Detecting Code Smells in React-based Web Apps

Fabio Ferreira[1,2] and Marco Tulio Valente[1]

[1]Department of Computer Science, UFMG, Brazil
[2]Center of Informatics, IF Sudeste MG - Campus Barbacena, Brazil

Abstract

Context: Facebook's React is a widely popular JavaScript library to build rich and interactive user interfaces (UI). However, due to the complexity of modern Web UIs, React applications can have hundreds of components and source code files. Therefore, front-end developers are facing increasing challenges when designing and modularizing React-based applications. As a result, it is natural to expect maintainability problems in React-based UIs due to suboptimal design decisions. **Objective:** To help developers with these problems, we propose a catalog with twelve React-related code smells and a prototype tool to detect the proposed smells in React-based Web apps. **Method:** The smells were identified by conducting a grey literature review and by interviewing six professional software developers. We also use the tool in the top-10 most popular GitHub projects that use React and conducted a historical analysis to check how often developers remove the proposed smells. **Results:** We detect 2,565 instances of the proposed code smells. The results show that the removal rates range from 0.9% to 50.5%. The smell with the most significant removal rate is Large File (50.5%). The smells with the lowest removal rates are Inheritance Instead of Composition (IIC) (0.9%), and Direct DOM Manipulation (14.7%). **Conclusion:** The list of React smells proposed in this paper as well as the tool to detect them can assist developers to improve the source code quality of React applications. While the catalog describes common problems with React applications, our tool helps to detect them. Our historical analysis also shows the importance of each smell from the developers' perspective, showing how often each smell is removed.

Keywords— JavaScript, React, Anti-Patterns, Code Smells, Maintainability, Software Design.

1 Introduction

According to the StackOverflow Survey,[1] JavaScript is the world's most popular programming language for the eighth year in a row[2]. Moreover, in the last three years, HTML and CSS emerged in second place[3]. Essentially, this popularity reflects the importance and prevalence of modern Web-based systems. Besides programming and markup languages, front-end frameworks—such as Angular, React, and Vue —are also relevant tools for building rich and responsive Web applications, usually called Single-Page Applications (SPAs) [1, 2]. Their popularity on the StackOverflow Survey[4] also illustrates this importance. For example, both Vue and React are more popular than traditional MVC-based web frameworks, such as Spring, Django, and Ruby on Rails.

JavaScript front-end frameworks provide abstractions—usually called components—for structuring and organizing the codebase of modern Web UIs. Thus, developers can modularize the UI into independent and reusable elements and reason about each one in isolation [3]. These components can also be reused in other pages and applications.

However, due to the complexity of SPAs, the front-end layer of a modern application can have hundreds of components and source code files. As a result, it is natural to expect that wrong, or suboptimal design decisions will eventually lead to code that is hard to maintain, understand, modify, and test. Although traditional code smells describe general problems in object-oriented design [4], we still lack studies investigating design problems specific to web-based systems implemented using JavaScript front-end frameworks. Since an important part of the developers are front-end developers, this is a topic that deserves further investigation. For example, according to the StackOverflow Survey, 27% of the developers classify themselves as front-end developers. Particularly, we identify the following problems in this context: (1) the lack of a code smells catalog and vocabulary that can be shared among practitioners when documenting and discussing possible design problems related to these frameworks; (2) the lack of detection tools that can warn developers about these smells; and (3) the lack of empirical studies on the prevalence of these smells.

To fill these gaps, in this paper, we propose a catalog of React code smells. We focus on React because it is currently the most popular JavaScript front-end framework [5].[5] Specifically, we ask the following research questions:

RQ1: What are the most common code smells when using React?

RQ2: How common are the identified React smells in open-source projects?

RQ3: How often are the identified React smells removed?

[1] https://insights.stackoverflow.com/survey/2020#most-popular-technologies

[2] However, we acknowledge these results might change according to the ranking and data sources. For example, in the May 2022 version of TIOBE — another popular ranking of programming languages – Python is the most popular language and JavaScript appears at the 7th position.

[3] Indeed, Stack Overflow ranks programming, scripting, and markup languages together. For this reason, HTML and CSS appear in their ranking.

[4] https://insights.stackoverflow.com/survey/2020#technology-web-frameworks

[5] https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/

To answer the first RQ, we identify a list of React smells by conducting a grey literature review and by interviewing six professional React developers. Then, to check whether the identified smells are common in open-source systems—and therefore to answer the second RQ—we first implement a prototype tool, called ReactSniffer, that detects the smells in JavaScript React-based applications. Using this tool, we detected 2,565 code smells in the top-10 most popular GitHub projects that use React. Finally, we conducted a historical analysis to check how often developers remove the proposed smells. We found that Large File has the most significant removal rate (50.5%). The smells with the lowest removal rates are Inheritance Instead of Composition (IIC) (0.9%), and Direct DOM Manipulation (14.7%).

To the best of our knowledge, and as our key contribution, we are the first to **propose a list of code smells for React-based JavaScript applications**. For example, previous studies identified code smells in pure JavaScript [6], HTML [7], Cascading Style Sheets (CSS) [8], Elixir [9], and MVC frameworks [10], but did not focus on smells specific to the usage of JavaScript front-end frameworks. Therefore, our contributions are threefold:

- We propose a catalog of 12 code smells for React, including five novel smells and one partially novel smell. They were proposed after carefully analyzing 52 documents that were collected in a grey literature review and by interviewing six professional software developers. This catalog can help front-end developers to better spot and fix design problems in the React components they are responsible for.

- We implement a prototype tool, called ReactSniffer, to detect the proposed code smells. This tool works as a complement to the proposed catalog, by supporting the automatic identification of the smells proposed in our work. It is also publicly available as a NPM package.[6].

- We use ReactSniffer to unveil the most common code smells in React-based systems. Moreover, we also show how often developers remove these smells. These studies were important to show the proposed smells indeed occur in real-world React-based projects. For example, we found smells in all systems in our dataset (10 systems, in total). To complement, we also showed how the removal rate varies among the smells in our catalog (since it is not enough to show the number of occurrences of the proposed smells, but also to show how frequently they are removed).

The remainder of this paper is organized as follows. In Section 2, we present background information on React. In Section 3, we present the methodology we used to define our catalog of React code smells. In Section 4, we present our catalog of smells. For each smell, we provide its definition and an illustrative example. We detail our code smell detection tool, ReactSniffer, in Section 5. Section 6 presents a field study when we use ReactSniffer in ten GitHub projects. In

[6]https://www.npmjs.com/package/reactsniffer

Section 7, we present the results of a historical analysis and show how often developers remove these smells. In Section 9, we discuss and put our findings and insights in perspective. In Section 10, we detail threats to validity. Finally, we present related work in Section 11 and conclude in Section 12.

2 React in a Nutshell

Released by Facebook in 2013, React is a JavaScript library for building user interfaces that does not enforce any architectural pattern (e.g., MVC). Instead, the library prioritizes interoperability, i.e., it can be incorporated into a system without rewriting existing code, therefore allowing gradual adoption.

React provides a syntax extension to JavaScript called JSX (JavaScript XML) for writing UI elements. Thus, instead of separating technologies by using markup and logic in separate files, React separates concerns under loosely coupled modularization units called *components* containing both logic and markup.

A React component can have parameters, called *props* (short for *properties*) and returns a React element via the `render` method representing what should appear on the UI. React also supports different kinds of components. The main ones are *class* and *function* components. The key difference between *class* and *function* is that the latter is just a JavaScript function that accepts props as an argument and returns JSX. There is no `render` method in functional components, and they usually do not have state. That is, a functional component is by definition a `render` method. For example, the following codes show the same component implemented using *class* component (see Listing 1) and *function* component (see Listing 2).

```
class Comment extends React.Component {
  render() {
    return (
        <div>
          <div>{this.props.text}</div>
          <div>{this.props.date}</div>
        </div>
    );
  }
}
```

Listing 1: Example of Class Component.

React components support one-way data binding. Therefore, when data changes in the model, React updates and re-renders the components in the view. r

```
function Comment(props) {
  return (
    <div>
      <div>{props.text}</div>
      <div>{props.date}</div>
    </div>
  );
}
```

Listing 2: Example of Function Component.

3 Methodology

Although JavaScript front-end frameworks foster reuse and modularity, developers may also make design decisions that lead to code that is hard to maintain, understand, modify, and test. However, we still lack studies investigating these design problems in web-based systems implemented using JavaScript front-end frameworks. For this reason, in this section, we describe the methodology we use to derive a catalog of React code smells. We collect the smells following two instruments: a grey literature review (Section 3.1) and semi-structured interviews with professional software developers (Section 3.2).

3.1 Grey Literature Review

In this initial step, our goal is to reveal bad design practices when implementing React applications. Since React emerged in recent years, there are few studies on it, and to the best of our knowledge, we are the first to study code smells in React-based systems. Therefore, as usual with emerging and technological topics, grey literature is recommended as a source of evidence rather than formal literature reviews since practitioners are the key protagonists in terms of using this new technology (and sharing their experiences in blogs, QA forums, and ebooks) [11, 12, 13, 14].

Thus, we conduct a grey literature review to answer our first research question:

RQ1: What are the most common code smells when using React?

Figure 1 summarizes the procedure we followed for selecting the articles and identifying the smells, which has four main steps: (1) Google search, (2) article selection, (3) data extraction, and (4) data validation. In the rest of the section, we describe each step.

