Análise e Detecção de Code Smells em Aplicações React e React Native

Lucas Silva de Mendonça, Leopoldo Motta Teixeira

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
50740-560 – Recife – PE – Brazil

lsm5@cin.ufpe.br, lmt@cin.ufpe.br

Abstract. *In this article, we present a catalog of 16 code smells extracted from a gray literature review, combined with a quick review, with the focus exclusively on React and React Native technologies. We enhanced an Open Source tool, ReactSniffer2, which, when applied to 16 Open Source projects, allowed us to verify the frequency of this catalog. The most frequent smells were: String Literals (19.04%), Props Spreading (16.39%), and Component Nesting/JSX Outside the Render (13.99%). In conclusion, this work contributes to the awareness about code smells in React and React Native, as well as for the improvement of linting tools.*

Resumo. *Neste artigo, apresentamos um catálogo de 16 code smells extraídos de uma revisão de literatura cinza, combinado com uma revisão rápida, com o foco exclusivamente nas tecnologias React e React Native. Aprimoramos uma ferramenta Open Source, ReactSniffer2, que, ao aplicá-la em 16 projetos Open Source, permitiu verificar a frequência desse catálogo. Os smells mais frequentes foram: String Literals (19,04%), Props Spreading (16,39%), e Component Nesting/JSX Outside the Render (13,99%). Em conclusão, este trabalho contribui para o conhecimento sobre code smells em React e React Native, bem como para o aprimoramento de ferramentas de lint.*

1. Introdução

A evolução do software é inevitável [Svahnberg 2003], seja para dar manutenção no código, resolvendo algum bug encontrado, seja para atender novos requisitos ou novas funcionalidades. Fazer isso ajuda a aumentar a vida útil do software e melhora aspectos como flexibilidade, integração, manutenibilidade e legibilidade. Contudo, a manutenção pode levar o programador, se não preparado, a implementar práticas ruins de programação. Essas práticas ruins, também conhecidas como *code smells*, não necessariamente tornam o sistema incorreto, mas dificultam que o sistema seja mantido, compreendido e refatorado. O termo *code smell* foi amplamente difundido por Kent Beck no livro "Refatoração: Aperfeiçoando o Design de Códigos Existentes" [Fowler 2006], escrito em parceria com Martin Fowler. Pode ser definido como um sintoma para um problema mais profundo na aplicação, sendo indicadores de um problema, podendo não ser a causa do problema em si.

Com o passar dos anos, a hegemonia da linguagem JavaScript para desenvolver sistemas front-end se torna cada vez mais evidente. Este ano, o JavaScript foi novamente reconhecido como a linguagem de programação mais popular entre os desenvolvedores,

de acordo com a pesquisa da Stack Overflow [1], marcando seu décimo primeiro ano como líder. Diante desta popularidade, é comum encontrar softwares que utilizam essa linguagem como base para seus sistemas. No entanto, para desenvolver sistemas que permitam uma fácil implementação, manutenção e reutilização, os programadores costumam recorrer a frameworks e/ou bibliotecas que auxiliam na implementação de interfaces de usuário complexas [Ramos et al. 2018]. Exemplos populares de frameworks e bibliotecas são Angular [2], React [3] e Vue [4], utilizados para o desenvolvimento de aplicações web, e React Native [5] e Flutter [6], utilizados para o desenvolvimento de aplicações mobile.

Dada a recência das tecnologias React e React Native, e suas semelhanças, existem poucos materiais que abordem os code smells comuns em aplicações que as utilizam [Ferreira and Valente 2022]. Visando isso, este artigo propõe criar um catálogo que agrupe os principais smells, ao mesmo tempo em que analisa o nível de frequência frente a projetos Open Source, com o intuito de servir como material de estudo para aprimoramento de ferramentas de linting.

A motivação inicial deste artigo é responder às seguintes perguntas:

P1) Quais são os principais code smells encontrados no React e React Native?
P2) Com que frequência são encontrados em projetos reais?

Para lidar com a primeira pergunta, fizemos uma revisão rápida e uma revisão de literatura cinza, a fim de identificar quais seriam os principais code smells que são divulgados e encontrados pelos programadores. Nossas pesquisas encontraram 121 code smells, dos quais foram extraídos 16 para um maior aprofundamento e análise.

E para responder à segunda pergunta, aprimoramos uma ferramenta Open Source, ReactSniffer2, na qual a utilizando-a e aplicando-a em projetos encontrados no Github, pudermos ver a frequência com que esses code smells são identificados em projetos reais.

Os smells mais encontrados foram: String Literals (19,04%), Props Spreading (16,39%), Component Nesting/JSX Outside the Render (13,99%). Os smells com as menores frequências foram: Too Many UseState (0,01%), Large UseEffect (0,03%), Procedural Patterns (0,54%).

O artigo está organizado da seguinte forma: na seção 2, encontram-se explicações sobre as tecnologias escolhidas (React e React Native) e suas semelhanças. Na seção 3, é aprofundada a metodologia utilizada no presente artigo. Na seção 4, apresentamos o catálogo de code smells proposto. Na seção 5, explicamos a ferramenta de detecção dos code smells, ReactSniffer2. Na seção 6, apresentamos os resultados obtidos da ferramenta. Na seção 7, apresentamos as ameaças à validade da metodologia. As conclusões são apresentadas na seção 8 e, na seção 9, explicamos as perspectivas sobre os trabalhos futuros.

[1] https://survey.stackoverflow.co/2023/#most-popular-technologies-language
[2] https://angular.io
[3] https://react.dev
[4] https://vuejs.org
[5] https://reactnative.dev
[6] https://flutter.dev

2. Aprofundamento React e React Native

Aberta ao público em 2013 pelo time da empresa Meta, o React é uma biblioteca, construída baseada na linguagem JavaScript, que ajuda desenvolvedores a criar interfaces de usuário. O React incentiva a criação de componentes dinâmicos e compostos, que podem ao longo do tempo mudar os valores dos dados que possuem, proporcionando uma experiência de usuário mais rica e responsiva [Hunt 2013].

O React possui um mecanismo chamado de Single Page Application (SPA), este mecanismo tem o poder de carregar apenas uma única página HTML e, em seguida, atualiza essa página dinamicamente à medida que o usuário interage com a aplicação. Isto ocorre porque o JavaScript consegue manipular a Document Object Model (DOM) e renderiza os novos conteúdos sem a necessidade de recarregar os antigos, assim toda a página não precisa ficar sendo recarregada. O React faz o uso de uma extensão de sintaxe para JavaScript, chamada de JavaScriptX (JSX) que combina Linguagem de Marcação de Hipertexto (HTML) com JavaScript. Foi através dessa sintaxe que o React ganhou ainda mais popularidade com a comunidade de desenvolvedores.

