UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Fábio da Silva Ferreira

Assisting JavaScript Front-End Developers in Maintaining and Evolving React-Based Applications: Code Smells and Refactoring Operations

Belo Horizonte
2023

Fábio da Silva Ferreira

Assisting JavaScript Front-End Developers in Maintaining and Evolving React-Based Applications: Code Smells and Refactoring Operations

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Marco Túlio de Oliveira Valente

Belo Horizonte
2023

2023, Fábio da Silva Ferreira.
Todos os direitos reservados

Ferreira, Fábio da Silva.

F383a Assisting JavaScript front-end developers in maintaining and evolving react-based applications: code smells and refactoring operations [recurso eletrônico] : / Fábio da Silva Ferreira – 2023.
1 recurso online (113 f. il., color.) : pdf.

Orientador: Marco Túlio de Oliveira Valente.

Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciências da Computação.
Referências: f.106-113

1. Computação – Teses. 2. Engenharia de Software – Teses. JavaScript (Linguagem de programação de computador) – Teses. 4. Mineração de repositórios de software – Teses. I. Valente, Marco Túlio de Oliveira. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. III.Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

ASSISTING JAVASCRIPT FRONT-END DEVELOPERS IN MAINTAINING AND EVOLVING REACT-BASED
APPLICATIONS: CODE SMELLS AND REFACTORING OPERATIONS

FÁBIO DA SILVA FERREIRA

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Marco Túlio de Oliveira Valente - Orientador
Departamento de Ciência da Computação - UFMG

Prof. Marcelo de Almeida Maia
Faculdade de Computação - UFU

Prof. Breno Bernard Nicolau de França
Instituto de Computação - UNICAMP

Prof. André Cavalcante Hora
Departamento de Ciência da Computação - UFMG

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 11 de agosto de 2023.

Documento assinado eletronicamente por **Marco Tulio de Oliveira Valente**, **Professor do Magistério Superior**, em 11/08/2023, às 16:07, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Andre Cavalcante Hora**, **Professor do Magistério Superior**, em 14/08/2023, às 11:46, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Eduardo Magno Lages Figueiredo**, **Professor do Magistério Superior**, em 14/08/2023, às 14:01, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Marcelo de Almeida Maia**, **Usuário Externo**, em 04/09/2023, às 14:52, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Breno Bernard Nicolau de França**, **Usuário Externo**, em 04/09/2023, às 15:39, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **2534826** e o código CRC **869D5C39**.

To Maria Marcília and Antônio, my dearly loved parents.

Acknowledgments

Este trabalho representa o resultado de uma intensa jornada de aprendizado. Por isso, agradeço a cada um dos familiares, amigos e colegas que tanto me ensinaram e incentivaram ao longo desse caminho. Agradeço, em especial:

À Deus por me amparar nos momentos difíceis, dando-me forças para superar as dificuldades e guiando-me nos momentos de incerteza.

Aos meus familiares, pelo incentivo constante ao longo de todo o curso. Em especial, gostaria de expressar minha profunda gratidão aos meus pais, Maria Marcília e Antônio Ferreira, pelo apoio incondicional e incentivo recebidos. Aos meus irmãos, Razzo e Aline, por estarem ao meu lado durante os momentos mais difíceis e por compartilharmos o cuidado e apoio necessário durante os tratamentos do nosso pai.

Ao meu orientador, professor Marco Túlio Valente, pela oportunidade oferecida, pela confiança e pelos desafios impostos durante todos esses anos. Pelas revisões, sugestões e suporte essenciais para que este trabalho fosse desenvolvido. Por me transmitir tantos ensinamentos que levarei comigo em minha vida e carreira.

Aos colegas do ASERG, pelo aprendizado constante e pelos momentos de crescimento pessoal e profissional. Especialmente à Hudson Borges por tanto contribuir com este trabalho. Gostaria de agradecer também à Luciana Silva, Laerte Xavier, Rodrigo Brito e João Eduardo Montandon pelas parcerias de trabalho.

Aos professores membros da banca, Marcelo Maia, Breno de França, André Hora e Eduardo Figueiredo, pela disponibilidade em contribuir com este trabalho.

Aos colegas do LABSOFT, João Paulo, Daniel Cruz, Johnatan Oliveira e Cleiton Tavares, pelas parcerias de trabalho e estudos.

Meu muito obrigado ao Instituto Federal do Sudeste de Minas Gerais - Campus Barbacena e aos colegas de trabalho que estiveram ao meu lado desde o início desta caminhada. Em especial, agradeço ao Núcleo de Informática pelo sacrifício realizado durante minha ausência.

Por fim, à Priscila Sad de Sousa por me apoiar de todas as formas possíveis do início ao fim deste curso. Agradeço principalmente pelo seu amor e carinho que inspira confiança e incentivo me ajudando a terminar este trabalho com êxito!

“*Education is an ornament in prosperity, a refuge in adversity, and the best provision for old age.*”
(Aristotle)

Resumo

Frameworks front-end baseados em JavaScript, como React e Vue, são ferramentas-chave para implementar aplicações Web modernas. No entanto, devido à complexidade das interfaces Web, essas aplicações podem atingir centenas de componentes e arquivos de código fonte. Consequentemente, os desenvolvedores front-end estão enfrentando desafios crescentes ao projetar e modularizar essas aplicações. No entanto, pouca pesquisa tem se aprofundado nos desafios de manutenção associados à adoção de tais frameworks. Especificamente, (1) falta-nos uma compreensão dos fatores que motivam a adoção desses frameworks, (2) falta-nos um catálogo de "code smells" comuns que podem surgir nos clientes desses frameworks, e, por fim, (3) faltam-nos informações sobre as operações de refatoração que os desenvolvedores realizam ao implementar sistemas Web usando esses frameworks. Nesse contexto, o principal objetivo desta tese de doutorado é auxiliar os desenvolvedores front-end na manutenção e evolução de seus sistemas. Para alcançar esse objetivo, apresentamos um catálogo de "code smells" comuns a esses sistemas, bem como um catálogo de refatorações que podem ser aplicadas para eliminar esses "smells" e, consequentemente, melhorar a qualidade do código front-end. Organizamos esta pesquisa em três unidades de trabalho principais. Começamos investigando os fatores que motivam a adoção de frameworks front-end em JavaScript. Particularmente, realizamos uma pesquisa com desenvolvedores de projetos JavaScript sobre sua motivação para adotar tais frameworks. No segundo estudo, propomos um catálogo de 12 "code smells" associados a problemas de design enfrentados pelos desenvolvedores ao implementar aplicações React. Além disso, para verificar se os "smells" identificados são prevalentes em projetos de software reais, também implementamos uma ferramenta chamada ReactSniffer para detectar os "smells" propostos. Como resultado, detectamos 2.565 ocorrências dos "code smells" propostos nos 10 projetos mais populares do GitHub que utilizam React. Finalmente, em nosso último e terceiro estudo, investigamos as operações de refatoração realizadas com frequência por desenvolvedores ao manter e evoluir aplicações React. Após inspecionar manualmente 320 commits de refatoração realizados em projetos de código aberto, catalogamos 69 operações de refatoração distintas, das quais 25 são específicas para código React, 17 são adaptações de refatorações tradicionais para o contexto React, 22 são refatorações tradicionais e seis são específicas para código JavaScript e CSS.