Google search: To retrieve an initial list of articles, we defined a search query on code smells related to React. We made two primary considerations when defining this query. First, we added the terms *react* and *reactjs* to restrict the search to React-related articles. Furthermore, we added *code smell*, *bad smell*, *anti-pattern* and *bad practice* to search for design problems related to this framework. As a result, we used the following search query:

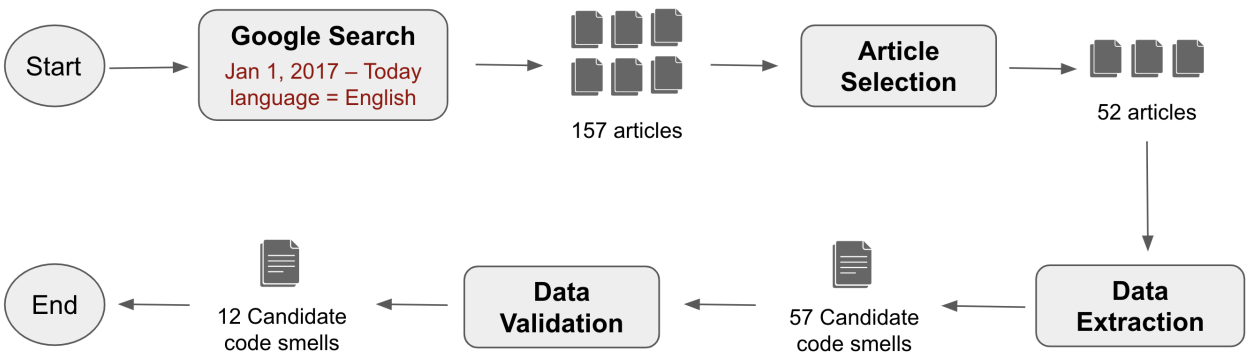

Figure 1: Overview of the grey literature methodology

```
("react" OR "reactjs") AND
("code smell" OR "bad smell" OR "anti-pattern" OR
"bad practice")
```

Then, we executed this query in the Google Search Engine. The first result included thousands of documents. Since JavaScript front-end frameworks are a recent and emerging technology, we set 2017 as the initial search date, and we only considered articles written in English, resulting in 157 articles.

Article selection: Since the grey literature is not peer-reviewed, practitioners can share their experiences without rigorous methodological concerns. For this reason, we rely on additional assessment levels to make sure the selected articles are appropriate to our purposes and to meet minimum qualification levels. In particular, we assessed the articles using the Quality Assessment Checklist proposed by Garousi *et al.* [15]. We selected articles that attend at least one of the following Authority of the Producer criteria from this checklist: (a) is the publishing organization reputable? (b) is an individual author associated with a reputable organization? (c) has the author published other works in the field? (d) does the author have expertise in the area?

As mentioned, we initially selected 157 articles, including React's official documentation.[7] Then, we sequentially removed articles that do not include a valid URL (9 articles), that are a copy of another selected article (5 articles), that are about humans reactions to chemical smells, i.e., aromas (6 articles), that do not meet our quality assessment levels (22 articles) or that do not discuss or answer our research question (63 articles). After discarding these articles, we ended up with 52 articles for analysis (157 - 105 = 52), which we refer to as A1 to A52.

Data extraction: After collecting the articles, the first author of this paper carefully read and analyzed each one. He used thematic analysis for identifying and recording patterns (or *themes*) related to code smells within the selected articles [16, 17]. A tabular data extraction form was used to keep track of the extracted information. In particular, each row of this form reports an article, and each column corresponds to a candidate code smell.

[7] https://reactjs.org/docs/getting-started.html

Table 1: Code smells identified in the grey literature

React Smell	Description	Articles	#
Props in Initial State	Initializing state with props	A1, A3, A6, A14, A15, A16, A17, A24, A25, A26, A27, A36, A41	13
Large Component	Component with too many props, attributes, and/or lines of code	A2, A5, A7, A9, A13, A14, A18, A21, A27, A40, A44, A47	12
Inheritance Instead of Composition	Using inheritance to reuse code among components	A1, A12, A25, A26, A29, A35, A36, A38, A40, A49	10
Prop Drilling	Passing properties through multiple levels of a components hierarchy	A1, A2, A12, A19, A29, A37, A45, A49	8
JSX outside the render method	Implementing markup in multiple methods	A2, A3, A7, A25, A44, A45, A51	7
Direct DOM Manipulation	Manipulating DOM directly	A7, A20, A28, A30, A31, A52	6
Duplicated Component	Code duplication among components	A2, A32, A40, A46, A48	5
Multiple Components in the same file	Multiple and unrelated components implemented in the same file	A7, A35, A46, A48	4
Too Many Props	Passing too many properties to a single component	A3, A44, A46	3
Uncontrolled Component	A component that does not use props/state to handle form's data	A1, A20, A26	3
Force Update	Forcing the component or page to update	A1, A34	2
Low Cohesion	Component with multiple responsibilities	A5, A32	2

Data Validation: In the data validation step, the second author analyzed each article to validate each candidate code smell retrieved during the data extraction step. Initially, we identified 57 candidate smells. However, in this validation step, we decided to consider only smells cited in more than one article, resulting in 22 candidate smells. Next, we discussed each candidate smell and removed nine cases that describe low-level concerns, i.e., unrelated to design. Particularly, all the removed smells are detected by state-of-the-practice linter tools (e.g., *modify state directly*, *array index as key*, *no access state in* `setState`, *props spreading*, etc.). We also removed the smell *Derived state from props* because it is related to another one, *Props in initial state*.

We finished with 12 candidate smells, which are summarized in Table 1. The complete list of the selected articles and candidate smells is available at `https://doi.org/10.5281/zenodo.6985604`.

3.2 Semi-Structured Interviews

We interviewed six professional React developers recruited through the first author's social networks to validate the candidate smells identified in the grey literature review. In these interviews, we asked the participants to comment on bad design practices they observed while working with React. Our ultimate goal was to double-check whether the smells identified in the grey literature indeed occur in real software projects.

The interviewed participants have from two to six years of experience with React. They work with industrial projects from distinct organizations, ranging from startups to big tech companies. The first author conducted the interviews, taking from 27 minutes (minimum) to 43 minutes (maximum). We report data about each participant in Table 2.

Table 2: Participants experience

Participant	React Experience	Industry
P01	6 years	E-commerce
P02	4 years	E-commerce
P03	4 years	IT services
P04	4 years	IT services
P05	2 years	Banking
P06	2 years	Banking

We started the interviews by asking the following question: *What are the main bad practices you observed while working with* React*?* Thus, the participants had the freedom to comment on problems not previously identified in the grey literature.

In the second part of the interview, we provide concrete examples of the smells identified in the grey literature that the participants did not mention. We also asked them whether they view these examples as design problems. In case of a positive answer, we asked whether it is common to find the smell in projects they worked on.

As a result, we were able to validate 11 out of 12 candidate smells, as described in Table 3. Particularly, we were not able to validate Multiple Components in the Same File. Two developers did not view this smell as a clear problem, as expressed in the following comment:

> *Working with too many open files can also be harmful. If the components are closely related, I do not see it as a problem, it is a developer preference.*

We also identified three new smells in the interviews: Large Bundlers, Lack of Accessibility, and Large Files, as summarized in Table 4. In this case, we selected Large Files to include in the catalog. We did not include the first two smells because they do not appear in our grey literature review.

Therefore, our catalog ended up with 12 React Smells. In Section 4, we provide examples of each one.

3.3 Code Smells Classification

After selecting the final smells, we also classified them into three major categories:

Novel Smells. We claim our list includes five novel smells: Force Update, Direct DOM Manipulation, Uncontrolled Components, Props in Initial State, and JSX Outside the Render Method.[8] They refer to features very specific to React, including View updates, components that manipulate DOM directly, components that refer to HTML elements, components'

[8] JSX outside the render method are usually Large Components as well. Indeed, 324 out of 401 JSX smells (80.7%) detected in our Field Study (Section 6) — occur in Large Comments.

Table 3: Code smells validated in the interviews, with examples of participants' comments

React Smell	**Participants' Comments**
Props in Initial State	*When we initialize a state with props, the component practically ignores all updated values of the props. If the props values change, the component would still render its first values.*
Direct DOM Manipulation	*React beginners usually use plain vanilla JavaScript to direct access a HTML DOM element, which can cause inconsistencies between React's virtual DOM and the real DOM.*
Prop Drilling	*If you have a grandparent component, which passes props to the child, which passes props to the grandchild, but only the grandchild needs these props, the mistake is to deliberately pass props down instead of using a contextAPI.*
Duplicated Component	*It is common to find components or parts of a component duplicated, especially if you do not have documentation such as a storybook.*
Inheritance Instead of Composition	*Using inheritance makes it difficult to reuse components elsewhere.*
JSX outside the render method	*It is a bad practice. Class methods with JSX should be components, also for reuse purposes.*
Too Many Props	*It is also common, and the problem is worse in* React *because every change in a prop generates a new request to update the view.*
Large Component	*When a component grows, it is a sign that it needs to be broken.*
Force Update	*This is a terrible practice, I have never used force update, but in [my-company] there is everywhere.*
Low Cohesion	*Components doing a lot. That is what we have the most.*
Uncontrolled Components	*It is common among inexperienced developers or when migrating to REACT to use uncontrolled components.*

Table 4: New code smells identified in the interviews

React Smell	**Description**
Large Bundles	A JS bundler is a tool that puts JS code and all its dependencies together in one JS file. A large JavaScript bundle contains several components, dependencies, utility libraries and so on.
Lack of Accessibility	Dynamic components transformed to HTML non-semantic elements, which can not be interpreted reliably by a wide variety of user agents, including assistive technologies.
Large Files	A file with several components and lines of code

state, and render methods.