O uso do JSX permite que os desenvolvedores escrevam código que são semelhantes a HTML, mas que na verdade são JavaScript. Isso facilita a criação de componentes de interface de usuário e a manipulação da DOM, o que torna o desenvolvimento mais eficiente e intuitivo. O JSX também permite o uso do JavaScript, o que é útil para gerenciar e manipular estados da aplicação e seus dados.

O React foi projetado tendo em mente dar suporte a múltiplas plataformas, para que não fosse apenas uma biblioteca focada na web, mas que pudesse expandir para outras áreas, isso permitiu o lançamento do framework móvel React Native, no ano de 2015.

O React Native foi criado, com a mesma filosofia do React, utilizando dos mesmos mecanismos e ferramentas. Com o React Native é possível escrever código de aplicativo para executá-lo em várias plataformas (através de componentes nativos), como Android e iOS, não havendo a necessidade de ter um código para cada uma. O React Native se assemelha muito com o React, pois ambos usam componentes como a unidade básica de construção de interfaces de usuário, possuem o conceito de estados e propriedades para gerenciar os dados dentro dos componentes e ambos possuem ciclo de vida dos componentes, que oferece um melhor dinamismo para seus componentes. Além disso, visualmente, ambos são parecidos, como pode ser observado nas figuras 1 e 2, onde temos um simples componente que possui o estado “message” que guarda a mensagem “Olá, mundo!” e é exibida na tela, quando apertado o botão “Alterar mensagem”, o valor do texto muda para “Mensagem alterada!” e este é exibido na tela.

Devido às suas semelhanças, ambas as tecnologias foram escolhidas para detectar code smells.

3. Metodologia

Este artigo tem como objetivo investigar os principais code smells encontrados nas tecnologias React e React Native, a fim de propor um catálogo e observar a frequência com que ocorrem em projetos Open Source. Para isso, o artigo foi orientado a responder as duas perguntas abaixo:

P1) Quais são os principais code smells encontrados no React e React Native?

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

const ReactComponent = () => {
  const [message, setMessage] = useState('Olá, mundo!');

  return (
    <div>
      <p>{message}</p>
      <button onClick={() => setMessage('Mensagem alterada!')}>
        Alterar mensagem
      </button>
    </div>
  );
};

export default ReactComponent;
```

Figura 1. Exemplo de código React

```jsx
import React, { useState } from 'react';
import { View, Text, Button } from 'react-native';

const ReactNativeComponent = () => {
  const [message, setMessage] = useState('Olá, mundo!');

  return (
    <View>
      <Text>{message}</Text>
      <Button title="Alterar mensagem" onPress={() => setMessage('Mensagem alterada!')} />
    </View>
  );
};