Palavras-chave: javaScript, front-end frameworks, refatoração, code smells, mineração de repositórios de software, manutenção de software

Abstract

JavaScript-based front-end frameworks, such as React and Vue, are key tools for implementing modern Web applications. However, due to the complexity of Web UIs, these applications can reach hundreds of components and source code files. Consequently, front-end developers are facing increasing challenges in designing and modularizing these applications. Surprisingly, only limited research has delved into the maintenance challenges associated with adopting such frameworks. Specifically, (1) we lack a deep understanding of the factors driving their adoption, (2) we lack a comprehensive catalog of common code smells that may arise in clients of these frameworks, and finally, (3) we lack information and documentation on the refactoring operations that developers perform when implementing Web-based systems using these frameworks. In this context, the main goal of this Ph.D. thesis is to assist JavaScript front-end developers in maintaining and evolving their systems. To achieve this goal, we introduce a catalog of code smells that are common in such systems, as well as a catalog of refactorings that can be applied to eliminate these smells and consequently improve the source code quality of front-end components. We organize the research into three major working units. We start by investigating the factors that motivate the adoption of JavaScript front-end frameworks. Notably, we survey developers of JavaScript projects about their motivation to adopt such frameworks. In the second study, we propose a catalog of 12 code smells associated with design problems developers face when using JavaScript front-end frameworks. We focus on React applications because it is currently the most popular JavaScript front-end framework. Moreover, to check whether the identified smells are prevalent in real software projects, we also implement a tool, called ReactSniffer, to detect the proposed smells. As a result, we detected 2,565 instances of the proposed code smells in the top-10 most popular GitHub projects that use React. Finally, in our last and third study, we investigate frequent refactoring operations developers perform when maintaining and evolving React applications. By manually inspecting 320 refactoring commits performed in open source projects, we catalog 69 distinct refactoring operations of which 25 are specific to React code, 17 are adaptations of traditional refactorings for the React context, 22 are traditional refactorings, and six are specific to JavaScript and CSS code.

Keywords: javaScript, front-end frameworks, refactoring, code smells, mining software repositories, maintainability

List of Figures

1.1 Multi-Page Application vs Single-Page Application 16

2.1 Gallery of images application 26
2.2 Structure of components to display comments 29

4.1 Overview of the grey literature methodology 52
4.2 Example of Prop Drilling (code smell). 58
4.3 ReactSniffer architecture 65
4.4 Time frames used in the analysis 73
4.5 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) 74
4.6 Issue questioning the use of Force Update 74
4.7 Direct DOM Manipulation Refactoring 75
4.8 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) 76

5.1 Example of an unclear commit. 82
5.2 Example of a discarded commit because it is revoking a previous refactoring operation. 82
5.3 Example of a false positive commit that does not preserve behavior. 84
5.4 Number of refactoring operations by category 85
5.5 Refactoring that extracts a stateful logic to a custom hook 87
5.6 Refactoring `CreateUserDialog` class component to function component . . . 88
5.7 Commit message indicanting an Extract HOC refactoring 90

List of Tables

3.1 Package names used by the studied frameworks 39
3.2 Number of clients by framework (Initial Selection) 40
3.3 Dataset to study framework's adoption 40
3.4 Front-end files extensions 41
3.5 Factors motivating the adoption of front-end frameworks 42
3.6 Factors that motivate the adoption of JavaScript front-end frameworks (answers per individual frameworks). 43
3.7 Weakness factors of JavaScript front-end frameworks (answers per individual frameworks). 44

4.1 Code smells identified in the grey literature 53
4.2 Participants experience 54
4.3 Code smells validated in the interviews, with examples of participants' comments 55
4.4 New code smells identified in the interviews 55
4.5 Front-end files extensions 66
4.6 Dataset (FF: number of front-end files; Comp: number of components) 70
4.7 Thresholds selection 70
4.8 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) 71
4.9 Dataset (FF: number of front-end files; Comp: number of components) 75
4.10 Relevance scores of ReactSniffer Evaluation 77

5.1 Dataset of React clients (FF: number of front-end files; Comp: number of components; Ref: Number of refactoring commits) 81
5.2 Example of refactoring operations labeled with different names (the final selected version is underlined). 83
5.3 React-specific refactorings 86
5.4 React-adapted refactorings 92
5.5 JavaScript and CSS refactorings 95
5.6 Traditional refactorings 96
5.7 React smells and the refactorings that eliminate them 97

Contents

1 Introduction **14**

1.1 Problem and Motivation 14
1.2 Objectives and Contributions 18
1.3 Publications 21
1.4 Thesis Outline 22

2 Background and Related Work **23**

2.1 JavaScript Front-end Frameworks 23
2.2 Related Work 31
2.3 Final Remarks 37

3 Adoption of JavaScript Front-end Frameworks **38**

3.1 Dataset 39
3.2 Survey Design 40
3.3 Survey Results 42
3.4 Implications 47
3.5 Threats to Validity 47
3.6 Final Remarks 48

4 Detecting Code Smells in React-based Web Apps **50**

4.1 Methodology 51
4.2 React Code Smells 57
4.3 ReactSniffer: Code Smell Detection Tool 65
4.4 Field Study 69
4.5 Historical Analysis 73
4.6 Validation with Developers 75
4.7 Discussion 77
4.8 Threats to Validity 78
4.9 Final Remarks 79

5 Refactoring React-based Web Apps **80**

5.1 Study Design 80
5.2 A Catalog of Refactorings for React-Based Web Apps 85
5.3 Discussion 95

5.4 Threats to Validity . . . 98
5.5 Final Remarks . . . 100

6 Conclusion **101**
6.1 Thesis Recapitulation . . . 101
6.2 Contributions . . . 102
6.3 Future Work . . . 103

Bibliography **106**

Chapter 1

Introduction

This chapter introduces this thesis. We start by stating our problem and motivation in Section 1.1. Section 1.2 details our objectives, goals, and intended contributions, while Section 1.3 present the list of publications resulted from this thesis. Finally, we present the structure of this thesis in Section 1.4.

1.1 Problem and Motivation

The World Wide Web (WWW) emerged in the 1990s with the expansion of the Internet around the world. Initially, the Web consisted of static documents identified by Uniform Resource Locators (URLs) and accessed using the Hypertext Transfer Protocol (HTTP). However, the success of the WWW increased the complexity of its content which evolved from static pages to Web systems that behave similarly to desktop applications [Araújo and Filho, 2020]. This evolution was possible mainly due to the emergence of JavaScript, a programming language introduced by Netscape in the mid-1990s that allows developers to enhance user interface by creating dynamic and responsive elements [Silva et al., 2017].

JavaScript has grown over the years and has become the de-facto programming language for the Web. For example, according to the most recent StackOverflow Survey,[1] JavaScript is the world's most popular programming language for the eleventh year in a row. Moreover, HTML and CSS emerged in the second place in the last six years.[2] Essentially, this popularity reflects the importance of modern Web-based systems.

JavaScript is characterized by a large, rich, and dynamic ecosystem of frameworks and libraries [Wittern et al., 2016]. For example, NPM—the largest package repository for the language—hosts more than 3.2 million projects.[3] Over time, several JavaScript

[1] https://survey.stackoverflow.co/2022/#most-popular-technologies-language

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

[3] https://www.npmjs.com/

frameworks and libraries also emerged to address problems that appear when engineering complex user interfaces [Araújo and Filho, 2020; Ramos et al., 2018]. As a distinguished example, we have front-end frameworks—such as ANGULAR,[4] REACT,[5] and VUE[6]— that are relevant tools for building Single-Page Applications (SPAs).