Partially Novel Smell. We classified a single smell in our catalog as partially novel: Prop Drilling. For example, Ousterhout *et al.* [18] mention a design red flag called Pass-Through Methods, i.e., a method that does nothing but only pass its arguments to another method with a similar signature. However, in our case, Prop Drilling does not relate to methods but to components. As a second observation, Prop Drilling resembles to some degree a Middle Man class, i.e., a class that delegates most of its work to other classes [4]. However, Middle Man classes by definition are anemic in the

terms of behavior and data, which is not the case of Prop Drilling. In other words, a complex component can also be used to pass through properties to its child components.

Traditional Smells. In the grey literature and in the interviews with developers, we identified six smells that are similar to traditional smells: Large File, Large Component (which can be considered a particular case of the traditional Large Class smell), Inheritance instead of Composition (since the inverse relation is considered a good object-oriented principle [19]), Duplicated Component (which is a particular case of Duplicated Code), and Low Cohesion. We also consider that Too Many Props is similar to Data Class and Large Class [4]. A Data Class has mostly data and only setter and getter methods. A Large Class is a class with several responsibilities. On the other hand, Too Many Props designates a component with a large number of props. Essentially, this smell is very similar to the well-kwon Long Parameter List smell, proposed by Fowler [4]. Just to clarify, in React, properties designates the parameters passed into components.

> *Summary:* We identify a list of 12 React smells by conducting a grey literature review and interviewing six professional React developers: Force Update, Direct DOM Manipulation, Uncontrolled Components, Props in Initial State, JSX Outside the Render Method, Large File, Large Component, Low Cohesion, Prop Drilling, Too Many Props, and Inheritance instead of Composition .

4 React Code Smells

In this section, we present our catalog of smells. For each smell, we provide its definition and an illustrative example. A more detailed presentation of each smell is available at: `https://github.com/fabiosferreira/React-Code-Smells`.

4.1 Prop Drilling

Props drilling refers to the practice of passing props through multiple components in order to reach a particular component that needs the property. Therefore, the intermediate components act only as bridges to deliver the data to the target component. For example, consider an App to create a `Gallery` that renders images and allows comments in each one (see Figure 2). First, the `App` renders `Gallery` passing the `user` and `avatar` props. Then, `Gallery` renders `Image` passing the `user` and `avatar` props. Next, the `Image` renders the `Comment` components, also passing the `user` and `avatar` props. Finally, `Comment` renders `Author`, passing again the `user` and `avatar` props. In other words, `Author` is the only component that really needs to use these props.

As the size of the codebase increases, Prop Drilling makes it challenging to figure out where the data is initialized, updated, or consumed. Since each component is usually in a separate file, in our Gallery example (see Figure 2), there are five different files to check for property updates,

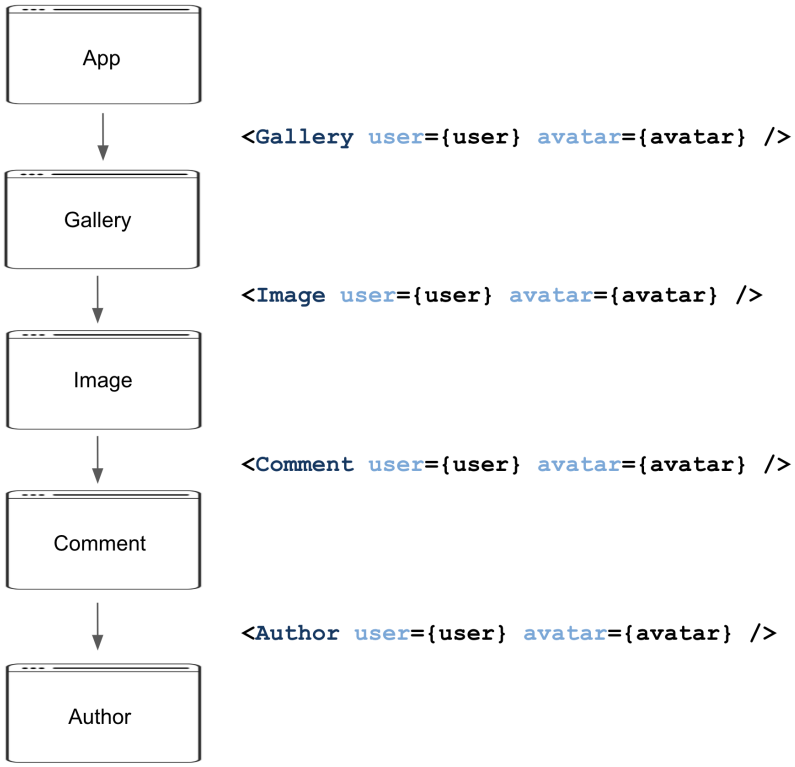

Figure 2: Example of Prop Drilling (code smell)

including, Author.jsx, Comment.jsx, Image.jsx, Gallery.jsx, and the file that calls the Gallery component, App.jsx. Moreover, it seems that current IDEs do have help on the task of tracking property initialization and updates. As a second problem, Prop Drilling results in tightly coupled components. For example, whenever the Author component needs more props from the top, all the intermediate levels must be updated. Alternatives to Prop Drillings include component composition and Context API. However, at least Context API has similar problems, for example, regarding the difficulty in tracking property updates. Finally, it is worth mentioning that React documentation recommends using component composition:

> *If you only want to avoid passing some props through many levels, component composition is often a simpler solution than context.*

4.2 Duplicated Component

This smell refers to almost identical components. For example, the problem usually occurs when multiple developers extract the same UI code to different components. As an example, in the Listing 3, `Comment` and `Opinion` have the same code.

```
function Comment(props) {
  return (
    <div>
```

```
      <User user={props.user} />
      <div>{props.text} </div>
      <div>{props.date}</div>
    </div>
  );
}
```

```
function Opinion(props) {
  return (
    <div>
      <User user={props.user} />
      <div>{props.text} </div>
      <div>{props.date}</div>
    </div>
  );
}
```

Listing 3: Example of Duplicated Component (code smell).

4.3 Inheritance instead of Composition

In React, developers tend to use inheritance to tackle two problems: (1) to express containment relations, particularly when a component does not know its possible children; (2) to express specialization relations, when components are "special cases" of other components. However, as usual in object-oriented design, React developers recommend using composition over inheritance whenever possible.

For example, consider a component that handles `Employee` and a special collaborator called `Developer`, which has a `level` and receives a `bonus`. The Employee component can share props with `Developer`, and a possible design to show the data consists of creating a generic view to show data common to all employees and reuse it in the view that handles `Developer`

However, inheritance usually results a tight coupling between components [19]. For example, changes in the base component can affect all child components. On the other hand, by using composition instead of inheritance, we can reuse only the UI behavior. Moreover, the article A40 comments about this smell: *We're fans of composition over inheritance in pretty much any programming language. It's tempting to treat your component's render method as a template method.*. Listing 5 shows the `Developer` component using composition instead of inheritance, which produces the same result.

In some cases, it may be necessary to share non-UI functionality between components, for which React documentation recommends using separate JavaScript modules:

> *If you want to reuse non-UI functionality between components, we suggest extracting it into a separate JavaScript module. The components may import it and use that function, object, or a class, without extending it.*

```
class Employee extends React.Component {
   render() {
      return (<div> Name: {this.props.name} </div>);
   }
}
```

```
class Developer extends Employee {
   render() {
      return (
         <div>
            {super.render()}
            <div> Level: {this.props.level} </div>
            <div> Bonus: {this.props.bonus} </div>
         </div>
      )
   }
}
```

Listing 4: Example of the Code Smell Inheritance instead of Composition.

```
class Developer extends React.Component {
   render() {
      return (
         <div>
            <Employee name="Bob"/>
            <div> Level: {this.props.level} </div>
            <div> Bonus: {this.props.bonus} </div>
         </div>
      )
   }
}
```

Listing 5: Example of composition instead of inheritance.

4.4 JSX outside the Render Method

The `render` method is the only method that is mandatory in a class component. It provides the JSX template for UI elements. Normally, we assume that all JSX code is confined in the `render` method. Therefore, the existence of JSX code in other methods indicates that the component is assuming too many responsibilities and providing a complex UI element. As a result, it is more difficult to reuse the component in other pages or apps. For example, in the following `Gallery` component (see Listing 6), we have three `render` methods: one to render an `Image` (that calls `renderComment()`), the method that only renders `Comments`, and the main render method (that calls `renderImage()`). However, suppose we should handle just a `Comment`. In this case, it is impossible to rely on `Gallery` since this component also provides other visual elements, such as the ones needed to display an `Image`.

```
class Gallery extends React.Component {
   renderComment() {
      return (<div> ... </div>)
   }

   renderImage() {
      return (
        <div>
            ...
            {this.renderComment()}
        </div>)
   }
   render() {
      return (
         <div>
            {this.renderImage()}
            ...
         </div>
      )
   }
}
```

Listing 6: Example of JSX outside the Render Method (code smell).

