P. Technical debt resulting from architectural degradation and code smells: A systematic mapping study. **SIGAPP Appl. Comput. Rev.**, Association for Computing Machinery, v. 21, n. 4, p. 20–36, 2022.

DAVIS, F. D. Perceived usefulness, perceived ease of use, and user acceptance of information technology. **MIS Quarterly**, Management Information Systems Research Center, University of Minnesota, v. 13, n. 3, p. 319–340, 1989.

EILERTSEN, A. M.; MURPHY, G. C. The usability (or not) of refactoring tools. In: **2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [*S. l.*]: IEEE Computer Society, 2021. p. 237–248.

FARD, A. M.; MESBAH, A. Jsnose: Detecting javascript code smells. In: **2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [*S. l.*]: IEEE Computer Society, 2013. p. 116–125.

FELDTHAUS, A.; MØLLER, A. Checking correctness of typescript interfaces for javascript libraries. In: **Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications**. [*S. l.*]: Association for Computing Machinery, 2014. p. 1–16.

FERREIRA, F.; VALENTE, M. T. Detecting code smells in react-based web apps. **Information and Software Technology**, Elsevier, v. 155, 2023.

FONTANA, F. A.; MANGIACAVALLI, M.; POCHIERO, D.; ZANONI, M. On experimenting refactoring tools to remove code smells. In: **Scientific Workshop Proceedings of the XP2015**. [*S. l.*]: Association for Computing Machinery, 2015.

FOWLER, M. **Refactoring**: improving the design of existing code. [*S. l.*]: Addison-Wesley Professional, 2018.

GAROUSI, V.; FELDERER, M.; MäNTYLä, M. V. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. **Information and Software Technology**, Elsevier, v. 106, p. 101–121, 2019.

HOZANO, M.; GARCIA, A.; FONSECA, B.; COSTA, E. Are you smelling it? investigating how similar developers detect code smells. **Information and Software Technology**, Elsevier, v. 93, p. 130–146, 2018.

IVANOVA, S.; GEORGIEV, G. Using modern web frameworks when developing an education application: a practical approach. In: **2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)**. [*S. l.*]: IEEE Computer Society, 2019. p. 1485–1491.

JAVEED, A. Performance optimization techniques for reactjs. In: **2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)**. [*S. l.*]: IEEE Computer Society, 2019. p. 1–5.

JOHANNES, D.; KHOMH, F.; ANTONIOL, G. A large-scale empirical study of code smells in javascript projects. **Software Quality Journal**, Springer International Publishing, n. 10, p. 1271–1314, 2019.

KAUR, A.; DHIMAN, G. A review on search-based tools and techniques to identify bad code smells in object-oriented systems. In: **Harmony Search and Nature Inspired Optimization Algorithms**. [*S. l.*]: Springer, 2019. p. 909–921.

KAUSHALYA, T.; PERERA, I. Framework to migrate angularjs based legacy web application to react component architecture. In: **2021 Moratuwa Engineering Research Conference (MERCon)**. [*S. l.*]: IEEE Computer Society, 2021. p. 693–698.

KIM, M.; ZIMMERMANN, T.; NAGAPPAN, N. An empirical study of refactoringchallenges and benefits at microsoft. **IEEE Transactions on Software Engineering**, IEEE Computer Society, v. 40, p. 633–649, 2014.

LACERDA, G.; PETRILLO, F.; PIMENTA, M.; GUéHéNEUC, Y. G. Code smells and refactoring: A tertiary systematic review of challenges and observations. **Journal of Systems and Software**, Elsevier, v. 167, p. 110610, 2020.

MENS, T.; TOURWE, T. A survey of software refactoring. **IEEE Transactions on Software Engineering**, IEEE Computer Society, v. 30, n. 2, p. 126–139, 2004.

MENSHAWY, R. S.; YOUSEF, A. H.; SALEM, A. Code smells and detection techniques: A survey. In: **2021 International Mobile, Intelligent, and Ubiquitous Computing Conference**. [*S. l.*]: IEEE Computer Society, 2021. p. 78–83.

NOVAC, C. M.; NOVAC, O. C.; SFERLE, R. M.; GORDAN, M. I.; BUJDOSó, G.; DINDELEGAN, C. M. Comparative study of some applications made in the vue.js and react.js frameworks. In: **2021 16th International Conference on Engineering of Modern Electric Systems (EMES)**. [*S. l.*]: IEEE Computer Society, 2021. p. 1–4.