SPA is an application that runs as a single Web page and that does not reload during its use [Mikowski and Powell, 2013]. Instead, the browser loads the entire application in a single HTML page along with JavaScript and CSS resources, which can update parts of the UI without reloading the entire page. The ultimate goal is to provide a user experience similar to that of a desktop application. Figure 1.1 illustrates the mechanics behind Multi-Page and Single-Page Applications lifecycles. In Multi-Page Applications (Figure 1.2(a)), user actions are embodied in HTTP requests (e.g., POST and GET) and the server responds with a fresh HTML page. On the other hand, Single-Page Applications (Figure 1.2(b)) handle user actions through Asynchronous JavaScript and XML (AJAX) calls. Therefore, instead of requesting a new page from the server, SPAs only request the necessary data to update parts of the UI. As a positive outcome, there is no need to reload the entire page.

The evolution of Web applications is so significant that developers roles focused on front-end concerns emerged recently as a new career [Montandon et al., 2020]. According to Stack Overflow Survey,[7] 25.9% percent of the surveyed programmers worldwide are front-end developers. Comparing Web frameworks and technologies, the same survey shows that both VUE and REACT are more popular than traditional MVC-based Web frameworks, such as Spring, Django, and Ruby on Rails.[8] Furthermore, the former are usually developed and maintained by major Internet companies, as is the case of REACT (Facebook) and ANGULAR (Google).

Essentially, the afore-mentioned 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 [Bajammal et al., 2018]. Moreover, 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 easily reach hundreds of components and source code files. As a result, **it is natural to expect that suboptimal design decisions will eventually lead to SPAs that are hard to maintain, understand, modify, and test**.

[4] https://angular.io/

[5] https://reactjs.org/

[6] https://vuejs.org/

[7] https://survey.stackoverflow.co/2022/#developer-profile-developer-roles

[8] https://survey.stackoverflow.co/2022/#section-most-popular-technologies-web-frameworks-and-technologies

Figure 1.1: Multi-Page Application vs Single-Page Application

(a) Multi-Page Application

(b) Single-Page Application

As previous research in the area has shown, a important indicator of maintenance problems is code smells [Fowler and Beck, 1999, 2018; Lacerda et al., 2020; Sobrinho et al., 2018; Sharma and Spinellis, 2018]. Such structures indicate problems like low cohesion, high coupling, and encapsulation-related problems that influence design decisions and maintenance [Sobrinho et al., 2018]. In this context, refactoring is a well-known technique to improve software design and an indispensable practice in modern software development [Fowler and Beck, 1999, 2018]. Refactoring and code smells are linked because refactoring is the principal strategy to remove/mitigate code smells, and thus improving software quality.

However, although traditional code smells and refactorings describe general problems in object-oriented design **we still lack studies investigating the key maintenance problems that occur when implementing Web-based systems using JavaScript front-end frameworks**. Particularly, we identify the following specific problems in this context:

1. A lack of understanding on the factors that motivate the adoption of modern JavaScript front-end frameworks. Since JavaScript is characterized by a large and dynamic ecosystem of frameworks and libraries, this information has practical value for the software's evolution and practitioners.

2. The lack of a code smells catalog that can be shared among practitioners for documenting and discussing specific design problems that occur when using these frameworks, as well as empirical studies on the prevalence and relevance of these smells. Furthermore, we also miss detection tools that can warn developers about these front-end specific code smells;

3. A lack of information and documentation on the refactoring operations that developers perform when maintaining and evolving Web systems using these frameworks.

Regarding the lack of studies on the factors that motivate the adoption of JavaScript front-end frameworks, an important exception is a study by Pano et al. [2018], where the authors interviewed 18 developers regarding their criteria for selecting JavaScript frameworks and, as a result, they propose a model of framework adoption factors. However, the study was conducted in July 2014, when front-end frameworks were still in their early adoption phases. For example, the second most popular framework nowadays (Vue) was not mentioned by any participant, and a single participant mentioned the most popular framework nowadays (Facebook's React). By contrast, jQuery—which is not widely used anymore—was cited by half of the interviewed developers.

Concerning code smells associated to design problems in Web-based systems, previous studies identified such structures in pure JavaScript [Fard and Mesbah, 2013; Saboury et al., 2017; Johannes et al., 2019], HTML [Harold, 2012; Nederlof et al., 2014], Cascading Style Sheets (CSS) [Mazinanian et al., 2014], and in MVC frameworks [Aniche et al., 2018], but did not focus on smells specific to the usage of JavaScript front-end frameworks, such as React.

Finally, although refactoring is a well-known technique to improve software design and an indispensable practice in modern software development, most studies on refactoring focus on mainstream programming languages, such as Java [Fowler and Beck, 1999] and JavaScript [Fowler and Beck, 2018]. There are also few studies targeting refactorings performed in particular domains, such as CSS [Mazinanian et al., 2014], Android [Peruma et al., 2020], Docker projects [Ksontini et al., 2021], and machine learning systems [Tang et al., 2021]. However, refactoring has not been studied in the relevant domain of JavaScript front-end frameworks.

In summary, to the best of our knowledge, **we are the first to propose a holistic investigation on code smells and refactorings in the particular but relevant context of Web-based systems implemented using JavaScript front-end frameworks**.

1.2 Objectives and Contributions

As previously mentioned, we still lack studies investigating design and maintenance problems specific to the context of Web-based systems implemented using JavaScript front-end frameworks. Therefore, the general objective of this thesis is described as follows:

> We aim to assist JavaScript front-end developers in the task of maintaining and evolving their systems. This assistance will be materialized by providing a catalog of code smells that are common in such systems, as well as a catalog of refactorings that can be applied to eliminate them and thus improve the source code quality of front-end components.

To make this research possible, we divided the work into three major working units:

1. First, we study the factors that motivate the adoption of JavaScript front-end frameworks. Notably, we survey developers of JavaScript projects about their motivation to adopt such frameworks. The primary objective was to gain an initial understanding of this area, which is new to our research group but also in the global landscape of software engineering research. Additionally, given JavaScript's expansive and dynamic ecosystem of frameworks and libraries, comprehending this dynamism becomes paramount in assisting developers to select frameworks that align with their requirements and avoid potential maintenance issues tied to these frameworks.

2. In the second study, we propose a catalog of code smells associated to design problems faced by developers when using JavaScript front-end frameworks. Based on the results of the first study, we focus on React applications because it is currently the most popular JavaScript front-end framework. Moreover, to check whether the identified smells are prevalent in open-source systems, we also implement a tool to detect the proposed smells.

3. Finally, in the third study, we investigate frequent refactoring operations that developers perform when maintaining and evolving React applications. We categorize these operations into Traditional, React-adapted, React-specific, and JavaScript and CSS related refactorings. We also identify the refactoring operations that eliminate the design problems found in the second study.