export default ReactNativeComponent;
```

Figura 2. Exemplo de código React Native

P2) Com que frequência são encontrados em projetos reais?

Para responder a primeira pergunta realizamos uma Revisão Rápida (RR) combinando com uma Revisão de Literatura Cinza (RLC). As revisões de literatura são uma ótima maneira de identificar, avaliar a qualidade e a relevância dos estudos existentes para propor uma visão geral sobre o estado atual do campo de estudo [Paré et al. 2015]. A análise e classificação de ambas as técnicas ajudaram a propor o catálogo de code smell em React e React Native.

3.1. Revisão Rápida

Para realizar a Revisão Rápida, seguimos o modelo proposto por [Cartaxo et al. 2018]. A RR tem como objetivo fornecer auxílio à tomada de decisões em problemas encontrados na prática de Engenharia de Software, ajudando os profissionais principalmente quando há uma limitação de tempo e/ou recursos.

A figura 3 retrata o procedimento para realizar a Revisão Rápida. O procedimento possui 3 etapas: A busca no Scopus, a seleção de artigos através de filtros e a extração dos code smells.

Na Revisão Rápida, utilizamos o mecanismo de busca Scopus [7], conforme recomendado pelo modelo. A Scopus é uma base de dados de artigos científicos, com outras

[7]https://www.scopus.com/search/form.uri?display=basic#basic

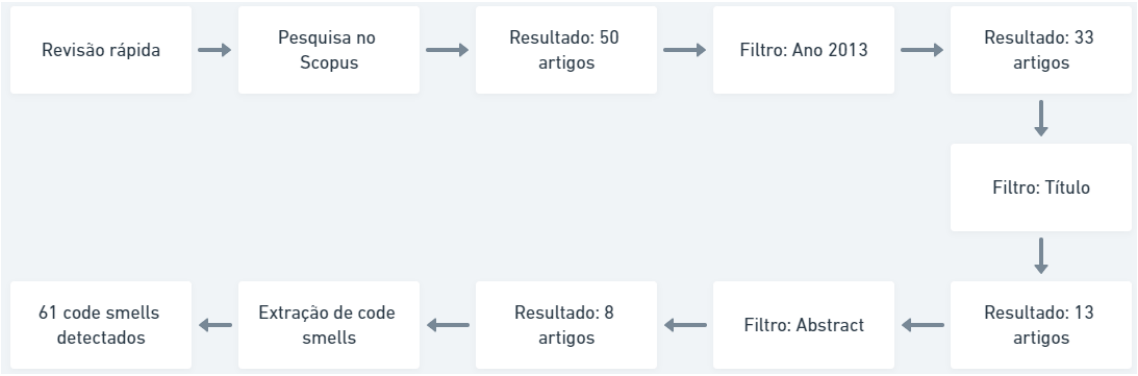


Figura 3. Processo abordado na Revisão Rápida

bases de dados indexadas a sua, ele conta com mais de 36.000 títulos em sua base, por isso, é uma das bases mais bem qualificadas para realizar pesquisas.

Foram realizados vários testes com diferentes strings a fim de encontrar quais os termos que melhor se adequar na busca por code smell. Os termos resultantes para realizar a busca foram:

("react" OR "reactjs" OR "react native") AND ("code smell" OR "bad smell" OR "anti-pattern" OR "bad practice" OR "code quality" OR "maintainability")

Os termos “react”, “reactjs” e “react native” foram usados para definir a tecnologia que seria investigada. Os termos “code smell”, “anti-pattern”, “bad practice” foram inseridos para descobrir o problema pesquisado e os termos “code quality” e “maintainability” foram usados a fim de encontrar o problema pesquisado por uma abordagem com foco na solução do problema.

O resultado da busca totalizou 50 artigos, nos quais precisaram ser filtrados a fim de atender o objetivo das perguntas. Foram aplicados 3 filtros, o primeiro filtro aplicado foi o filtro de artigos de até 2013, já que a tecnologia mais antiga é a React e ela data do ano de 2013, por isso limitamos a busca para esse ano. A quantidade total de artigos foi reduzida para 33.

O segundo filtro foi a leitura dos títulos dos artigos, com o intuito de remover os artigos que nada tinham a ver com o tema, como por exemplo o artigo com o título “Communicating the UX Vision: 13 Anti-Patterns That Block Good Ideas” que não possui relação com as tecnologias citadas. Após aplicar o filtro, o número total de artigos foi reduzido para 13.

O terceiro e último filtro foi o filtro da leitura do resumo, onde através dos resumos, definimos se o artigo havia informações relevantes sobre code smells. O número total foi reduzido para 8.

Para a extração dos code smells, realizamos a leitura completa dos 8 artigos selecionados. No entanto, todos os code smells identificados foram extraídos de apenas um

artigo. Após essa etapa de extração, identificamos um total de 61 code smells únicos. Mais detalhes podem ser encontrados em https://abrir.link/revisao-rapida.

3.2. Literatura Cinza

Para complementar os code smells encontrados, realizamos a revisão de literatura cinza. Essa metodologia permite incluir a literatura cinza (relatórios de empresas, documentos de trabalho, postagens de blogs e discussões em fóruns) em revisões de literatura, como recomenda [Garousi et al. 2019]. A literatura cinza deve ser incluída com o objetivo de fornecer uma visão mais completa do campo de estudo.

O processo de revisão de literatura cinza pode ser retratado na figura 4. O processo inclui 3 etapas: Busca no Google, seleção de fontes e extração dos code smells.

Figura 4. Processo de Revisão de Literatura Cinza

Para realizar a busca de fontes para a Revisão de Literatura Cinza utilizamos o Google. Foram utilizados várias strings diferentes para compor os termos de busca mais eficiente. Os termos utilizados foram:

("react" OR "reactjs" OR "react native") AND ("code smell" OR "bad smell" OR "anti-pattern" OR "bad practice" OR "code quality" OR "maintainability")

Utilizamos menos termos com o objetivo de ser mais eficaz nas buscas por fontes que retratem o nosso problema. O resultado totalizou cerca de 161.000 fontes. Devido à limitação de recursos humanos e tempo, decidimos aplicar filtros sobre essas fontes.

O primeiro filtro aplicado foi o filtro dos 120 links mais relevantes do Google. Em seguida, aplicamos o segundo filtro, que consistiu na leitura parcial das fontes, a fim de eliminar as que fugiam do nosso problema. Após a aplicação desse filtro, removemos 25 links, totalizando 95 restantes.

Foi aplicado também um terceiro filtro, que teve o objetivo de qualificar minimamente as fontes encontradas, então foi feito a leitura completa das fontes e a aplicação de um pequeno questionário de qualidade, como proposto pelo [Garousi et al. 2019].

O questionário era composto pelas seguintes perguntas:

P1) A organização editorial é respeitável?
P2) É um autor individual associado a um respeitável organização?
P3) O autor publicou outro trabalho na área?
P4) O autor possui expertise na área?
P5) A fonte tem um objetivo claramente declarado?
P6) A declaração nas fontes é a mais objetiva possível? Ou, a declaração é uma opinião subjetiva?

Após responder essas perguntas, o artigo era classificado como apto ou não para seguir para próxima etapa. Mais detalhes podem ser encontrada em https://abrir.link/literatura-cinza.

Após este filtro, a quantidade total de links foi 45. Em seguida, foi realizada a extração dos code smells desses links, onde encontramos 105 code smells citados. Desses, realizamos uma breve síntese para eliminar os duplicados, resultando em 60 code smells únicos.

3.3. Classificação e Síntese

Após finalizar a revisão de literatura, iniciamos o processo de síntese, a figura 5 mostra todo o processo de classificação e síntese dos code smells encontrados.

Esta etapa dividimos em 2 fases, a fase inicial e a fase completa. Na fase inicial apenas combinamos os code smells a fim de deixar únicos entre a revisão de literatura cinza e a revisão rápida, o resultado foram 97 code smells.

Na fase seguinte, agrupamos code smells que tinham nomes diferentes mas tratavam do mesmo cenário ou de um cenário muito semelhante, como por exemplo o code smell prop drilling e o prop plowing, que se transformou apenas no prop drilling, pois no contexto geral seria passar propriedades entre componentes apenas para eles repassarem para outros componentes. Foi nessa fase também que começamos a levantar se já existiam ferramentas que detectavam esses code smells e qual seria o grau de dificuldade para identificá-los dentro dessas ferramentas, esse levantamento foi muito importante porque deixou claro os code smells que tinham um grau muito complexo de implementação da sua detecção, ou que tinham um grau muito abstrato, como é o caso do code smell “incompatible props”.

Após esse levantamento, chegamos em um catálogo de 16 code smells, levando em consideração seu grau de dificuldade, sua complexidade e sua relevância, pois muitos eram abordados apenas quando o React usava classe em sua implementação. Também levamos em consideração a quantidade de menções na revisão de literatura, estabelecendo um mínimo de 2 menções para classificar como relevante. A tabela 1 mostra a quantidade de menções encontrada em cada um dos code smells selecionados.

Com isto respondemos à primeira pergunta sobre **quais são os principais code smells encontrados no React e React Native?**. Mais detalhes sobre o processo de classificação e síntese podem ser encontrados em https://abrir.link/sintese-classificacao.

Figura 5. Processo de Classificação e Síntese

A pergunta seguinte **com que frequência são encontrados em projetos reais?** foi respondida através do aperfeiçoamento de uma ferramenta de análise e detecção de smell, chamada ReactSniffer2, essa ferramenta será abordada na seção 5.

4. Catálogo de Code Smell

Nessa seção, apresentaremos um catálogo de 16 code smells, detectado através da nossa revisão. Fornecemos uma breve explicação e uma ilustração.

4.1. Props in Initial State

O code smell de "Props in Initial State" refere-se à prática de copiar valores de props ou propriedade para o estado inicial de um componente no React ou React Native. Por exemplo, ao passar a propriedade text diretamente para o useState (veja figura 6), o componente passaria a ignorar novas atualizações da propriedade text, o que tornaria mais propenso a bugs.

4.2. Use of index as key in rendering with loops

Este code smell trata-se de utilizar o índice como chave ao mapear elementos de uma lista para componentes React. Como por exemplo na figura 7, ao utilizar o índice do array "names" como chave de identificação do componente React, quando este precisar reordenar, adicionar ou remover um elemento, o React pode apresentar problemas de desempenho pela confusão dos índices atualizados, ou até bugs de não atualizar elementos.

Tabela 1. Tabela da quantidade de menções do code smells selecionados

Code Smell	Menções na RR	Menções na RLC	Total de Menções
Props in Initial State	2	9	11
Use of index as key in rendering with loops	1	9	10
Component Nesting/JSX Outside the Render	1	8	9
Large Components	1	6	7
Prop Drilling	1	4	5
Too many useState	2	3	5
Direct DOM Manipulation	1	3	4
Props Spreading	2	2	4
Deep Indentation	1	2	3
Too many props	1	2	3
Large useEffect	1	1	2
Mutable Variables	1	1	2
Procedural Patterns	1	1	2
String Literals	1	1	2
Never Using Class Components	2	0	2
Use PrevState	1	1	2

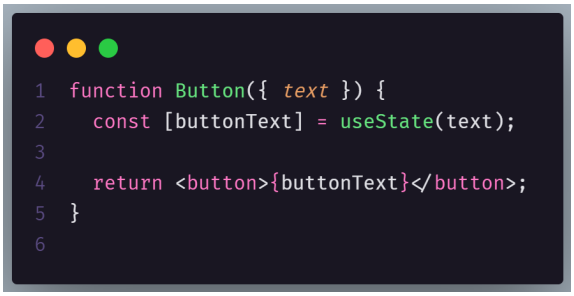


Figura 6. Exemplo de Props in Initial State

4.3. Component Nesting/JSX Outside the Render

É confiado o modelo de JSX para elementos de interface do usuário ao método render de um componente React em formato de classe ou ao return em um componente React em formato de função. Contudo, quando o JSX é extrapolado para outras regiões além dessas, é considerado um code smell, pois é um indício de que o componente está assumindo muitas responsabilidades, como no exemplo da figura 8, onde JSX referente ao avatar de um usuário está em uma constante fora do escopo do return. Portanto, torna-se mais difícil reutilizar esse componente em outros locais e dificulta sua legibilidade.

4.4. Large Components

O code smell “Large Components” ocorre quando um componente se torna muito grande e/ou complexo, devido à presença de várias responsabilidades ou funcionalidades no mesmo componente. Uma forma de medir isso é pela quantidade de funcionalidades, quantidade de funções ou até mesmo quantidade de linhas desse componente. Como na figura 9, onde o componente “LargeComponent” possui 2 regras complexas, tornando-o um componente complexo.