OIZUMI, W.; GARCIA, A.; SOUSA, L. da S.; CAFEO, B.; ZHAO, Y. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In: **Proceedings of the 38th International Conference on Software Engineering**. [*S. l.*]: Association for Computing Machinery, 2016. p. 440–451.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; FASANO, F.; OLIVETO, R.; LUCIA, A. D. A large-scale empirical study on the lifecycle of code smell co-occurrences. **Information and Software Technology**, Elsevier, v. 99, p. 1–10, 2018.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; FASANO, F.; OLIVETO, R.; LUCIA, A. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. **Empirical Software Engineering**, Springer International Publishing, 2018.

PANO, A.; GRAZIOTIN, D.; ABRAHAMSSON, P. Factors and actors leading to the adoption of a javascript framework. **Empirical Software Engineering**, Springer International Publishing, 2018.

PANTIUCHINA, J.; ZAMPETTI, F.; SCALABRINO, S.; PIANTADOSI, V.; OLIVETO, R.; BAVOTA, G.; PENTA, M. D. Why developers refactor source code: A mining-based study. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, v. 29, n. 4, 2020.

PERUMA, A.; SIMMONS, S.; ALOMAR, E. A.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A. How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. **Empirical Software Engineering**, Springer, v. 27, n. 1, 2021.

PSAILA, G. Virtual dom: An efficient virtual memory representation for large xml documents. In: **2008 19th International Workshop on Database and Expert Systems Applications**. [*S. l.*]: IEEE Computer Society, 2008. p. 233–237.

RASTOGI, A.; SWAMY, N.; FOURNET, C.; BIERMAN, G.; VEKRIS, P. Safe efficient gradual typing for typescript. In: **Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. [*S. l.*]: Association for Computing Machinery, 2015. p. 167–180.

SABOURY, A.; MUSAVI, P.; KHOMH, F.; ANTONIOL, G. An empirical study of code smells in javascript projects. In: **2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [*S. l.*]: IEEE Computer Society, 2017. p. 294–305.

SANTOS, H. M. dos; DURELLI, V. H. S.; SOUZA, M.; FIGUEIREDO, E.; SILVA, L. T. da; DURELLI, R. S. Cleangame: Gamifying the identification of code smells. In: **Proceedings of the XXXIII Brazilian Symposium on Software Engineering**. [*S. l.*]: Association for Computing Machinery, 2019. p. 437–446.

SARAFIM, D.; DELGADO, K.; CORDEIRO, D. Random forest for code smell detection in javascript. In: **Anais do XIX Encontro Nacional de Inteligência Artificial e Computacional**. [*S. l.*]: SBC, 2022. p. 13–24.

SHARMA, T.; GUPTA, S.; SINGH, U. R. Analyzing the difference between reactjs and angularjs. In: **2023 International Conference on Computational Intelligence, Communication Technology and Networking (CICTN)**. [*S. l.*]: IEEE Computer Society, 2023. p. 37–42.

SILVA, D.; TSANTALIS, N.; VALENTE, M. T. Why we refactor? confessions of github contributors. In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [*S. l.*]: Association for Computing Machinery, 2016. p. 858–870.

SZOKE, G.; NAGY, C.; FüLöP, L. J.; FERENC, R.; GYIMóTHY, T. Faultbuster: An automatic code smell refactoring toolset. In: **2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [*S. l.*]: IEEE Computer Society, 2015. p. 253–258.

WALKER, A.; DAS, D.; CERNY, T. Automated code-smell detection in microservices through static analysis: A case study. **Applied Sciences**, MDPI AG, v. 10, p. 1–20, 2020.

XING, Y.; HUANG, J.; LAI, Y. Research and analysis of the front-end frameworks and libraries in e-business development. In: **Proceedings of the 2019 11th International Conference on Computer and Automation Engineering**. [*S. l.*]: Association for Computing Machinery, 2019. p. 68–72.

YAMASHITA, A.; MOONEN, L. Do developers care about code smells? an exploratory survey. In: **2013 20th Working Conference on Reverse Engineering (WCRE)**. [*S. l.*: *s. n.*], 2013. p. 242–251.