These objectives are covered in three studies detailed in next chapters. Particularly, in **Chapter 3**, we investigate the factors that motivate the adoption of JavaScript front-end frameworks. In **Chapter 4**, we propose a catalog of code smells documenting design

problems in JavaScript React-based applications. To build this catalog, we conduct a grey literature review and interview professional React developers. We focus on React because it is currently the most popular JavaScript front-end framework [Hora, 2021].[9] For example, according to BuiltWith,[10] more than 18 million websites are powered by React. Of the top 10K sites by traffic, 40.9% are built with React. Finally, in **Chapter 5**, we investigate the most important refactoring operations that developers perform when maintaining and evolving React-based applications.

We summarize each study and highlight their contributions in the remainder of this section.

1.2.1 Adoption of JavaScript Front-end Frameworks

Due to the increasing complexity of Web applications and the number of JavaScript front-end frameworks available, developers face difficulties defining the most suitable characteristics of a framework to use in their projects. A wrong choice can lead to maintenance problems over the application's lifetime. Despite that, we still lack empirical results on the factors that motivate the adoption of these frameworks. Therefore, in Chapter 3, we report the results of a study on the factors that drive the adoption of JavaScript frameworks. Particularly, **we reveal a list of nine key factors developers consider when selecting contemporary JavaScript front-end frameworks**. This list can help JavaScript developers that plan to adopt a front-end framework in their projects. It can also be used by framework developers, helping them to better position their projects in a very competitive software market.

1.2.2 Catalog of Code Smells

Modern Web applications can reach hundreds of components, lines of code, and files. As a result, it is natural to expect that suboptimal design decisions will eventually lead to code that is hard to maintain. In this context, identifying design problems when using these frameworks has a key importance. For example, front-end developers can use this information for refactoring and better understanding and evolving their systems.

[9] https://survey.stackoverflow.co/2023/#most-popular-technologies-webframe

[10] `https://trends.builtwith.com/javascript/React`

Therefore, in Chapter 4, **we propose a set of 12 code smells common in React applications**. These smells were identified by conducting a grey literature review and by interviewing six professional software developers. **We also implement a tool, called ReactSniffer, to detect the proposed code smells.** We use this tool to unveil the most common code smells in REACT-based Web systems. For example, in the top-10 most popular GitHub projects that use React, we detect 2,565 instances of the proposed code smells.

In summary, in this working unit we present the following contributions:

- A catalog of code smells for JavaScript REACT-based applications.
- The method employed for constructing our catalog of code smells through a grey literature review followed by validation with developers represents a valuable contribution that can be used or adapted to other technologies.
- A tool, called REACTSNIFFER, to detect the proposed code smells. It is also publicly available as a NPM package.[11]
- We used ReactSniffer to unveil the most common code smells in REACT-based systems.
- A dataset of code smells for REACT applications.

1.2.3 Refactoring React-based Web Apps

Refactoring is a well-known technique to improve software design and an indispensable practice in modern software development. Despite that, there is a relevant domain where refactoring has not been studied in depth before. It includes the front-end components that are part of modern Web UIs. Particularly the components implemented using JavaScript-based frameworks, such as REACT and VUE.

Therefore, in Chapter 5, we report the results of a large empirical study on refactoring operations that developers perform when maintaining and evolving REACT-based Web applications. By manually inspecting 320 refactoring commits performed in open source projects, **we catalog 69 distinct refactoring operations of which 22 refactorings are traditional transformations (*i.e.,* described in Fowler's book), six are specific to JavaScript and CSS code, 25 are specific to React code, and 17 are**

[11] `https://www.npmjs.com/package/reactsniffer`

React-adapted refactorings (*i.e.,* although related to the React context, they are adaptations of traditional refactorings). The catalog of refactorings proposed as a result of this work might support practitioners when improving the maintainability of React applications. The main contributions of this final working unit encompass the following:

- A catalog of refactorings that developers employ when maintaining and evolving React-based applications.
- The approach utilized to identify refactoring operations by analyzing commits in front-end files, filtered through keywords in their log messages, is applicable and adaptable to other technologies and objectives.
- A dataset of refactorings performed in open-source React projects from GitHub.

1.3 Publications

The following publications are a result of this thesis:

- **SPE '22** Ferreira, F., Borges, H. S., and Valente, M. T. On the (Un-) Adoption of JavaScript Front-end frameworks . *Software: Practice and Experience*, 52(4): 947–966, 2022. (**Chapter 3**)
- **IST '23** Ferreira, F. and Valente, M. T. Detecting Code Smells in React-based Web Apps. *Information and Software Technology*, 155:1–35, 2023. (**Chapter 4**)

Furthermore, we also contributed to the following work during this Ph.D.:

- **EMSE '22** Xavier, L., Montandon, J. E., Ferreira, F., Brito, R., and Valente, M. T. On the Documentation of Self-Admitted Technical Debt in Issues . *Empirical Software Engineering*, 27(7):1–34, 2022.
- **ACM SAC '21** Ferreira, F., Silva, L. L., and Valente, M. T. Software engineering meets deep learning: a mapping study. In *36th Annual ACM Symposium on Applied Computing (SAC)*, pages 1542–1549, 2021.
- **SBES '20** Ferreira, F., Silva, L. L., and Valente, M. T. Turnover in Open-Source Projects: The Case of Core Developers. In *34th Brazilian Symposium on Software Engineering (SBES)*, pages 447–456, 2020.

- **MSR '20** Xavier, L., Ferreira, F., Brito, R., and Valente, M. T. Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems. In *17th International Conference on Mining Software Repositories (MSR)*, pages 137–146, 2020.
- **SCAM '20** Diniz, J. P., Cruz, D., Ferreira, F., Tavares, C., and Figueiredo, E. GitHub Label Embeddings. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 249–253, 2020.

1.4 Thesis Outline

We organized the remainder of this work as follows:

- Chapter 2 gives an overview of modern JavaScript front-end frameworks and presents the works that are directly related to this thesis.
- Chapter 3 presents a study aiming to understand the factors that motivate the adoption of JavaScript front-end frameworks. We reveal a list of nine key factors developers consider when selecting contemporary JavaScript front-end frameworks.
- Chapter 4 proposes a list of code smells for React-based JavaScript applications. To detect these smells, we implement a prototype tool, called ReactSniffer, and we use this tool to unveil the most common code smells in React-based Web systems.
- Chapter 5 presents the results of an empirical study on refactoring operations that developers perform when maintaining and evolving React-based Web applications. We also propose a catalog with 69 distinct refactoring operations of which 25 are specific to React code, 17 are adaptations of traditional refactorings for the React context, 22 are traditional refactorings, and six are specific to JavaScript and CSS code.
- Chapter 6 summarizes the conclusions we leveraged throughout this thesis and outlines some ideas we find interesting to investigate in the future.

Chapter 2

Background and Related Work

We start this chapter by giving an overview of modern JavaScript front-end frameworks (Section 2.1). Next, we describe the characteristics of the most popular JavaScript front-end frameworks (React) in Section 2.1.2, and comment on other frameworks in Section 2.1.3. Finally, in Section 2.2, we present other works related to this thesis.