```
const names = ['Alberto', 'Bruno', 'Carlos', 'Davi'];

const NameList = () => {
  return (
    <ul>
      {names.map((name, index) => (
        <li key={index}>{name}</li>
      ))}
    </ul>
  );
};
```

Figura 7. Exemplo de Use of index as key in rendering with loops

```
const ProfileUser = () => {
  const avatar = (<img src={avatarUser} alt="avatarUser"/>)

  return (
    <div>
      <h1>{nameUser}</h1>
      {avatar}
    </div>
  );
}
```

Figura 8. Exemplo de Component Nesting/JSX Outside the Render

4.5. Prop Drilling

Esse code smell é comumente encontrado em projetos de programadores iniciantes no uso da tecnologia React ou React Native. O "Prop Drilling" é um termo usado quando se tem um componente A que precisa fornecer informações para um componente C e faz isso passando essas informações para o componente B, que por sua vez repassa para o componente C sem fazer uso algum dessas informações. Como no exemplo da figura 10, onde o "ComponentPropDrilling" recebe uma propriedade "props" e repassa essa propriedade para o componente filho "Component".

4.6. Too Many Usestate

O "Too Many Usestate" é um code smell que de forma similar ao "Large Component" ocorre quando o componente se torna complexo, precisando fazer o uso de muitos estados por meio do useState (hook fornecido pelo React). Ao utilizar muitos useState pode desencadear muitos efeitos colaterais caso estes também sejam usados como array de dependências de hooks useEffects (veja a figura 11).

4.7. Direct DOM Manipulation

O code smell "Direct DOM Manipulation" surge quando o desenvolvedor recorre à manipulação direta do Document Object Model (DOM) para modificar elementos HTML, em vez de utilizar as capacidades do React para gerenciar e renderizar os estados e atributos dos componentes, aproveitando o VirtualDOM do React (veja a figura 12).

4.8. Props Spreading

O "Props Spreading" é uma técnica usada no React para passar um conjunto de propriedades (props) para um componente. Isso é útil quando se quer realmente fazer uso de

```
const LargeComponent = () => {
  const complexBusinessLogic = () => {
    // linha 1
    // ... lógica complexa
    // linha 345
  }

  const anothercomplexBusinessLogic = () => {
    // linha 1
    // ... lógica complexa
    // linha 224
  }

  return (
    <View>
      <Text>{text}</Text>
    </View>
  );
}
```

Figura 9. Exemplo de Large Components

```
import {Component} from './path/to/component';

const ComponentPropDrilling = (props) => {
  return <Component title={props.title} />;
}
```

Figura 10. Exemplo de Prop Drilling

todas as propriedades no componente filho, mas o seu uso excessivo indica problemas de design e ineficiência, por isso é considerado um code smell (veja a figura 13).

4.9. Deep Indentation

O code smell "Deep Indentation" refere-se à componentes que possuem muitos níveis de indentação devido a muitas condicionais aninhadas. Isso torna o código ilegível e mais difícil de dar manutenção, exemplo disso é a figura 14, onde o componente "Component-DeppIndentation" possui várias condicionais para renderizar diferentes estados.

4.10. Too Many Props

Esse code smell ocorre quando o componente React ou React Native apresenta dependência com muitos dados externos, ou seja, o componente precisa de muitas propriedades para o seu uso. Isso pode ocasionar problemas de legibilidade (veja a figura 15).

4.11. Large UseEffect

O code smell "Large UseEffect" refere-se ao uso de um único useEffect com multiplas operações ou tarefas, resultando-o em um useEffect com muitas funcionalidades agrupadas. Esse padrão pode levar o código a ser confuso, dificultando sua legibilidade, e mais propenso a erros, pois seu array de dependências terá muitos atributos, complicando ainda mais os efeitos colaterais que desencadeiam a execução deste useEffect (veja a figura 11).