APÊNDICE A – FONTES DA LITERATURA CINZENTA

Tabela 2: Fontes rastreáveis

	Link	Critério
F1	https://react.dev	Documentação do React
F2	https://itnext.io	Autor com experiência e publicou outros
F3	https://blog.bitsrc.io	Autor com experiência e publicou outros
F4	https://yosua-halim.medium.com	Autor com experiência e publicou outros
F5	https://perssondennis.com	Publicou outros e exemplos
F6	https://javacript.plainenglish.io	Autor com experiência e publicou outros
F7	https://rules.sonarsource.com/RSPEC-2966	Texto objetivo e exemplos
F8	https://rules.sonarsource.com/RSPEC-6481	Texto objetivo e exemplos
F9	https://rules.sonarsouce.com/RSPEC-6439	Texto objetivo e exemplos
F10	https://rules.sonarsouce.com/RSPEC-4204	Texto objetivo e exemplos
F11	https://rules.sonarsouce.com/RSPEC-4323	Texto objetivo e exemplos
F12	https://bluepnume.medium.com	Publicou outros e exemplos
F13	https://languageimperfect.com	Texto objetivo e exemplos
F14	https://rules.sonarsouce.com/RSPEC-109	Texto objetivo e exemplos

F15	https://totaltypescript.com	Autor com experiência e publicou outros
F16	https://stackoverflow.com/questions/75919242	Texto objetivo e exemplos
F17	https://softwareengineering.stackexchange.com	Texto objetivo e com exemplos
F18	https://dev.to/itshugo	Autor com experiência e publicou outros
F19	https://zhenghao.io	Autor com experiência e publicou outros
F20	https://reddit.com/r/typescript	Texto objetivo e exemplos
F21	https://codeburst.io	Autor com experiência e publicou outros
F22	https://makerkit.dev/blog	Texto objetivo e exemplos
F23	https://betterprogramming.pub	Autor com experiência e publicou outros
F24	https://articles.wesionary.team	Autor com experiência e publicou outros
F25	https://vhudyma-blog.eu	Autor com experiência e publicou outros
F26	https://prismic.io/blog	Autor com experiência e publicou outros
F27	https://antongunnarsson.com/	Texto objetivo e exemplos
F28	https://blog.logrocket.com/	Texto objetivo e exemplos
F29	https://isamatov.com	Texto objetivo e exemplos

F30	https://developerway.com	Texto objetivo e exemplos
F31	https://levelup.gitconnected.com	Texto objetivo e exemplos
F32	https://quokkalabs.com	Autor com experiência e publicou outros
F33	https://riptutorial.com/reactjs	Texto obetivo e exemplos
F34	https://aroundreact.com	Texto objetivo e exemplos
F35	https://crystallize.com/blog	Autor com experiência e publicou outros
F36	https://flaming.codes	Texto objetivo e exemplos
F37	https://www.tiny.cloud/blog	Autor publicou outros e texto objetivo
F38	https://ed.software/articles	Autor com experiência e publicou outros
F39	https://hackernoon.com	Autor com experiência e publicou outros
F40	https://www.kantega.no/blogg	Autor publicou outros e texto objetivo
F41	https://steven-lemon182.medium.com	Autor publicou outros e texto objetivo

Fonte: Elaborado pelo autor.

APÊNDICE B – FORMULÁRIO DA *SURVEY*

1. Tempo de experiência com React
    a) Menos de 1 ano
    b) Entre 1 e 3 anos
    c) Entre 3 e 5 anos
    d) Mais de 5 anos
2. Quantos projetos que você participou/contribuiu utilizavam React com TypeScript?
    a) 1-2 projetos
    b) 3-5 projetos
    c) 6 ou mais projetos
3. Com que frequência esses *code smells* ocorrem nos projetos React que você trabalhou?
    – Any Type
    – Non-Null Assertions
    – Missing Union Type Abstraction
    – Enum Implicit Values
    a) Muito raro
    b) Raro
    c) Ocasionalmente
    d) Às vezes
    e) Sempre
4. Qual é a relevância dos *code smells* apresentados em termos de seu potencial impacto negativo na manutenção, legibilidade e evolução do código?
    – Any Type
    – Non-Null Assertions
    – Missing Union Type Abstraction
    – Enum Implicit Values
    a) Muito baixo
    b) Baixo
    c) Moderado
    d) Alto
    e) Muito alto
5. Com que frequência esses *code smells* ocorrem nos projetos React que você trabalhou?