2.1 JavaScript Front-end Frameworks

2.1.1 Overview

To facilitate reuse and shorten the development time of Web applications, programmers have created libraries and frameworks with pre-implemented JavaScript code [Pano et al., 2018]. Thus, developers can reuse the code and concentrate on the application domain. The first frameworks and libraries maintained the classic separation of responsibilities among CSS, data, structure (HTML), and dynamic interactions (JavaScript) [Araújo and Filho, 2020]. On the other hand, modern JavaScript front-end frameworks follow a component-based development paradigm. Essentially, these 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 [Bajammal et al., 2018]. Nowadays, developers have several alternatives of such frameworks, including React,[1] Vue,[2] and Angular.[3]

Particularly, the popularity of JavaScript frameworks is rising quickly mainly due to the Single-Page Application (SPA) architecture model, which allows an application

[1] `https://reactjs.org/`
[2] `https://vuejs.org/`
[3] `https://angular.io/`

to run as a single Web page that does not reload during its use. These frameworks support an architecture for creating SPAs and allow developers to create UIs with a user experience similar to a desktop application. On the one hand, frameworks for building SPAs have a lot in common. They are usually open-source, allow the creation of reusable UI components, and provide an architecture for keeping model data in sync with the view. In addition, they allow writing HTML and JavaScript code together. On the other hand, they differ in particular aspects, for example, React allows writing HTML in JavaScript, while Angular and Vue allow writing JavaScript in HTML. In the following subsections, we comment on the characteristics of these frameworks.

2.1.2 React

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, React prioritizes interoperability, *i.e,* it can be incorporated into a system without rewriting existing code, therefore allowing gradual adoption. React allows developers to focus on the view layer before introducing any other resources to their applications.

React provides a domain-specific language called JSX (which stands for JavaScript XML) for writing UI elements. Thus, rather than separating markup and logic in different files, React separates concerns under modularization units called *components*, which contain both logic and markup. By modularizing the UI into independent and reusable components, code becomes more easy to maintain.

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 supports different kinds of components. The main ones are *class* and *function* components. Though class components retain support within React, the official React documentation is recommending the use of function components in new codebases. However, it is essential to study class components since they are still largely used in React projects. The key difference between *class* and *function* is that the latter is just a JavaScript function that accepts `props` as an argument and returns a JSX code. Furthermore, there is no `render` method in functional components, and they usually do not have `state`.[4] That is, a functional component is itself a `render` method. For example, the following listings show the same component implemented using *class* (see Listing 2.1) and *function* component (see Listing 2.2). In both cases, the component accepts a single

[4]However, the introduction of hooks in React 16.8 allows adding state to function components using the `useState()` method.

object argument (`name`) via `props` and returns a React element that displays a welcome message.

Listing 2.1: Example of Class Component.

```
1  class Welcome extends React.Component {
2    render() {
3      return (
4          <h1>Hello, {this.props.name}</h1>;
5      );
6    }
7  }
```

Listing 2.2: Example of Function Component.

```
1  function Welcome(props) {
2    return (
3      <h1>Hello, {props.name}</h1>;
4    );
5  }
```

React follows functional programming principles, such as immutability, pure functions, composition, and higher-order functions. React elements are immutable. Once created, their data cannot be changed. Therefore, React assumes that every component is a pure function and must always return the same JSX given the same inputs. Moreover, components can not change existing variables while rendering. Nevertheless, React takes a step forward by introducing hooks, which allow developers to incorporate stateful behavior and manage side effects within functional components. For example, the `useState` hook facilitates the creation of stateful logic with functional components, while `useEffect` enables the management of side effects in a controlled and declarative manner. However, these changes may not happen during rendering.

Furthermore, to render a React element, we must create a root to display React components inside a browser DOM node and then pass the React element to the `root.render()` method. For example, consider the HTML page in the Listing 2.3. We can use the HTML element with id "root" (line 3) to render the React element that displays a welcome message (Listings 2.1 and 2.2).

Listing 2.3: Example of HTML page used to render a React component.

```
1  <html>
2    <body>
3      <div id="root"></div>
4    </body>
5  </html>
```

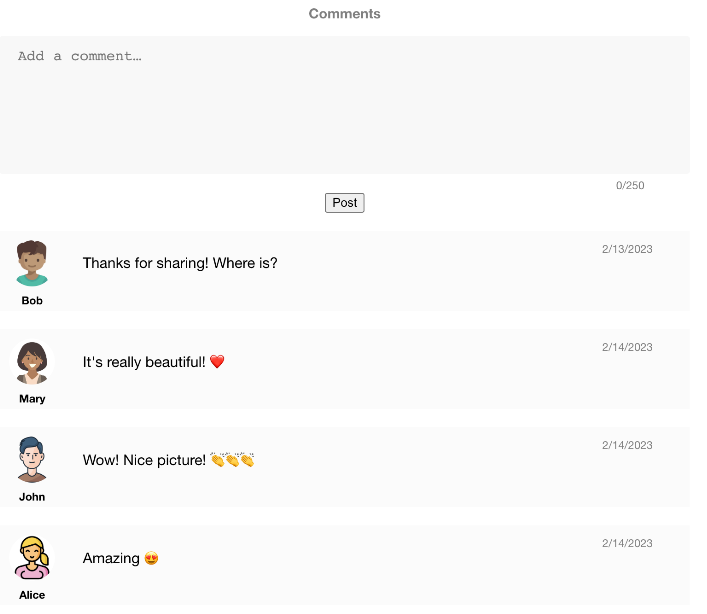
Listing 2.4 shows how to render a React element using the `Welcome` component. First, we have to pass the DOM element to `ReactDOM.createRoot()` (lines 1-3). Then, we instantiate the `Welcome` component (line 4). Finally, we pass the React element to the `root.render()` method.

Listing 2.4: Rendering a React element.

```
1  const root = ReactDOM.createRoot(
2    document.getElementById('root')
3  );
4  const reactElement = <Welcome name="Fabio">;
5  root.render(reactElement);
```

To better illustrate the React-related concepts used in this thesis, we use a small application that implements a gallery of images that allows comments by users, as illustrated in Figure 2.1.

Figure 2.1: Gallery of images application



Listing 2.5 shows the `Comment` component, which provides the code that handles each comment. This component displays the comment data via `props (lines 5-6)` and instantiates the component that displays the avatar of the user who created the comment (line 4).

Listing 2.5: Comment Component.

```
1 function Comment(props) {
2   return (
3     <div>
4       <Avatar user={props.user} />
5       <div>{props.text}</div>
6       <div>{props.date}</div>
7     </div>
8   );
9 }
```

To avoid repeating the code that displays the comments, we also create a `CommentList` component (see Listing 2.6). It receives the list of comments via `props` (line 1) and iterates in this list rendering each comment using our previous `Comment` component (line 5).

Listing 2.6: CommentList Component.

```
1 function CommentList(props) {
2   return (
3     <div>
4       { props.comments.map((comment) => (
5           <Comment date={comment.date} text={comment.text} user={comment.user} />
6       )) }
7     </div>
8   );
9 }
```

Besides *props*, a React component can also have a *state*, which differentiates stateful from stateless components. *Props* and *state* are both plain JavaScript objects. However, while both hold information that can be used in the `render` output, they have a fundamental difference: *props* are immutable and they are passed to the component (similar to function parameters), whereas the *state* is managed within the component (similar to the local variables of a function or attribute in a class). The *state* starts with a default value when the component mounts and then can change over time (mainly as a result of user events).

Therefore, a stateless component does not have a *state* and only renders what it receives via *props*. On the other hand, a stateful component in addition to taking input data (accessed via `props`) can maintain internal data (accessed via `state`). Furthermore, when a component's state changes, the component is automatically re-rendered to update the View, which is how React supports data binding. This concept refers to the association between the view with the data that populates it. There are two types of data binding: when any change in the component's state is reflected in the View, and when any change in the View is propagated to the component's state. React supports one-way

data binding since changes in the model are automatically propagated to the View, but not in the other way.