```
const ComponentTooManyState = () => {
  const [estado1, setEstado1] = useState('');
  const [estado2, setEstado2] = useState(0);
  // mais 20 estados ...
  const [estado21, setEstado21] = useState('')

  useEffect(() => {
    // linha 1
    // regras complexas
    // linha 342
  },[estado1, estado2, /* ... mais estados*/, estado21])

  return (
    <View>
      <Text>{text}</Text>
    </View>
  );
}
```

Figura 11. Exemplo de Too Many UseState

```
import React, { useEffect } from 'react';

function ComponentDirectDomManipulation() {
  useEffect(() => {
    const element = document.getElementById('div-DOM');
    if (element) {
      element.style.color = 'blue';
    }
  }, []);

  return <div id="div-DOM">Direct DOM Manipulation</div>;
}
```

Figura 12. Exemplo de Direct DOM Manipulation

4.12. Mutable Variables

O React oferece o hook useState para gerenciamento de estado nos componentes funcionais, que se atualiza e dispara re-renderizações com base nas mudanças desses estados. Utilizar variáveis mutáveis, como let ou var, para valores que podem ser alterados após a inicialização é considerado um code smell. Isto pode ocasionar a não re-renderização do componente quando este estado novo muda, o que pode levar a estados inconsistentes e dificuldades na manutenibilidade e compreensão do código.

4.13. Procedural Patterns

O code smell “Procedural Patterns” ocorre quando os desenvolvedores utilizam padrões de programação procedural, que são desencorajados pelo paradigma orientado a componentes, que é usado pelo React. Um exemplo disso é o uso de loops com o operador for para iterar um array, em vez de usar as funções do próprio array como map ou forEach.

4.14. String Literals

Este code smell refere-se ao uso excessivo de strings literais no código, principalmente quando estas mesmas strings são usadas em diversos lugares ou que pode mudar com o tempo. Este code smell pode tornar o código mais difícil de manter, pois pode ser necessária a mudanças em várias regiões do código. Por não gerar problemas de performance, esse code smell pode ser encontrado com mais facilidade.

```
const ComponentPropSpreading = (props) => {
  return <div {...props}>{props.children}</div>;
}
```

Figura 13. Exemplo de Props Spreading

```
const ComponentDeepIndentation = () => {
  return (
    <div>
      {condicao1 ? (
        <h1>Estado  1</h1>
      ) : condicao2 ? (
        <h1>Estado  2</h1>
      ) : condicao3 ? (
        <h1>Estado  3</h1>
      ) : null}
    </div>
  );
}
```

Figura 14. Exemplo de Props Spreading

4.15. Never Using Class Components

O code smell "Never Using Class Components" sugere que o uso de componentes de classe em React não é recomendado, embora tenha sido o método tradicional para renderizar componentes e gerenciar estados até a versão 16.8 do React. Desde então, o React introduziu hooks, permitindo que os componentes funcionais gerenciem o estado e todos os outros recursos do React sem a necessidade de classes. Isso resultou em código mais legível e com maior flexibilidade para adicionar novas funcionalidades através dos Hooks. Como por exemplo não é recomendado o uso de classe como na figura 16.

4.16. Use Prevstate

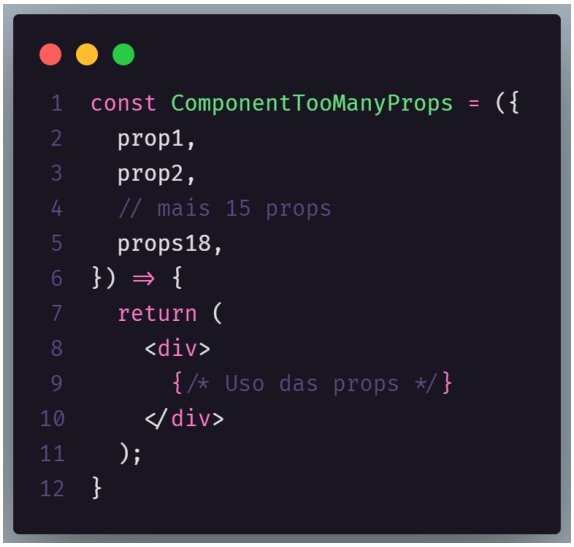
Quando se é necessário gerenciar um estado no React ou React Native, o hook de useState pode ser a solução, porém quando é preciso atualizar tendo a informação do estado atual, muitos desenvolvedores podem acabar caindo nesse code smell. O "Use Prevstate" é o code smell de quando não se utilizam o segundo argumento do hook useState, que é uma função que permite acessar o estado anterior durante a atualização.

Sem utilizar o segundo argumento, os desenvolvedores costumam pegar o estado diretamente do primeiro argumento do useState, o que pode levar a bugs quando há múltiplas atualizações rápidas, podendo em cada atualização não estar contido dentro do estado o seu valor mais recente (veja a figura 17).

5. Ferramenta de detecção

Para responder à pergunta **com que frequência são encontrados em projetos reais?**, aprimoramos uma ferramenta de detecção de code smells, chamada ReactSniffer2[8].

[8]https://github.com/lsm-5/reactsniffer2



```
const ComponentTooManyProps = ({
  prop1,
  prop2,
  // mais 15 props
  props18,
}) => {
  return (
    <div>
      {/* Uso das props */}
    </div>
  );
}
```

Figura 15. Exemplo de Too Many Props

```
class ComponentNeverUsingClassComponents extends React.Component {
  render() {
    return <h1>Texto</h1>;
  }
}
```

Figura 16. Exemplo de Componente React usando classe

Na figura 18 podemos conferir a arquitetura dessa ferramenta. Ela foi construída em Node e recebe como entrada o caminho do arquivo ou da pasta do projeto React ou projeto React Native através de uma Interface de Linha de Comando (CLI). Quando a ferramenta detecta e checa que o caminho é válido, ela começa a percorrer todas as pastas e subpastas filtrando se há arquivos com as extensões .js, .jsx, .ts, .tsx, que são os arquivos encontrados em projeto React ou React Native. Após essa filtragem, um compilador JavaScript, chamado Babel, entra em ação convertendo os arquivos em Árvore Sintática Abstrata (AST) resultando em formato JSON, esse é um compilador muito poderoso, pois consegue entender novos elementos de versões recentes do JavaScript e transformá-lo em uma versão compatível com versões anteriores do JavaScript. Após a geração da AST é feito um novo filtro, para selecionar apenas as AST que possuem importação para o React ou React Native. Por fim, é feito uma varredura nas AST's restantes percorrendo de forma recursiva usando um algoritmo de pré-ordem para detectar se existe code smells nos arquivos. Como output, a ferramenta gera um arquivo json contendo um resumo com todos os smells detectados, uma lista dos arquivos e componentes contendo os smells que cada um possuem e outra lista de todos os arquivos e todos os componentes informando se possuem smells ou não.

A ferramenta está disponível publicamente para uso através do gerenciador de pacotes NPM. Para mais detalhes acesse o link: https://www.npmjs.com/package/reactsniffer2.

A checagem do code smell "Props in Initial State" é feita através da verificação da propriedade que um componente recebe é usada como estado inicial.