In summary, the presence of JSX code in multiple methods indicates a complex UI element, which might be decomposed into smaller and reusable ones.

4.5 Too Many Props

Props (or properties) are arguments passed to components via HTML attributes. However, it is hard to understand components with a long list of props. For example, consider the `Comment` component used as an example in React's documentation (see Listing 7). This component has many properties, including the four properties presented in the following code:

```
function Comment(props) {
  return (
    <div>
      <div>{props.name}</div>
      <img src={props.avatarUrl}/>
      <div>{props.text} </div>
      <div>{props.date}</div>
      // other props
    </div>
  );
}
```

Listing 7: Example of Too Many Props (code smell).

To reduce the number of props handled by `Comment`, we can extract the props related to avatars (i.e., `name` and `avatarUrl`) to a new component, called `Avatar`. After that, `Comment` just need to reference this new component, as shown in the Listing 8:

```
function Comment(props) {
  return (
    <div>
      <Avatar avatar={props.avatar} />
      <div>{props.text} </div>
      <div>{props.date}</div>
    </div>
  );
}
```

Listing 8: Example used to reduce the number of props.

4.6 Force Update

JavaScript frameworks rely on a data binding mechanism to keep the View layer updated with data automatically. Particularly, React supports one-way data binding, which automatically reflects model changes in the View. For this reason, it is considered a bad practice to force the update of components or even reload the entire page to update some content, as recommended in React documentation: *normally, you should try to avoid all uses of forceUpdate() and only read from this.props and this.state in render().*

4.7 Uncontrolled Components

Forms are key elements in Web UIs. The official React documentation recommends using controlled components to implement forms. In such components, React fully handles the form's data. However, developers sometimes implement forms using vanilla HTML, where the form data is handled by the DOM itself, leading to so-called Uncontrolled Components. For example, the following `AddComment` component (see Listing 9) is considered Uncontrolled, since it uses a `ref` to get the comment value.

On the other hand, when developers use controlled components, all data is stored in the component's state, making it easy to validate fields instantly or render them conditionally.

4.8 Low Cohesion

As usual in software design, the implementation of components must be cohesive and follow the Single Responsibility Principle [20]. In other words, the component's data and behavior must be related to make the component reusable and easy to understand.

```
class AddComment extends React.Component {
  // ...
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>Comment:</label>
        <input type="text" ref={this.input}  />
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Listing 9: Example of Uncontrolled Component (code smell).

4.9 Large Files

A large file is one with several components whose implementation requires several lines of code. Indeed, some developers advocate that we should have exactly one component per file. See for example the following comment from one the interviewed developers:

> *I got tired of trying to fix a bug on a component co-located with other six components in a file with thousands of lines.*

4.10 Large Component

Clean and small components improve readability and maintainability. For example, React documentation provides this recommendation for refactoring large components:

> *If a part of your UI is used several times, or is complex enough on its own, it is a good candidate to be extracted to a separate component.*

4.11 Direct DOM Manipulation

React uses its own representation of the DOM, called virtual DOM, to denote what to render. When props and state change, React updates the virtual DOM and propagates the changes to the real DOM. For this reason, manipulating the DOM using vanilla JavaScript can cause inconsistencies between React's virtual DOM and the real DOM.

4.12 Props in Initial State

Initializing state with props makes the component to ignore props updates. If the props values change, the component will render its first values. React documentation also states about this smell:

Using props to generate state in the constructor (or `getInitialState`) often leads to duplication of "source of truth", for example where the real data is. This is because the constructor (or `getInitialState`) is only invoked when the component is first created.

5 ReactSniffer: Code Smell Detection Tool

We also implemented a prototype tool—called ReactSniffer—to detect the proposed smells. In this section, we describe its architecture (Section 5.1), thresholds selection policy (Section 5.2), and limitations (Section 5.3).

5.1 Architecture

As described in Figure 3, ReactSniffer architecture has two key components: a parser for analyzing React files and a Smells Detector module for identifying the smells.

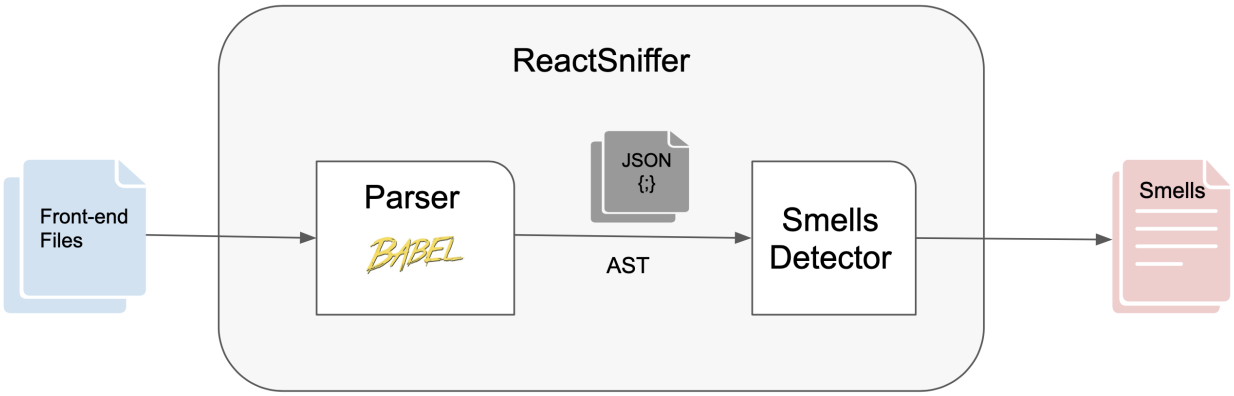

Figure 3: ReactSniffer architecture

The Parser is a Command-Line Interface (CLI) implemented in Node, which receives as input a valid front-end file and generates an Abstract Syntax Tree (AST) in a JSON format. We filter front-end files according to the extensions listed in Table 5.The principal element of this parser is Babel,[9] a JavaScript compiler commonly used to convert novel JavaScript syntactic elements (e.g., ES6 syntax) into backward-compatible elements, therefore allowing programs to run in older browsers. A powerful component of Babel is its parser,[10] which generates an AST for JSX code.

Table 5: Front-end files extensions

`*.js, *.jsx, *.ts, *.tsx`

The Smells Detector module relies on the AST to search and inspect React elements. It recursively traverses the AST using a preorder algorithm. To better understand ReactSniffer

[9] https://babeljs.io/
[10] https://babeljs.io/docs/en/babel-parser

architecture, Listing 10 shows the pseudocode of the ReactSniffer tool and Listing 11 details the major steps and computations performed by ReactSniffer to detect smells in components.

```
ReactSniffer(dirname)
    smells = []
    FrontEndFiles = all files in dirname with .js, .jsx, .ts, and .tsx extensions
    FOREACH file IN FrontEndFiles DO
        AST = GenerateAST(file)
        IF isReactFile(AST) THEN
            components = getComponents(AST)
            FOREACH component IN components DO
                smells = DetectComponentSmells(component)
        IF (file[LOC] > LOC_F_th OR components.length > N_Components_th
           OR file[N_Imports] > N_Imports_th) THEN
            smells.add(LF);
    return smells
```

Listing 10: ReactSniffer algorithm

Next, we discuss the smells supported by our tool and which detection strategy we implemented to recognize their instances.

JSX OUTSIDE THE RENDER METHOD: we can create methods in class components using two different syntaxes: traditional or arrow functions. Thus, we rely on two types of nodes to check whether a component implements UI features outside its `render` method: `ClassMethod` and `ClassProperty`. For `ClassProperty` nodes, we also verify whether they receive an ArrowFunction. The detection strategy considers the number of methods containing UI elements, which we call *NM_JSX*. If *NM_JSX* is higher than a given threshold, the component is tagged as a smell.

TOO MANY PROPS: To detect this smell, we count the component's number of props, which we call *N_Props*. If this value is higher than a threshold, the component is marked as a smell.

LARGE COMPONENT: To detect this smell, we rely on the number of lines of code (*LOC*), the number of props (*N_Props*), and the number of methods (*NM*) in the component. We then check whether at least one of these values is greater than the given thresholds (each metric has its own threshold).

FORCE UPDATE: To detect components that force updates, we check whether its functions include calls to `forceUpdate()` or `reload()` methods.

DIRECT DOM MANIPULATION: to detect components that manipulate the DOM directly, we check whether they include calls to any HTML DOM methods, such as `getElementById` and `getElementsByTagName`.

PROPS IN INITIAL STATE: we check whether the state is initialized with any component props to detect this smell.

LARGE FILE: To detect large files, we rely on the number of lines of code (*LOC*), the number of