To illustrate these concepts, consider our gallery of images application. The component to create new comments deals with the user and comment data. For example, to create a new comment, the component needs to know the comment data (text and date) and the user who is creating it. The user provides the comment's data at runtime. Therefore, the component itself handles this data in its `state`. By contrast, the user data does not change. For this reason, the component gets user data through `props`.

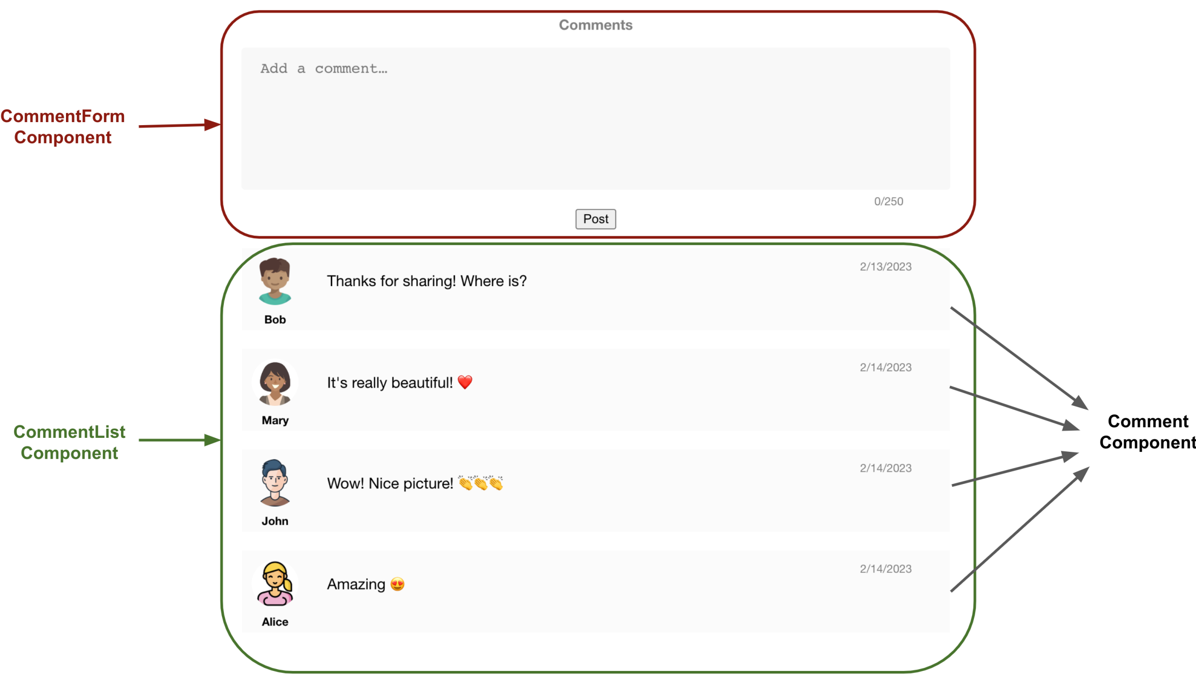
Listing 2.7: Component to create new comments.

```
1  function CommentForm(props){
2    const [comments, setComments] = useState([]);
3    const [commentValue, setCommentValue] = useState('');
4
5    const handleChange = e => {
6      setCommentValue(e.target.value);
7    };
8
9    const handleSubmit = e => {
10     e.preventDefault();
11
12     const newComment = {
13       user: {
14         name: props.name,
15         url: props.url
16       },
17       commentValue: commentValue,
18       date: Date.now()
19     };
20
21     setComments(comments.concat(newComment));
22     setCommentValue('');
23   }
24
25   return (
26     <div>
27       <form onSubmit = {handleSubmit}>
28         <h3>Comments</h3>
29         <textarea placeholder="Add a comment..." onChange={handleChange}
30         value={commentValue}></textarea><br />
31         <div className="limitChar">{commentValue.length}/250</div>
32         <button>Post</button>
33         <CommentList comments={comments} />
34       </form>
35     </div>
36   );
37 }
```

To add new comments, we create the `CommentForm` component (see Listing 2.7). This component gets user data (`name` and `url`) via `props` and uses the `useState` hook to track the current list of comments (line 2), including the user's input text. Since the list of comments is in the *state*, every time this list changes, the component view is updated.

Figure 2.2 summarizes the structure of components to display comments on the Web page. The `CommentForm` component has a form (lines 27-34 in Listing 2.7) that renders an input to receive new comments' data (lines 29-30), a button to submit the form (line 32), and the current list of comments via `CommentList` component (line 33). Handling events in React is similar to handling events with DOM elements. Thus, we use the `handleChange` and `handleSubmit` methods to catch changes in the input element and the form submission, respectively.

Figure 2.2: Structure of components to display comments



Since React relies on one-way data binding, the View changes are not automatically reflected in the model. For this reason, we use the `handleChange` method (lines 5–7) to keep the comment text updated in the model and to track how many characters the comment has (there is a limit of 250 characters). Thus, the input element (lines 29-30) receives the comment text, and the `handleChange` method propagates it to the state (lines 5–7) via the `setCommentValue` method.

On the other hand, when a user posts a comment, the `handleSubmit` method (lines 9–23) creates a new comment object using the user data received via *props* (lines 13–16) and the comment data in the *state* (line 17). Finally, the method adds the comment to the list of comments (line 21). Since this list is part of the component's state, the View is refreshed automatically.

2.1.3 Other Frameworks

Released by Evan You in 2014, Vue is a framework for building user interfaces that is entirely developed by an open-source community and not by a large enterprise. Similar to React, Vue also allows gradual adoption. For example, its core library also focuses on the View layer, simplifying integration with other libraries or existing projects. However, Vue relies on a Model–View–ViewModel (MVVM) architecture, with the ViewModel layer connecting the View and the Model via two-way data bindings. Vue also separates concerns under components. However, instead of writing HTML elements in JavaScript code through JSX, Vue offers an HTML-based template syntax that allows rendering data to the DOM declaratively. Thus, Vue templates are valid HTML that browsers can interpret.

Angular is an open-source front-end Web application framework for building dynamic and robust Web applications. The first version of Angular was released by Google in 2010. Then, in 2016, Google launched a new framework version, which was rewritten from scratch. Similar to React and Vue, Angular is a component-based framework. It provides developers with a set of tools and libraries that make it easier to build complex Web applications, including declarative templates, dependency injection, end-to-end testing, and more. In addition, it dictates a Model-View-Controller (MVC) architecture pattern, which separates data, presentation, and user interaction concerns [Krasner and Pope, 1988]. Furthermore, a two-way data binding mechanism keeps the view and model in sync. As mentioned before, this mechanism automatically reflects model changes in the view and vice-versa.

Created in 2011 by Yehuda Katz, Ember.js, commonly referred to as Ember, is an open-source JavaScript front-end framework that follows the Model-View-ViewModel (MVVM) pattern. Ember offers a complete solution for building JavaScript applications, including features such as templates, components, two-way data binding, computed properties, observers, and routes. One of the main features of Ember is its set of default conventions for organizing files, managing data, and routing, which makes it easier for developers to get started and maintain their applications as they scale.