```jsx
const ComponentUsePrevState = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    setCount(count + 1) // forma incorreta
    setCount(prevState => prevState + 1) // forma correta
  },[])

  return (
    <View>
      <Text>texto</Text>
    </View>
  );
}
```

Figura 17. Exemplo de Use PrevState da forma correta e incorreta

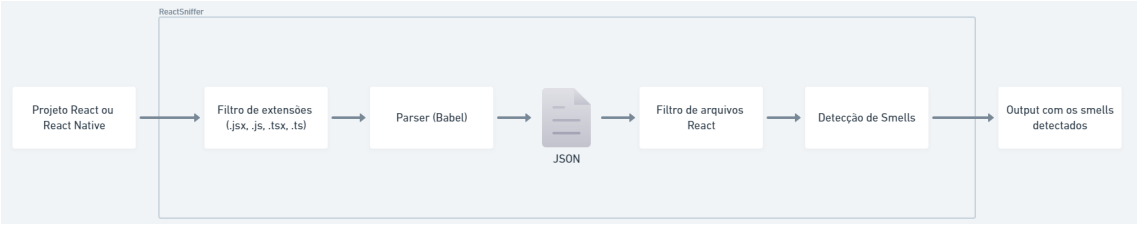


Figura 18. Arquitetura da ferramenta ReactSniffer2

Para o code smell “Use of index as key in rendering with loops” é verificado o uso do index como key para uma estrutura de repetição.

No “Component Nesting/JSX Outside the Render” a ferramenta identifica o número de componentes fazendo parte de outros componentes.

Para o code smell “Large Components” foi necessário criar um limite para a quantidade máxima de linhas encontrado em um componente e para a quantidade máxima de método desse componente, pois esses seriam os limites para considerar um smell.

Para “Prop Drilling” é analisado se a propriedade recebida por um componente é repassada pra outro componente.

Para o code smell “Too many useState” foi definido um limite para o número de uso do useState em cada componente para ser considerado um smell.

No “Direct DOM Manipulation” é verificado se há o uso de algum método de manipulação direta da árvore DOM.

No “Props Spreading” é analisado o uso do operador spread.

No “Deep Indentation” é feito a verificação de alguma indentação profunda.

Para o “Too many props” foi definido a quantidade máxima de propriedades que um componente poderia receber.

Para o code smell “Large useEffect” foi definido não o número de linhas como limite para virar um smell mas o número de operações dentro do uso do useEffect.

Para “Mutable Variables” é identificado se existem variáveis que podem ser alterada sem o uso de estado do React.

Para “Procedural Patterns” é analisado se faz o uso de funções próprias para interar arrays e coleções ao invés de usar estrutura de repetições.

Para “String Literals” é verificado se existem comparações com string literais no código fonte.

Para “Never Using Class Components” é identificado se o componente React foi construído como classe.

Para “Use PrevState” é verificado se as atualizações de estados dos componentes fazem o uso do parâmetros da própria função de atualização e não do uso do parâmetro de visualização do estado.

Para a detecção de alguns code smells foram necessário colocar limites para serem considerados smells. Os limites podem ser conferidos na tabela 2.

Tabela 2. Tabela da quantidade de menções do code smells selecionados

Code Smells	Métrica	Limite
Large Components	Linhas de Código do Componente	128
Large Components	Número de métodos	4
Too many useState	Número de useState	20
Too many props	Número de Props	13
Large useEffect	Número de operações	8

Os limites para os code smell “Large Components” e “Too many props” foram coletados da própria pesquisa da criação da ferramenta ReactSniffer [Ferreira and Valente 2022]. Os limites restantes dos code smells Too many “useState” e “Large useEffect” foram definidos de forma empiríca, pois não há fontes que sugiram limites para tais, entretanto foi questionado à um pequeno grupo de desenvolvedores que aprovaram esses limites.

6. Resultados

A tabela 3 mostra os resultados obtidos pela ferramenta ReactSniffer2 em projetos Open Source. Selecionamos os projetos baseados nos maiores números de estrelas resultantes da pesquisa de projetos React e React Native no Github. Como mostra a tabela 4, os projetos selecionados possuem um mínimo de 11.000 estrelas. E todos foram selecionados verificando se no arquivo packages possuíam a dependência react, como:

Selecionamos um total de 16 projetos React e React Native. A quantidade de arquivos analisados e dos componentes analisados pode ser encontrada na tabela 5. Discutiremos brevemente os resultados.

Props in Initial State (PIS): Aparece em 14 dos 16 projetos selecionados, ocorrendo em um total de 914 vezes (13,69%). O projeto com mais ocorrências foi o apache/superset com 47,81% (437) de todas as ocorrências.