```
DetectComponentSmells(Component)
    smells = []
    IF (component[LOC] > LOC_th OR component[N_Props] > N_Props_th or component[NM] > NM_th) THEN
        smells.add(LC)
    IF (component[N_Props] > N_props_th) THEN
        smells.add(TP)
    IF (component.hasNode('SuperClass') OR component.hasNode('super')) THEN
        smells.add(IIC)
    IF (component.hasNode('forceUpdate') OR (component.hasNode('reload')) THEN
        smells.add(FU)
    IF (component.hasAnyDOMManipulationNode()) THEN
        smells.add(DOM)
    IF (component.hasInputWithoutState()) THEN
        smells.add(UC)
    FOREACH method IN component[methods] DO
        IF (method.hasJSXElement()) THEN
            smells.add(JSX)
        IF (method == constructor AND method.hasPropsInState()) THEN
            smells.add(PIS)
    return smells
```

Listing 11: Smell detection algorithm

components (*N_Components*), and the number of imports (*N_Imports*) in a file. We then check whether at least one of these values is greater than the given thresholds (each metric has its own threshold).

Inheritance instead of Composition: We use the `SuperClass` AST node to detect inheritance relations. If a component has the `SuperClass` node and it is not a default React component, it is inheriting from another component.

Uncontrolled Components: to detect uncontrolled components, we check whether components have inputs that values are not binding with a state.

5.2 Benchmark-based Thresholds Selection

As described, most heuristics used by ReactSniffer rely on thresholds. Therefore, to define these thresholds, we propose the usage of a benchmark-based approach [21, 22], when the thresholds are derived from a dataset of systems. In this case, we need a dataset of systems (more details in Section 6.1). Next, we first must compute the respective metrics for all systems and then use the 90th percentile value of each metric as a threshold. For example, the same threshold is used by Alves *et al.* [21] to characterize very-high risk code. On the other hand, Aniche *et al.* [10] use a threshold of 75% (third quartile) in the context of traditional MVC frameworks. However, since this is the first study on React-based smells, we decided to be more conservative in our thresholds selection policy.

5.3 Limitations

Currently, ReactSniffer does not detect the following code smells: Low Cohesion, Duplicated Component, and Prop Drilling. The reason is that these smells require a more complex implementation or metric selection. For example, usually, there is a lack of consensus on recommended cohesion metrics [23, 24]. Regarding Duplicated Component and Prop Drilling, their detection depends on a more sophisticated static analysis. We plan to support these three smells in future versions of our tool.

5.4 Availability

ReactSniffer is publicly available on GitHub and can be easily installed via the NPM package manager. More details about the tool and installation are available at `https://www.npmjs.com/package/reactsniffer`.

6 Field Study

Our catalog emerged from the analysis of 52 documents from the grey literature and was validated with six professional React developers. Despite that, we argue it is important do check whether the identified smells indeed happen in the wild. Moreover, it is also important to provide quantitative data about the frequency of each smell described in the catalog, since it is not reasonable to assume that they appear in the same number in real-world projects. For this reason, this section reports the results of a field study conducted to investigate whether the proposed React smells are common in open-source systems.

First, we formulated the following research question:

RQ2: How common are the proposed React smells in open-source projects?

We start by creating a dataset of GitHub projects that use React (Section 6.1) and by using ReactSniffer to search for smells in these projects (Section 6.2).

6.1 Dataset

We intend to validate our catalog of smells with relevant GitHub projects. Thus, as the first step for creating a dataset with React clients, we searched for the names that identify React in two popular package managers: npm and bower. First, we randomly selected 10 GitHub projects that are React clients (using GitHub's Used-by feature). Then, we inspected their package.json and bower.json files.[11] We concluded that both npm and bower refer to React dependencies using the string "react", as in this file:

[11] We needed to perform this inspection because the Used By feature does not allow filtering relevant projects. Moreover, the feature only works in projects that explicitly enable the dependency graph computation.

```
"dependencies": {
    "react": "17.0.1",
    "react-beautiful-dnd": "13.0.0",
    "react-diff-viewer": "^3.1.1",
    ...
}
```

Next, we selected the top–15,000 most popular GitHub projects ranked by stars, a metric commonly used to rank projects by popularity [25, 26]. Then, we discarded 541 projects labeled as archived by GitHub since these projects are no longer actively maintained. We also searched for forks to ensure the selected projects are not a clone of someone else's project. However, we did not find forks among the selected projects. Finally, for the remaining 14,459 projects (15,000 - 541), we checked out their package.json and bower.json files to retrieve their dependencies. Then, we selected only projects that depend on React. In total, we found 988 of such projects.

From the 988 projects, we selected the top-10 projects by stars after manually discarding non-software projects and projects that use React only in examples, tutorials, documentation, and tests. Table 6 shows the number of stars, number of front-end files (FF), and number of components of the selected projects. We analyzed 2,060 front-end files and 2,695 React components.

Table 6: Dataset (FF: number of front-end files; Comp: number of components)

Project	**Stars**	**FF**	**Comp.**
grafana/grafana	47,629	914	1116
apache/superset	45,230	387	438
prometheus/prometheus	41,650	34	46
RocketChat/Rocket.Chat	31,970	532	815
ant-design/antd-design-pro	31,661	19	20
carbon-app/carbon	29,936	62	103
mastodon/mastodon	29,746	203	222
joplin/joplin	28,799	52	52
metabase/metabase	27,881	1159	1344
getredash/redash	20,736	295	352
Total	-	3,657	4,508

For detecting smells, ReactSniffer relies on a benchmark-based threshold selection policy. Therefore, we computed the required thresholds before running the tool, as previously explained in Section 5.2. As a result, we obtained the values in Table 7, which are then used to obtain the results presented in the following section.

6.2 Results

Table 8 details the number of instances of each smell, as we found in our dataset. We briefly discuss the results for each one.

Table 7: Thresholds selection

React smell	**Metric**	**Threshold**
JSX outside the render method	NM_JSX	2
Too many props	N_Props	13
Large Component	LOC	116
	N_Props	13
	NM	2
Large File	LOC	225
	N_Components	2
	N_Imports	19

Inheritance Instead of Composition (IIC): We found five projects reusing UI elements by means of inheritance instead of composition, with 20 occurrences in total. The project joplin/joplin concentrates ten occurrences.

JSX Outside the Render method (JSX): this is a common smell in our dataset: nine out of ten projects have at least one occurrence, with 401 occurrences in total. In relative terms, 8.89% of the components in our dataset have methods implementing JSX outside the render method.

Too many Props (TP): 10.07% of the components contain too many props, according to our thresholds. In total, we found 454 occurrences of this smell, distributed over nine projects. The component with the highest number of props has 131 props. Interestingly, this same component (`DashboardContainer`) has 790 LOC, seven methods with JSX outside the render method, and inherits props from `StatefulUIElement`, which in turn inherits from `UIElement`.

Large Files (LF): all projects have at least one large file. Grafana is the project with the highest number, both in relative and absolute terms. According to our thresholds, out of 914 React files in Grafana, 265 files (28.9%) are large. The largest file in our dataset has 1,413 LOC.

Large Components (LC): all projects have large components. Specifically, 726 (16.10%) out of 4,508 components are considered large components. The largest component has 1,089 LOC.

Props in Initial State (PIS): We found four projects initializing their state with props, with 22 occurrences in total. The project apache/superset concentrates 11 occurrences.

Direct DOM Manipulation (DOM): we found five projects with this smell and 26 occurrences in total. The HTML DOM methods they call are `getElementById`, `createElement`, and `getElementByClassName`.

Uncontrolled Component (UC): we found five projects with this smell and eight occurrences in total. They usually use a `ref` HTML attribute to read/update form values directly from the DOM instead of using React features to handle this kind of data. The Listing 12 illustrates one

Table 8: Code smells by project (LC: Large component, TP: Too many props, IIC: Inheritance instead of Composition; PIS: props in Initial State; DOM: Direct DOM manipulation; JSX: JSX outside the render method; FU: Force update; UC: Uncontrolled component; LF: Large File)

Project	LC	TP	IIC	FU	DOM	JSX	UC	PIS	LF
grafana/grafana	202	101	5	26	7	136	1	4	265
apache/superset	129	84	0	4	5	56	1	11	152
prometheus/prometheus	9	5	0	0	0	5	0	0	6
RocketChat/Rocket.Chat	70	61	0	0	0	0	0	0	156
ant-design/ant-design-pro	4	4	0	2	0	4	0	0	5
carbon-app/carbon	17	7	0	0	2	2	2	0	19
mastodon/mastodon	77	38	1	2	7	78	2	0	35
joplin/joplin	35	28	10	1	0	29	1	0	16
metabase/metabase	138	100	2	3	5	65	0	5	164
getredash/redash	45	26	2	3	0	26	0	2	44
Total	726	454	20	41	26	401	8	22	867

of these occurrences where a `ref` attribute is used to clear the value of a file input:

```
const clearModal = () => {
    //...
    if (fileInputRef && fileInputRef.current) {
      fileInputRef.current.value = '';
    }
};

<input ref = {fileInputRef} type= "file" ... />
```

Listing 12: Example of Uncontrolled Component found by the ReactSniffer Tool.

When we handle form data directly, we assume the responsibility of keeping the form (UI element) and the component's state synchronized. In other words, instead of relying on React's one-way data binding mechanism for handling this synchronization, we assume the burden to handle this task.

Force Update (FU): we found seven projects that force updates and 41 occurrences in total. A single project (grafana) concentrates 26 occurrences of this smell. We also manually analyzed each case. Interestingly, in one case, we found a comment self-admitting a technical debt (SATD), as shown in the Listing 13:

```
// Angular HACK: Since the target does not actually change!
this.forceUpdate();
```

Listing 13: Example of Force Update found by the ReactSniffer Tool.

We also discovered that grafana migrated to React after using Angular since October 2017. Indeed, the project still has some parts implemented in Angular. Therefore, the use of

`forceUpdate` seems to be a hack to allow a rapid migration instead of fully reimplementing the UI according to React best practices.

> *Summary:* Using ReactSniffer, we detected 2,565 code smells in the top-10 most popular GitHub projects that use React. The smells with the highest number of occurrences are Large Component, Too Many Props, JSX outside the render method, and Large File.

7 Historical Analysis

In the first RQ, we relied on a grey literature review and on interviews to come up with a catalog of 12 code smells for React-based systems. In the second RQ, we provided quantitative data on the frequency of each smell. This is important, for example, to provide guidance to developers on how often they should expect to find each smell in their projects. However, a key information is still missing in the previous RQs, i.e., the removal rate of each smell. For example, a smell can be very common but rarely removed. Therefore, this indicates that developers tend to see this smell as less harmful. On the other hand, smells that are rapidly removed from the code tend to be more important.

Therefore, this section reports the results of a historical analysis conducted to investigate how often developers remove the proposed React smells in open-source systems. First, we formulated the following research question:

RQ3: How often are the identified React *smells removed?*

We start by checking out the repository of the projects in our dataset and defining three times frames of six months for our analysis: Aug-2020 to Jan-2021, Feb-2021 to Jul-2021, and Aug-2021 to Feb-2022, as illustrated in Figure 4. For each time frame, we downloaded the repository versions at the time frame start and end dates and used ReactSniffer to search for smells. Then, we computed the smells identified at the time frame start date that were removed at the time frame end date.

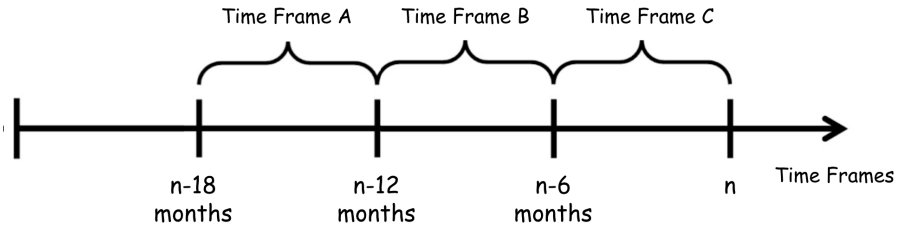

Figure 4: Time frames used in the analysis

To facilitate the visualization and analysis, we group the results of all projects by time frame. Figure 5 shows the overall removal rates.

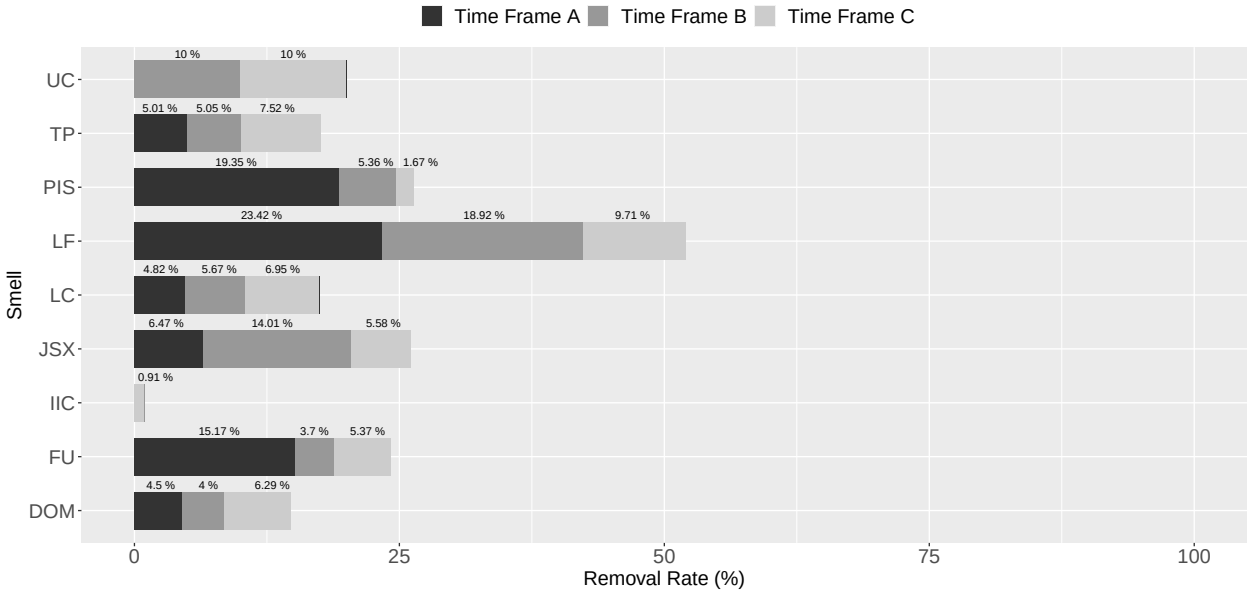

Figure 5: Removal rates of each smell by time frame (LC: Large component, TP: Too many props, IIC: Inheritance instead of Composition; PIS: props in Initial State; DOM: Direct DOM manipulation; JSX: JSX outside the render method; FU: Force update; UC: Uncontrolled component; LF: Large File)

The removal rates range from 0.9% to 50.5%; the smell with the greatest removal rate is Large File (50.5%). In fact, all projects contain refactorings for large files. On the other hand, the Inheritance instead of Composition smell has the lowest removal rate (0.91%), and only one project contains a removal of Inheritance Instead of Composition (IIC).

However, we also have other smells with a significant removal rate, such as Props in Initial State (26.3%), JSX outside the render method (26.0%), and Force Update (24.2%). For example, Figure 6 shows a developer from grafana project questioning the use of Force Update before it was refactored.

Finally, the removal rates of the other smells are Direct DOM Manipulation (14.7%), Large Component (LC) (17.4%), and Too many props (17.5%). Four projects contain removal of Direct DOM Manipulation. Figure 7 shows a refactoring removing an occurrence of this last smell with a comment self-admitting a technical debt.

Summary: The removal rates range from 0.9% to 50.5%. The smell with the highest removal rate is Large File (50.5%). The smells with the lowest removal rates are Inheritance Instead of Composition (IIC) (0.9%), and Direct DOM Manipulation (14.7%).

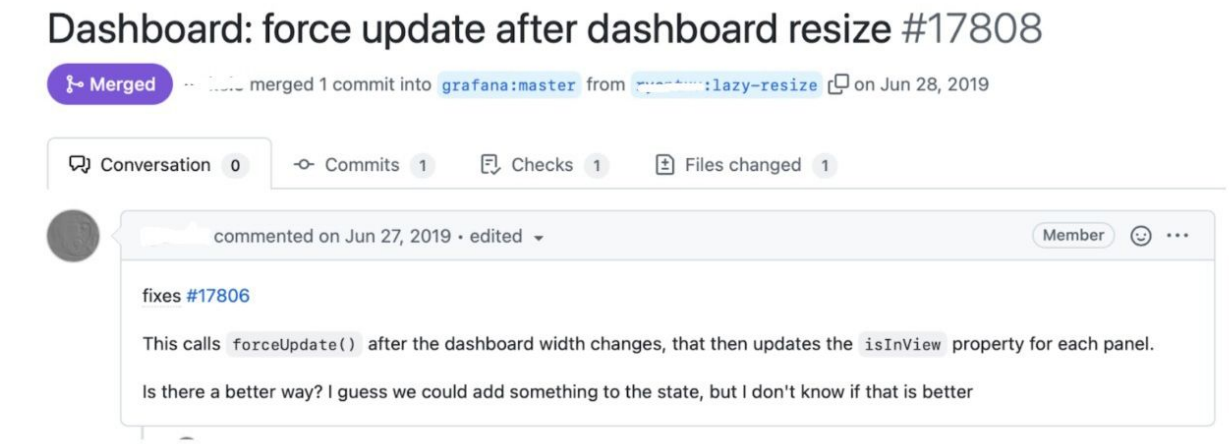

Figure 6: Issue questioning the use of Force Update