Backbone was released by Jeremy Ashkenas in 2010 and therefore is the oldest framework we will consider in Chapter 3. One of the main benefits of Backbone is its simplicity. It is a lightweight open-source JavaScript framework and includes features such as models, views, collections, and events, which make it easier to manage data and update the UI of the application. Similar to React, Backbone supports only one-way data binding.

2.2 Related Work

The studies which approach the challenges of engineering modern Web systems using JavaScript front-end frameworks present several particularities. Thus, in the following subsections, we present the works we identified as relevant to this thesis. First, we describe studies that explore factors that can motivate the adoption and the unadoption of JavaScript front-end frameworks. Next, we discuss studies on code smells in Web technologies and strategies and tools to detect them. Finally, we present studies on refactorings in Web systems.

2.2.1 Adoption of JavaScript Front-End Frameworks

There is a fair amount of studies on the use and acceptance of technologies. For example, Polančič et al. [2011] investigated the characteristics and individual differences that impact the users' perceptions about object-oriented frameworks by using the Technology Acceptance Model (TAM) [Davis, 1989]. However, and particularly, the number of studies that explore the adoption of JavaScript front-end frameworks is limited. In an exploratory study, Graziotin and Abrahamsson [2013] claimed that these studies focused on benchmarks and other quantitative metrics, whereas practitioners also have interests in other aspects. For example, they argue that practitioners are driven by different concerns when choosing a JavaScript framework, such as the age of the latest release, the size of the framework, the license, the presence of features, and the browser support. Thus, they propose a comparison framework that combines researchers' interests with practitioners's interests to meet the best of both worlds.

Based on these findings, Pano et al. [2018] conducted a study on factors and actors that explain the adoption of JavaScript frameworks. They relied on semi-structured interviews with 18 developers. By using the Unified Theory of Acceptance and Use of Technology (UTAUT) [Venkatesh et al., 2003], they ended up with five major groups of adoption factors: (i) performance expectancy (performance and size), (ii) effort expectancy (automatization, learnability, complexity, and understandability), (iii) social influence (competitor analysis, collegial advice, community size, and community responsiveness), (iv) facilitating conditions (suitability, updates, modularity, isolation, and extensibility), and (v) price.

However, their study was conducted in 2014, when front-end frameworks were still in the early adoption phases. For example, the second most popular framework nowadays

(Vue) was not mentioned by any participant and a single participant mentioned the most popular framework nowadays (Facebook's React). The characteristics of these new frameworks can influence the factors of adoption.

Other studies evaluate other aspects of the frameworks, which can influence their adoption. For example, Gizas et al. [2012] conducted several quality, performance, and validation tests to evaluate general aspects of JavaScript frameworks (i.e., they are not only restricted to front-end implementation). The authors relied on well-established software quality metrics, such as size metrics (e.g., lines of code), complexity metrics (e.g., McCabe's Cyclomatic Complexity), and maintainability metrics (e.g., Halstead metrics). Their quality tests revealed that all frameworks have many functions with a high cyclomatic complexity and a low maintainability index, indicating that these functions probably need to be improved. In contrast, their validation tests focused on parts that must be modified to harmonize with browsers' continuous evolution. However, their study was conducted in 2012 and does not consider popular frameworks used to create SPAs.

In another study, Mariano [2017] compared three well-known frameworks (React, Angular, and Backbone) using benchmarks and complexity metrics. Using the three frameworks, the author implemented a benchmark application (a TODO application). Regarding performance, Backbone outperformed the other frameworks in their experiment (probably because it is a lightweight framework with just 6.5 KB plus 43.5 KB of required dependencies). In terms of complexity (measured using the cyclomatic complexity metric per function), ReactJS has the highest mean per-function measure (1.85), followed by Backbone (1.25) and Angular (1.16).

Nakajima et al. [2019] proposed a playground tool named Jact that enables developers to compare the runtime performance of JavaScript frameworks based on typical tasks in Web development. The authors argue that by sharing tasks and source code written by developers, Jact can continuously provide information related to JavaScript frameworks, including benchmark results. Zerouali et al. [2019] empirically analyzed the relationship between different software popularity measures of JavaScript libraries from three open source tracking systems (libraries.io, npm, and GitHub). They report that popularity can be measured using different metrics related to both social and technical aspects. However, the authors observed that many popularity metrics are not strongly correlated, implying that using different metrics may produce different outcomes.

Finally, there are websites that provide data on the popularity of JavaScript libraries and frameworks. For example, the most recent Stack Overflow Survey[5] shows that React.js and Vue are more popular than traditional MVC-based Web frameworks, such as Spring, Django, and Ruby on Rails. Another popular site with information on

[5] `https://survey.stackoverflow.co/2022/#section-most-popular-technologies-web-frameworks-and-technologies`

the usage of JavaScript-based technologies is State of JS.[6] The site annually runs a survey including mostly closed questions. Notably, the platform designates React as the foremost front-end framework in usage. Additionally, it has announced its intention to conduct a dedicated survey specifically addressing the state of React starting in 2023.[7]

> *Summary:* The last empirical study on the adoption of JavaScript front-end frameworks was conducted in 2014, when such frameworks were still in their early adoption phases. As the adoption of JavaScript front-end frameworks has evolved significantly since then, we decided to start this thesis by gathering up-to-date information on their adoption. To achieve this, we surveyed JavaScript developers to understand the current status of adoption and un-adoption of modern front-end frameworks. Chapter 3 provides more details on our survey study and findings.

2.2.2 Code smells in Web Technologies

There is a large body of papers, chapters, and books on code smells [Palomba et al., 2013; Moha et al., 2009; Liu et al., 2019; Fontana et al., 2015; Vegi and Valente, 2023; Nguyen et al., 2012; Sobrinho et al., 2018]. The principal one is a chapter by Fowler and Beck [1999], which proposes 22 code smells for object-oriented design and associates each one with a possible fixing refactoring. However, more recently, code smells have also been studied for Web technologies. These works study bad practices in HTML, CSS, JavaScript, and MVC frameworks. For example, Nederlof et al. [2014] investigated deviations from best practices in performance, accessibility, and correct structuring of HTML documents. Their findings reveal that most sites contain a substantial number of problems, making them unnecessarily slow, inaccessible for the visually impaired, and subjected to unpredictable layouts due to errors in dynamically modified DOM trees. For example, dynamic components might not be interpreted reliably by a wide variety of user agents, including assistive technologies, therefore impacting accessibility.

Many researchers have also focused on detecting duplicated content in Web pages or finding Web pages with similar structures [Boldyreff and Kewish, 2001; Lucca et al., 2001, 2002]. For example, Boldyreff and Kewish [2001] replaced the content of the Web pages (i.e., the content delimited by different tags) with hash values and compared them to find duplicated content. Synytskyy et al. [2003] use island grammar to find clones in

[6] `https://stateofjs.com`
[7] `https://stateofreact.com`

multilanguage Web documents. Lucia et al. [2005] use the Levenshtein distance to quantify the structural similarity between Web pages. Finally, to locate code clones in the source code of Web applications written in multiple languages, Rajapakse and Jarzabek [2005] utilize CCFinder, a clone detection tool that uses token-to-token comparison to find the clones. They examined 17 Web applications and found a 17% to 63% duplication rate.