```
"dependencies": {
    "react": "17.0.1",
    "react-dom": "17.0.1",
    "react-native": "0.63.2"
    // outras dependências ...
  }
```

Tabela 3. Tabela de resultado a análise da ferramenta ReactSniffer2 em projetos Open Source

Repositório	PIS	KRL	CN	LC	PD	TMS	DDM	PS	DI	TMP	LE	MV	PP	SL	NUCC	UPS
ant-design/ant-design-mobile	16	8	3	13	9	0	9	19	4	4	1	44	6	107	4	7
ant-design/ant-design-pro	0	0	0	3	0	0	2	2	0	0	0	0	0	12	0	1
chakra-ui/chakra-ui	17	10	3	4	3	0	3	90	0	0	0	1	0	39	2	13
facebook/docusaurus	103	20	2	6	2	0	17	151	1	0	0	3	0	55	12	1
atlassian/react-beautiful-dnd	4	0	71	15	22	0	0	3	0	0	0	0	0	43	66	0
pmndrs/react-spring	9	16	0	2	3	0	7	11	6	0	0	3	0	32	2	2
bvaughn/react-virtualized	111	0	80	25	8	0	3	11	4	0	0	47	11	59	40	0
apache/superset	437	22	450	173	265	1	39	257	22	21	1	245	13	462	99	28
pmndrs/zustand	9	2	3	0	0	0	2	3	0	0	0	0	0	0	3	0
react-native-elements/react-native-elements	46	19	32	25	32	0	0	94	6	13	0	22	0	90	9	6
GeekyAnts/NativeBase	63	5	2	16	37	0	1	264	20	2	0	68	2	120	0	28
react-native-maps/react-native-maps	0	1	120	15	0	0	0	2	2	0	0	25	3	26	64	1
danilowoz/react-content-loader	1	0	4	1	0	0	0	16	0	1	0	0	0	1	1	0
FaridSafi/react-native-gifted-chat	15	0	51	8	1	0	0	32	0	3	0	5	1	20	9	0
wix/react-native-navigation	11	2	100	42	3	0	2	7	0	0	0	3	0	56	107	0
callstack/react-native-paper	72	3	13	39	252	0	2	132	48	18	0	31	0	149	8	18
Total	**914**	**108**	**934**	**387**	**637**	**1**	**87**	**1094**	**113**	**62**	**2**	**497**	**36**	**1271**	**426**	**105**
Total em %	**13,69%**	**1,62%**	**13,99%**	**5,80%**	**9,54%**	**0,01%**	**1,30%**	**16,39%**	**1,69%**	**0,93%**	**0,03%**	**7,45%**	**0,54%**	**19,04%**	**6,38%**	**1,57%**

Use of index as key in rendering with loops (KLR): Com uma ocorrência baixa, apenas 108 vezes, aparece em 11 dentre todos os projetos selecionados. O projeto apache/superset concentra 22 de todas as ocorrências no total.

Component Nesting/JSX Outside the Render (CN): Está entre os smells mais frequentes encontrados, alcançando o 3° lugar em code smell mais frequentes. O projeto com mais ocorrências foi o apache/superset com 450 das 934 vezes. Também é neste projeto que encontramos o componente com mais componentes aninhados, o componente "CRUDCollection" possui 21 componentes aninhados.

Large Components (LC): Encontrado em 387 componentes, esse smell possui uma frequência de 5,80% dentre todos os smells. O projeto com o componente com a maior quantidade de linhas foi o projeto apache/superset, com um componente que possui 1495 linhas de código. E o projeto com o componente com o maior número de métodos é o bvaughn/react-virtualized com um componente com 36 métodos.

Prop Drilling (PD): Com um total de 637 vezes (9,54% entre todos os smells). O projeto com mais ocorrências é o apache/superset (265 ocorrências) seguido do projeto callstack/react-native-paper (252 ocorrências).

Too many useState (TMS): O smell com a menor frequência encontrada, ocorrendo apenas 1 vez de acordo com o nosso limite estabelecido, no projeto apache/superset. O componente em questão possui 21 estados internos.

Tabela 4. Tabela de quantidade de estrelas dos projetos selecionados

Repositório	Estrelas
ant-design/ant-design-mobile	11.300
ant-design/ant-design-pro	35.500
chakra-ui/chakra-ui	36.000
facebook/docusaurus	51.300
atlassian/react-beautiful-dnd	31.900
pmndrs/react-spring	27.100
bvaughn/react-virtualized	35.700
apache/superset	56.600
pmndrs/zustand	40.100
react-native-elements/react-native-elements	24.400
GeekyAnts/NativeBase	19.900
react-native-maps/react-native-maps	14.600
danilowoz/react-content-loader	13.300
FaridSafi/react-native-gifted-chat	13.000
wix/react-native-navigation	13.000
callstack/react-native-paper	11.900

Direct DOM Manipulation (DOM): Com uma ocorrência de 87 vezes (1,30%), este smell foi encontrado principalmente no projeto apache/superset (39 vezes) seguido do projeto facebook/docusaurus (17 vezes). Os smells eram encontrados principalmente com a manipulação direta da DOM através do uso dos métodos como: getElementsByTagName, getElementById, createElement, getElementsByClassName.

Props Spreading (PS): Com a ocorrência em todos os projetos, foi verificado 1094 vezes o uso do operador Spread. O projeto com o maior uso foi o projeto GeekyAnts/NativeBase com 264 usos. Este smell está em 2° colocado em smell mais frequêntes nos projetos.

Deep Indentation (DI): Com apenas 113 ocorrências, é classificado como um dos smells com baixa frequência. O projeto com mais ocorrências é o projeto callstack/react-native-paper com 48 ocorrências.

Too many props (TMP): Ocorrendo em 7 dos 16 projetos selecionados, esse smell teve uma frequência de 0,93% dentre todos os smells encontrados. O projeto com mais ocorrências foi o apache/superset (21 ocorrências), porém foi o projeto callstack/react-native-paper que tem o componente com o maior número de props (33 props).

Large useEffect (LE): Ocorrendo apenas 2 vezes, uma no projeto ant-design/ant-design-mobile e outra no projeto apache/superset. O projeto com o maior número de operações no useEffect foi o projeto ant-design/ant-design-mobile com 10 operações.

Mutable Variables (MV): Com uma ocorrência de 497 vezes, foi encontrada em 12 dos 16 projetos selecionados. O projeto com mais ocorrências foi o apache/superset, com 49,29% de todas as ocorrências.

Procedural Patterns (PP): Com apenas 33 ocorrências (0,54%), foi mais um dos smells com baixa frequência, com a ocorrência acontecendo principalmente no projeto

Tabela 5. Tabela da arquivos analisados e componentes analisados dos projetos selecionados

Repositório	Arquivos Analisados	Componentes Analisados
ant-design/ant-design-mobile	452	263
ant-design/ant-design-pro	12	16
chakra-ui/chakra-ui	258	439
facebook/docusaurus	357	435
atlassian/react-beautiful-dnd	205	131
pmndrs/react-spring	165	101
bvaughn/react-virtualized	65	54
apache/superset	1197	810
pmndrs/zustand	12	59
react-native-elements/react-native-elements	262	201
GeekyAnts/NativeBase	757	555
react-native-maps/react-native-maps	74	67
danilowoz/react-content-loader	35	17
FaridSafi/react-native-gifted-chat	52	35
wix/react-native-navigation	190	171
callstack/react-native-paper	280	236
Total	**4373**	**3590**

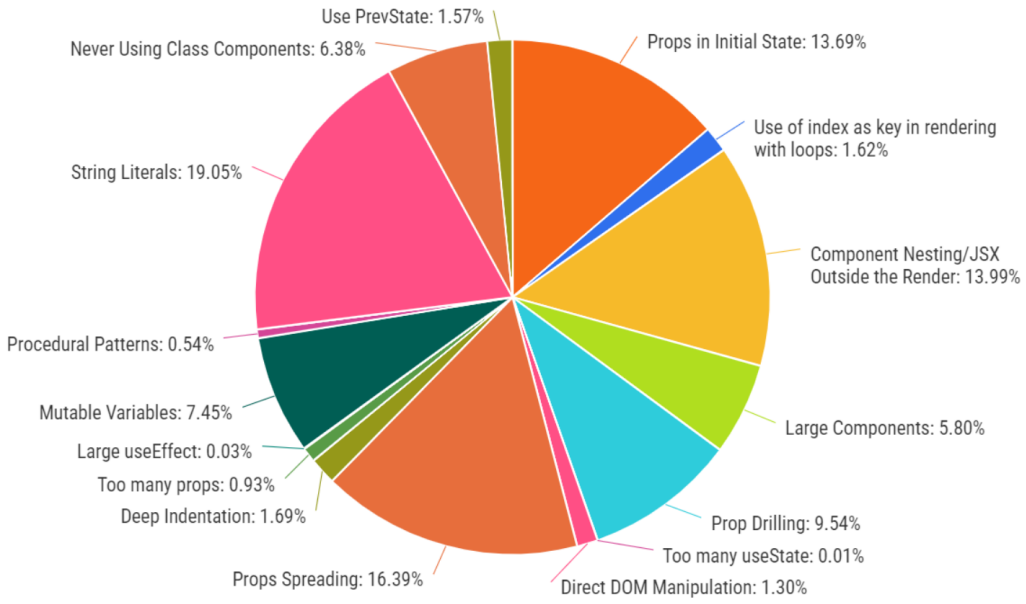
apache/superset (13 ocorrências).