- Multiple Booleans for State
- Props with Children Pitfall

a) Muito raro
b) Raro
c) Ocasionalmente
d) Às vezes
e) Sempre

6. Qual é a relevância dos *code smells* apresentados em termos de seu potencial impacto negativo na manutenção, legibilidade e evolução do código?
   - Multiple Booleans for State
   - Props with Children Pitfall

   a) Muito raro
   b) Raro
   c) Ocasionalmente
   d) Às vezes
   e) Sempre

APÊNDICE C – FORMULÁRIO DE VALIDAÇÃO DA FERRAMENTA

1. Tempo de experiência com o React
    a) Menos de 1 anos
    b) Entre 1 e 3 anos
    c) Entre 3 e 5 anos
    d) Mais de 5 anos
2. Qual o seu curso?
    a) Engenharia de Software
    b) Ciência da Computação
    c) Engenharia da Computação
    d) Redes de Computadores
    e) Design Digital
    f) Sistemas de Informação
3. Para as sentenças sobre utilidade percebida, assinale:
    – Utilizar a ferramenta proposta me ajudará a ter mais controle sobre a qualidade do meu código, identificando possíveis problemas
    a) Discordo totalmente
    b) Discordo
    c) Neutro
    d) Concordo
    e) Concordo totalmente
    – Utilizar a ferramenta proposta me ajudará a identificar antecipadamente possíveis problemas
    a) Discordo totalmente
    b) Discordo
    c) Neutro
    d) Concordo
    e) Concordo totalmente
    – Utilizar a ferramenta proposta aumentará a minha produtividade, ao permitir uma identificação ágil das áreas problemáticas do código
    a) Discordo totalmente
    b) Discordo

c) Neutro
d) Concordo
e) Concordo totalmente

– A ferramenta me fornece *insights* sobre áreas do código React que podem precisar de melhorias para evitar problemas futuros

a) Discordo totalmente
b) Discordo
c) Neutro
d) Concordo
e) Concordo totalmente

– Usar essa ferramenta permitirá que eu economize tempo nos meus projetos, identificando rapidamente áreas problemáticas no código do React

a) Discordo totalmente
b) Discordo
c) Neutro
d) Concordo
e) Concordo totalmente

– A ferramenta proposta será útil no meu dia-a-dia, ajudando a compreender e aprimorar o entendimento sobre o assunto

a) Discordo totalmente
b) Discordo
c) Neutro
d) Concordo
e) Concordo totalmente

– A ferramenta é valiosa devido à sua habilidade de gerar uma saída com detalhes sobre a localização dos *code smells*

a) Discordo totalmente
b) Discordo
c) Neutro
d) Concordo
e) Concordo totalmente

– A enumeração da quantidade de cada tipo de *code smell* identificado é valiosa

a) Discordo totalmente

b) Discordo

c) Neutro

d) Concordo

e) Concordo totalmente

– No geral, acho a ferramenta proposta útil para o meu trabalho/projetos

a) Discordo totalmente

b) Discordo

c) Neutro

d) Concordo

e) Concordo totalmente

4. Para as sentenças sobre facilidade de uso, assinale:

– Compreender e interagir com a ferramenta proposta é uma experiência intuitiva

a) Discordo totalmente

b) Discordo

c) Neutro

d) Concordo

e) Concordo totalmente

– A ferramenta proposta mostra informações claras sobre os *code smells*

a) Discordo totalmente

b) Discordo

c) Neutro

d) Concordo

e) Concordo totalmente

– Com a ferramenta proposta, consigo informações claras sobre a localização dos *smells* nas linhas de código

a) Discordo totalmente

b) Discordo

c) Neutro

d) Concordo

e) Concordo totalmente

– No geral, achei a ferramenta proposta fácil de se utilizar

a) Discordo totalmente
b) Discordo
c) Neutro
d) Concordo
e) Concordo totalmente

5. Sobre as sentenças sobre intenção de uso futuro, assinale:
   – Utilizaria a ferramenta proposta em meu trabalho/projetos
   a) Discordo totalmente
   b) Discordo
   c) Neutro
   d) Concordo
   e) Concordo totalmente
   – Acho a visão fornecida pela ferramenta sobre os *code smells* mais interessante do que o que outras ferramentas ou *plugins* oferecem
   a) Discordo totalmente
   b) Discordo
   c) Neutro
   d) Concordo
   e) Concordo totalmente