```
72  -   // TODO: Something less hacky than createElement to help TypeScript / AntD
73  -   getContainer = () => this.containerRef.current || document.createElement('div');
```

Figure 7: Direct DOM Manipulation Refactoring

8 Validation with Developers

To validate ReactSniffer's results, we recruited an experienced React Developer to execute the tool in one of his company's projects using the same thresholds presented in Table 5.2. We also asked him to evaluate the tool results. Table 9 shows the number of front-end files (FF), and number of components of the selected project.

Table 9: Dataset (FF: number of front-end files; Comp: number of components)

Project	**FF**	**Comp.**
E-commerce project	189	181

For this particular validation, we instrumented ReactSniffer to generate as output a csv file containing detailed information about each smell, such as the smell name, the smell file path, and the line of code where the smell was detected. We also added a column to this csv file, asking the developer to rate the relevance of each smell in a 5-point scale, where 1 means the smell is not important at all and probably will not be refactored and 5 means the smell is very important and therefore it should be refactored in the near future. In the last column, the developer was invited to add comments about each detected smell, in case he found it relevant.

ReactSniffer detected 157 smells instances. However, the developer did not evaluate 24 of such smells for two key reasons: 15 smells occurred in deprecated code (i.e., code responsible for features that are not used anymore in the project) and nine smells occur in code required by third-party modules (i.e., the developer sees this code as external to their project).

Figure 8 shows the number of instances of each remaining smell. The most common smell was Large Component (LC), with 45 instances (28%). In the other extreme of the char, there are no occurrences of the Inheritance Instead of Composition (IIC) and Uncontrolled Component (UC) smells.

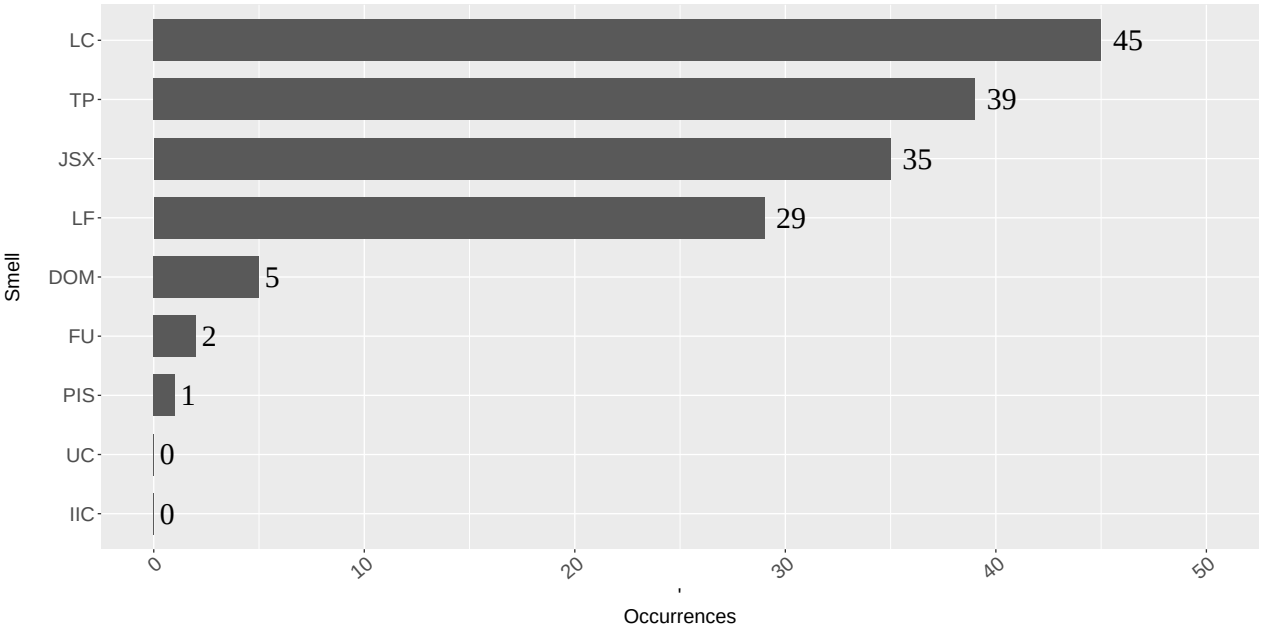

Figure 8: ReactSniffer Validation with Developers (LC: Large component, TP: Too many props, IIC: Inheritance instead of Composition; PIS: props in Initial State; DOM: Direct DOM manipulation; JSX: JSX outside the render method; FU: Force update; UC: Uncontrolled component; LF: Large File)

Table 10 shows the average relevance scores for each smell, as answered by the contacted developer. All smells have an average relevance score greater than 3.3. The exception is JSX Outside the render method (JSX), with an average score of 2.2. The developer justified that most JSX occurrences are correlated and caused by two other smells: Large Component (LC) and Too Many Props (TP). For example, he added the following comment to justify some JSX instances:

> *As the components were very large, developers decided to separate some chunks of JSX into non-render methods to organize the code. Then in the render, they would only invoke these methods.*

Table 10: Relevance scores of ReactSniffer Evaluation

	LC	**TP**	**DOM**	**JSX**	**LF**
Average	3,4	3,3	3,5	2,2	3,7

The developer considered the smells Force Update (FU) and Props in Initial State (PIS) severe. However, they are also in third-party modules and will not be refactored. For example, he

added the following comment to justify these smells instances:

> *They are grave and should be rated five. However, they are all in third-party modules, so they will not be refactored, as it is not the project's fault.*

Summary: ReactSniffer detected 157 instances of the proposed smells in a commercial React-based project. According to an experienced developer, the relevance of such smells in a 5-point scale range from 2.2 (JSX Outside the Render Method) to 3.7 (Large File).

9 Discussion

Code smells are a widely studied concept, not only in object-oriented designs but also in particular domains, such as MVC-based applications [10], JavaScript [6], HTML [7], and CSS [8]. Therefore, a key discussion relates to the novelty of smells we identified for React systems. In this regard, in Section 3.3 we have classified our smells into three major categories: Novel, Partially Novel, and Traditional ones.

Another discussion refers to the small number of some smells, as detected in the field study. Although they indicate well-known problems in React apps, we detected few cases of Direct DOM Manipulation, Uncontrolled Components, and Inheritance instead of Composition. However, this might be explained by the fact that our dataset is composed of popular projects, which might follow more strict programming and design guidelines.

Particularly, it is not easy to provide the precise reasons for the low removal rate of Inheritance instead of Composition. However, we hypothesize two main reasons for this result: (a) inheritance is a widely-known object-oriented mechanism; therefore, developers that already use inheritance in mainstream object-oriented languages may keep using this mechanism in their React-based projects; (b) inheritance leads to a tight-coupling between subclasses and superclasses; however, this coupling does not necessarily cause maintenance problems, including bugs, particularly when the classes are responsible for stable and rarely changed requirements. For example, 14 out of 20 instances of the Inheritance instead of Composition smell occur in files that have not been changed in the time frame of our study.

Since we focused on React-based Web systems, a discussion that arises relates to the possibility of generalizing our smells to other front-end frameworks. Since these frameworks also rely on components for structuring and organizing Web UIs, code smells such as Large Component, Duplicated Component, Uncontrolled Components, Large Files, Too Many Props, Low Cohesion, Direct DOM Manipulation, and Prop Drilling can be easily generalized to these frameworks. It is also possible to find bad practices that force the update of components or even reload the entire page in applications based on other frameworks. For example, we can use `forceUpdate()` and `location.reload()` methods when working with Vue.

Vue recommends using templates to build HTML in most cases (instead of JSX, as in React). Despite that, Vue also provides the `render` method, which is a closer alternative to templates and also allows the usage of JSX. Therefore, we can generalize the JSX Outside the Render Method smell to Vue-based systems. Finally, the Inheritance instead of Composition smell also applies to Vue, since Vue provides extension and mixins mechanisms to reuse features.

Therefore, this preliminary analysis reveals that it possible to generalize most of our detected smells to other frameworks, particularly Vue. However, we acknowledge that a further analysis should be conducted in this direction, possibly also considering other frameworks, such as Angular and Svelte.

10 Threats to Validity

In this section we discuss threats to the validity of our results [27]. Since we started with a list of smells from grey literature documents, our results and observations may include questionable smells, threatening validity. However, to reinforce the validity of our findings, we only selected articles that attend to at least one of the "authority of the producer criteria", proposed by Garousi et al. [15]. Moreover, we only considered smells cited by more than one document. In a second step, all smells were validated with professional React developers. Finally, we conducted a field study that showed the collected smells are prevalent in open-source systems.

Another threat relates to the developers' interviews. Some developers may be concerned on stating that a bad practice was common in their company. To minimize this threat, we allowed the participants to comment on any smells they observed while working with React, regardless of place and time.

The values chosen as thresholds can also threaten the validity of this study. To minimize this thread, we decided to be conservative in our thresholds selection policy and used the 90th percentile value of each metric. Moreover, we also achieved promising results using these same thresholds when we asked an experienced developer to evaluate the smells detected by ReactSniffer in a commercial project. Specifically, the average relevance score was 3.16 (in a scale from 1 to 5).

A final threat relates to decisions that may affect our experimental results. As usual in empirical software engineering studies, our dataset might not represent the whole population of React-based projects. For example, we selected only 10 GitHub projects (plus one closed project, which we used to provide a first validation of our results with a professional software developer). Therefore, future studies might also include more closed projects. Moreover, as a complementary selection criteria, future studies might also consider the proportion of React-based code in a repository. This extra criteria might contribute to select projects that heavily depend on React to build their front-end components.

11 Related Work

There is a large body of papers, chapters, and books on code smells. The principal one is a chapter by Fowler and Beck [4], which proposes 22 code smells for object-oriented design and associates each one with a possible fixing refactoring. More recently, code smells have also been studied for web technologies. However, to the best of our knowledge, no research has focused on code smells associated with the adoption of JavaScript front-end frameworks.

In the following subsections, we discuss studies on code smells in Web technologies and detection strategies and tools.

11.1 Code smells in Web Technologies

These works study bad practices in HTML, CSS, JavaScript, and MVC frameworks. For example, Nederlof *et al.* [7] investigated deviations from best practices in performance, accessibility, and correct structuring of HTML documents. As cited in our interviews, the process of transforming dynamic components into HTML can impact these topics. For example, dynamic components might not be interpreted reliably by a wide variety of user agents, including assistive technologies, therefore impacting accessibility. Mazinanian *et al.* [8] propose an automated approach to remove duplication in CSS code. Their approach detects three types of CSS declaration duplication and recommends refactorings to eliminate each one. The authors show that duplication in CSS is widely common. Since the main way to style React applications is by writing CSS styling separate from JSX files, their work also applies to React applications. However, none of theses works focused on smells specific to the usage of JavaScript front-end frameworks. In our RQ1, we filled this gap by answering what are the most common code smells in React.