String Literals (SL): Foi o smell com mais ocorrências (19,04%), mesmo não sendo encontrado em todos os projetos. Seu principal foco foi no projeto apache/superset com 462 ocorrências.

Never Using Class Components (NUCC): Foi encontrado em 426 componentes dos 3590 componentes analisados. Seu smell teve uma frequência de 6,38% dentre todos os smells encontrados.

Use PrevState (UPS): Encontrado 105 vezes, seu smell teve uma frequência de 1,57% dentre os smells. O projeto com mais ocorrência foram os projetos apache/superset e GeekyAnts/NativeBase. Ambos com o mesmo número de ocorrência (28 vezes).

Figura 19. Gráfico da frequência dos smells selecionados



Para mais detalhes acesse: https://abrir.link/resultados

7. Ameaças à validade

Nesta seção apresentaremos as ameaças à validade da nossa metodologia. A revisão rápida pode ser considerada uma técnica leve em relação a uma revisão sistemática da literatura completa. Já a revisão de literatura cinza tem suas ameaças devido às fontes não serem científicas. Além disso, pela revisão ser conduzida por apenas um pesquisador, há a possibilidade de existir viés de seleção.

No entanto, em relação à revisão rápida, utilizamos uma base de dados que agrega outras bases de dados para tornar a seleção mais abrangente. Já em relação à revisão de literatura cinza, a seleção das primeiras 120 fontes pode levar ao descartes de fontes significativas para nossa pesquisa, entretanto, é comum ao pesquisar no Google que as páginas iniciais correspondam melhor ao que se procura. Outro ponto que pode elevar minimamente a qualidade da revisão é o uso do questionário seguindo as sugestões da própria metodologia.

Para a seleção dos code smells, selecionamos apenas aqueles que foram mencionados mais de uma vez. Finalmente, o resultado também mostra que os code smells são encontrados em projetos Open Source.

Os valores usados como limites podem ameaçar a validade desse artigo, porém 3 deles foram coletados de outro artigo científico e os coletados empiricamente foram questionados à desenvolvedores experientes que também aprovaram o valor definido.

Um último ponto é que alguns code smells selecionados, como por exemplo o "Props Spreading", não necessariamente implicam em uma má performance do sistema ou em uma dificuldade de legibilidade ou manutenção. No entanto, a proposta do catálogo é alertar aos desenvolvedores que o uso deles pode ser a raiz de um problema maior no futuro do código, que vale a reflexão durante o seu uso.

8. Conclusão

Neste artigo, buscamos explorar a análise e detecção de code smell em aplicações React e React Native, com o propósito de compor um catálogo dos principais smells encontrados na nossa revisão de literatura. Através da nossa análise, identificamos que o catálogo de smells se mostrou consistente pois foram encontrados todos os smells em projetos Open Source. Os projetos com os maiores números de smells foram: apache/superset (2535 smells), callstack/react-native-paper (785 smells) e GeekyAnts/NativeBase (628 smells), porém vale ressaltar que o seu grande número de smells também reflete no grande número de arquivos e componentes que esses projetos possuem.

Outro ponto importante é que a identificação de alguns smells não necessariamente implica na prática ruim de programação, mas que o seu uso excessivo pode implicar em problemas de performance e legibilidade, como por exemplo o smell: "string literals" e "props spreading". Vale ressaltar também que os limites estabelecidos podem e devem ser ajustados para compor cada vez mais a realidade dos projetos de mercado.

As conclusões apresentadas neste estudo não apenas contribuem para a compreensão atual sobre os code smells em aplicações React e React Native, mas também abrem caminho para futuras pesquisas e desenvolvimentos na área, além de aprimoramento em ferramentas de lint. O importante desse estudo é o alerta que fica aos desenvolvedores e

programadores, para que tenham uma maior reflexão antes de uso dessas práticas em suas aplicações.

9. Trabalhos Futuros

Como continuação desse estudo, esperamos que a ferramenta ReactSniffer2 possa ser ainda mais aprimorada, englobando mais code smells, como os que ficaram de fora da nossa análise, e também possa incluir novos frameworks e bibliotecas, como: Angular, Vue e etc. Para garantir que os aprimoramentos não prejudiquem os smells detectados atualmente, ReactSniffer2 possui um sistema de testes com exemplos de code smells que são válidos e inválidos. O que torna ReactSniffer2 mais atrativo para novas contribuições assim como esse estudo o fez.

Também esperamos que, à medida que novas ferramentas de lint surjam ou que existentes se aprimorem, elas possam se basear nos smells analisados e detectados neste estudo para melhorar sua eficácia. Sugerimos a realização de integrações entre IDEs e ferramentas de lint, a fim de proporcionar uma experiência de desenvolvimento mais eficiente. Além disso, propomos a realização de estudos de caso em aplicações de grande escala, bem como a análise e detecção de smells de código em mais projetos, para avaliar a eficácia em ambientes práticos e realistas.

Referências

Cartaxo, B., Pinto, G., and Soares, S. (2018). O papel das revisões rápidas no apoio à tomada de decisão na prática de engenharia de software. In *EmEASE'18*.

Ferreira, F. and Valente, M. T. (2022). Detecting code smells in react-based web apps. *Information and Software Technology*.

Fowler, M. (2006). Codesmell. `https://martinfowler.com/bliki/CodeSmell.html`. Acesso em 2023-06-14.

Garousi, V., Felderer, M., and Mäntylä, M. V. (2019). Diretrizes para incluir literatura cinza e conduzir revisões de literatura multivocal em engenharia de software. *Tecnologia da Informação e Software*.

Hunt, P. (2013). Why did we build react? Acessado em: 16 de fevereiro de 2024.

Paré, G., Trudel, M.-C., Jaana, M., and Kitsiou, S. (2015). Synthesizing information systems knowledge: A typology of literature reviews. *Information & Management*, 52(2):1–15.

Ramos, M., Valente, M. T., and Terra, R. (2018). Angularjs performance: A survey study. *IEEE Software*, 35(2):72–79.

Svahnberg, M. (2003). *Supporting software architecture evolution*. PhD thesis, Blekinge Institute of Technology, Sweden.