Fard and Mesbah [6] propose a set of 13 JavaScript code smells and a code smell detection technique, called JSNOSE. They investigate the occurrence of these smells in 11 Web applications and show that Lazy Object, Long Method/Function, Closure Smells, Coupling between JavaScript, HTML, CSS, and Excessive Global Variables are the most prevalent smells. Interestingly, they include HTML in JavaScript and JavaScript in HTML as code smells, which is exactly one of the key characteristics of modern JavaScript front-end frameworks. For example, React allows writing HTML in JavaScript, and Vue allows writing JavaScript in HTML [12].

Aniche *et al.* [10] present a catalog of six smells for Web-based MVC applications. To define the catalog, the authors rely on interviews with Spring developers (Spring is a popular Java-based MVC framework). They show that the proposed smells are more subjected to changes and defects and that developers indeed perceive them as relevant problems. The authors claim the proposed smells can be generalized to other frameworks, although they are more related to object-oriented design than to SPA-related technologies. For example, Brain Repository, Laborious Repository Method,

[12]However, this is not a surprise, considering that web development technologies have drastically changed since when the article was published (2013).

and Fat Repository smells refer to persistence classes. The other smells are Promiscuous Controller, Brain Controller, and Meddling Service, which refer to the back-end layer of Web-based systems.

As can be found in Fard and Mesbah [6] (but for code smells in JavaScript) and Aniche et al. [10] (but for code smells in MVC architectures), in our RQ2, we show which REACT smells are most common in real-world projects. Finally, Aniche et al. [10] also investigate when these smells are introduced and how long it takes to refactor/remove them from a sample of Java-based systems. Similarly, R3 shows how often the identified REACT smells are removed.

11.2 Detection Strategies and Tools

An essential point when detecting code smells regards the definition of the thresholds for the selected metrics. Some authors propose thresholds based on their experience only. For example, in the seventies McCabe *et al.* [28] proposed the value ten as a threshold for the Cyclomatic Complexity metric based on his past experience. Other approaches, like ours, use real-world software systems to derive metric thresholds[29, 30, 31]. For example, Alves *et al.* [21] determine thresholds empirically from measurement data of a benchmark of software systems and derive the threshold values by choosing the 70%, 80%, and 90% percentiles. Aniche *et al.* [10] use the third quartile (3Q) and the interquartile range (3Q - 1Q) to define the thresholds. and approaches that derive relative thresholds from real systems based on a statistical analysis [32, 33]. Oliveira *et al.* [32, 33] propose an empirical method for extracting relative thresholds from real systems based on a statistical analysis.

As described in Section 5, we used as a threshold the 90th percentile of each metric distribution.

There are also approaches that rely on evolution history information to detect the smells, such as HIST (Palomba *et al.* [22]) and DECOR (Moha *et al.* [34]), and approaches that use Deep Learning (Fakhoury *et al.* [35]; Liu *et al.* [36]).

Existing tools to detect smells in Web languages and frameworks generally use static analysis or a blend of static and dynamic analysis. For example, JSNOSE [6] applies static and dynamic analysis for detecting code smells in JavaScript code. CILLA [37] is a tool that also relies on dynamic analysis for detecting unused CSS code. Another related tool is WEBSCENT [38], which identifies six types of Embedded Code Smells in Dynamic Web Applications.

12 Conclusion

Code smells were first proposed for mainstream object-oriented languages. For example, the examples presented in Fowler and Beck's original catalog of smells are implemented in Java [4]. However, software engineering and related technologies have evolved considerably in the last decades. Particularly, web-based systems evolved to include full and non-trivial applications running in the browsers, which are implemented using frameworks such as Facebook's REACT. Therefore, as the key implication of our study we showed that REACT-based applications include new and specific smells that are not covered in general-purpose catalogs. We claim this finding can help front-end

developers—who represent 27% of the developers according to recent surveys—to better maintain and improve the quality of their code.

React is a very popular JavaScript front-end framework. It is now used to implement complex and Reactive Web interfaces, which can reach thousands of lines of code. Therefore, it is important to assure the maintainability of React-based applications. In this paper, by using a grey literature review and by interviewing professional React developers, we derived a list of 12 code smells for React-based apps. We provided examples for each smell; implemented a tool to detect them; and used this tool in a sample of ten GitHub projects, when we were able to detect 2,565 smells instances, covering nine out of 12 smell types proposed in the paper. As future work, we plan to consider new frameworks, such as Vue.js, Angular, and Svelte. This is particularly important to provide a general catalog of smells for front-development and therefore to avoid the explosion of smells for a wide range of frameworks. We also plan to extend our prototype tool to handle all identified smells.

References

[1] M. Mikowski and J. Powell, *Single page web applications: JavaScript end-to-end*. Manning Publications Co., 2013.

[2] F. Ferreira, H. Borges, and M. T. Valente, "On the (un-)adoption of JavaScript front-end frameworks," *Software: Practice and Experience*, vol. 52, pp. 947–966, 2022.

[3] M. Bajammal, D. Mazinanian, and A. Mesbah, "Generating reusable web components from mockups," in *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 601–611.

[4] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[5] A. Hora, "Googling for software development: What developers search for and what they find," in *18th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2021, pp. 1–12.

[6] A. M. Fard and A. Mesbah, "Jsnose: Detecting JavaScript code smells," in *13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2013, pp. 116–125.

[7] A. Nederlof, A. Mesbah, and A. van Deursen, "Software engineering for the web: the state of the practice," in *36th International Conference on Software Engineering (ICSE)*, 2014, pp. 4–13.

[8] D. Mazinanian, N. Tsantalis, and A. Mesbah, "Discovering refactoring opportunities in cascading style sheets," in *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 496–506.

[9] L. Vegi and M. T. Valente, "Code smells in Elixir: Early results from a grey literature review," in *30th International Conference on Program Comprehension (ICPC)*, 2022, pp. 1–5.

[10] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen, "Code smells for model-view-controller architectures," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2121–2157, 2018.

[11] V. Garousi, M. Felderer, and M. V. Mäntylä, "The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature," in *20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2016, pp. 1–6.

[12] T. Barik, B. Johnson, and E. Murphy-Hill, "I heart hacker news: expanding qualitative research findings by analyzing social news websites," in *10th Joint Meeting on Foundations of Software Engineering (FSE)*, 2015, pp. 882–885.

[13] F. Kamei, I. Wiese, C. Lima, I. Polato, V. Nepomuceno, W. Ferreira, M. Ribeiro, C. Pena, B. Cartaxo, G. Pinto *et al.*, "Grey literature in software engineering: A critical review," *Information and Software Technology*, vol. 1, no. 1, pp. 1–26, 2021.

[14] H. Zhang, X. Zhou, X. Huang, H. Huang, and M. A. Babar, "An evidence-based inquiry into the use of grey literature in software engineering," in *42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1422–1434.

[15] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, no. 1, pp. 101–121, 2019.

[16] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2011, pp. 275–284.

[17] D. S. Cruzes and T. Dybå, "Research synthesis in software engineering: A tertiary study," *Information and Software Technology*, vol. 53, no. 5, pp. 440–455, 2011.

[18] J. Ousterhout, *A Philosophy of Software Design*. Yaknyam Press, 2018.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[20] R. C. Martin, J. Grenning, and S. Brown, *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall, 2018.

[21] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *26th IEEE International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.

[22] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 268–278.

[23] J. Pantiuchina, M. Lanza, and G. Bavota, "Improving code: The (mis) perception of quality metrics," in *34th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 80–91.

[24] M. Ó. Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam, "Experimental assessment of software metrics using automated refactoring," in *6th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2012, pp. 49–58.

[25] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 334–344.

[26] H. Silva and M. T. Valente, "What's in a GitHub star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, no. 1, pp. 112–129, 2018.

[27] C. Wohlin, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[28] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, vol. 1, no. 4, pp. 308–320, 1976.

[29] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[30] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *6th International Workshop on Emerging Trends in Software Metrics (WETSoM*, 2015, pp. 44–53.

[31] P. Oliveira, M. T. Valente, and F. Lima, "Extracting relative thresholds for source code metrics," in *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 254–263.

[32] ——, "Extracting relative thresholds for source code metrics," in *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 254–263.

[33] P. Oliveira, F. Lima, M. T. Valente, and A. Serebrenik, "RTTool: A tool for extracting relative thresholds for source code metrics," in *30th International Conference on Software Maintenance and Evolution (ICSME), Tool Demo Track*, 2014, pp. 629–632.

[34] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[35] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?" in *SANER*, 2018, pp. 602–611.

[36] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, "Deep learning based code smell detection," *IEEE Transactions on software Engineering*, pp. 1–28, 2019.

[37] A. Mesbah and S. Mirshokraie, "Automated analysis of css rules to support style maintenance," in *34th International Conference on Software Engineering (ICSE)*, 2012, pp. 408–418.

[38] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Detection of embedded code smells in dynamic web applications," in *27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2012, pp. 282–285.