Mesbah and Mirshokraie [2012] propose a technique to detect unused and ineffective CSS code automatically. Having a similar goal, Geneves et al. [2012] use tree logic to detect unused CSS code. Relate to JavaScript, Fard and Mesbah [2013] propose a set of 13 JavaScript code smells and a code smell detection tool, 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.[8]

Aniche et al. [2018] 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 also 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, 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. Sobrinho et al. [2018] conducted a systematic review of the literature on articles about code smells published between 1990 and 2017. The review identified 104 code smells within this timeframe, however, none of them refer to JavaScript front-end frameworks.

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 [1976] proposed the value ten as a threshold for the Cyclomatic Complexity metric based on his past experience. Other approaches, use real-world software systems to derive metric thresholds [Lanza and Marinescu, 2007; Fontana et al., 2015; Oliveira et al., 2014; Vale et al., 2019]. For example, Alves et al. [2010] 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. [2018] use the third quartile (3Q) and the interquartile range (3Q - 1Q) to

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

define the thresholds.

There are also approaches that rely on evolution history information to detect the smells, such as HIST [Palomba et al., 2013] and DECOR [Moha et al., 2009], and approaches that use Deep Learning [Fakhoury et al., 2018; Liu et al., 2019]. On the other hand, 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 [Fard and Mesbah, 2013] applies static and dynamic analysis for detecting code smells in JavaScript code. Cilla [Mesbah and Mirshokraie, 2012] is a tool that also relies on dynamic analysis for detecting unused CSS code. Another related tool is WebScent [Nguyen et al., 2012], which identifies six types of embedded code smells in dynamic Web applications.

There are also studies that evaluate code smells detection tools [Fernandes et al., 2016; Paiva et al., 2017]. For example, Paiva et al. [2017] evaluate and compare four code smell detection tools, namely inFusion, JDeodorant, PMD, and JSpIRIT. These tools were applied to different versions of the same software systems, namely MobileMedia and Health Watcher. The results show that the evaluated tools present different levels of accuracy according to the context. For example, for all smells in both systems, JDeodorant identified most of the correct entities, but reports many false positives. Finally, Sobrinho et al. [2018] studied a wide range of tools used in experimental settings for addressing code smells. However, a significant number of these tools do not provide access to their implementations, thereby limiting their applicability. Furthermore, from the perspective of open science, the unavailability of these implementations hinders reproducibility and creates obstacles to conducting empirical studies, especially those aiming to assess new approaches.

> *Summary:* Previous works investigated code smells in HTML, CSS, and JavaScript code, including, for example, duplicated code. However, to the best of our knowledge, none focuses on smells specific to JavaScript front-end frameworks that support the implementation of Single Pages Applications. Therefore, after confirming in Chapter 3 that these frameworks are extensively used today, we continued our research by exploring code smells specific to Facebook React, the most popular JavaScript front-end framework nowadays. For more information on our approach and findings, refer to Chapter 4.

2.2.3 Refactorings in Web Technologies

Refactoring is recognized as a fundamental practice to maintain a healthy code base [Fowler and Beck, 1999; Beck, 2000]. For this reason, a significant amount of empirical research was conducted to extend the knowledge of this practice [Murphy-Hill et al., 2011; Hora and Robbes, 2020; Kim et al., 2014; Alizadeh et al., 2018; Tsantalis and Chatzigeorgiou, 2009]. However, most of these studies have focused on mainstream programming languages, such as Java [Fowler and Beck, 1999] and JavaScript [Fowler and Beck, 2018]. For example, Silva et al. [2016] monitored Java projects hosted on GitHub to detect recently applied refactorings and asked the developers to explain why they decided to refactor the code. As a result, they compiled a catalog of 44 distinct motivations for 12 well-known refactoring types. Tsantalis et al. [2018] designed and implemented a tool called RMiner, which automatically detects refactorings in a project's commit history. To empirically evaluate the tool, the authors create an oracle of refactoring operations, comprising 3,188 refactorings found in 538 commits from 185 open-source projects. However, their approach is also constrained to a granular analysis of traditional refactorings in Java-based projects.

There are also studies that target refactoring opportunities in Web technologies, such as JavaScript [Fard and Mesbah, 2013], HTML [Nederlof et al., 2014], CSS [Mesbah and Mirshokraie, 2012], and MVC frameworks [Aniche et al., 2018]. For example, Harold [2012] published a book to explain how to use refactoring to improve virtually any Web site or application. They presented refactorings related to the layout, accessibility, validity, and well-formedness of Web sites or applications. Related to CSS, Mazinanian et al. [2014] 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 also show that duplication in CSS is widely common. Since the main practice to style React applications is by writing CSS styling separate from JSX files, their work also applies to React applications.

However, despite the papers mentioned above, refactoring practices related to Web technologies need to receive more research attention. For example, there is a relevant domain where refactoring has not been studied in depth before. It includes the front-end components that are part of modern Web UIs. Mainly the components implemented using JavaScript-based frameworks, such as React and Vue. For example, refactoring duplicated content in Web pages contributes to developers modularizing static UIs into independent and reusable components provided by such frameworks. However, as mentioned earlier, there is a lack of studies investigating both traditional and specific refactorings related to the context of JavaScript front-end frameworks.

For this reason, in Chapter 5, we investigate refactoring operations that developers

perform when maintaining and evolving React applications. A study by Tang et al. [2021] that empirically investigates refactorings and technical debt in machine learning systems and a study by Ksontini et al. [2021] that investigates refactorings in Docker projects served as our inspiration. Tang et al. [2021] empirically investigated common refactorings in 26 open-source machine learning (ML) systems. Their study aimed to identify specific and tangential refactorings related to ML performed in these systems. Similar to our study, the authors used commit logs containing the keyword "refactor*" and selected a random subset of these commits for manual analysis. The study revealed that code duplication was a major crosscutting theme that affects ML configuration and model code, which were then the most frequently refactored block of code. Additionally, the authors introduced 14 new ML-specific refactorings and seven technical debt categories.

Finally, Ksontini et al. [2021] also employed a similar methodology to study refactorings in 68 Docker projects and the related technical debt. First, the authors extracted commits containing the keywords "refactor" and "docker" in their log messages. Then, they manually analyzed 193 unique commits from different projects to identify refactorings. The study resulted in the documentation of 24 new Docker-specific refactorings and seven technical debt categories.

> *Summary:* Since there are no studies on code smells for React-based front-ends, it is not exactly a surprise that we also have not found relevant papers on refactorings specific to such key components of Web systems. Therefore, as a natural consequence of our previous study, we concluded our thesis by investigating novel refactorings specific to React components and by also adapting existing refactorings to this framework. For more details, we refer the reader to Chapter 5.

2.3 Final Remarks

In this chapter, we started by providing an overview of modern JavaScript front-end frameworks (Section 2.1). Notably, we presented the key characteristics of these frameworks and why they are used to build front-ends. In Section 2.1, we describe with examples the two most popular JavaScript front-end frameworks, React and Vue. Finally, we concluded by discussing in Section 2.2 studies that are closely relate to this thesis. First, we presented studies on the adoption of JavaScript front-end frameworks (Section 2.2.1). Then, we discuss studies on code smells in Web systems (Section 2.2.2). Last, we presented studies on refactorings in Web technologies (Section 2.2.3).

Chapter 3

Adoption of JavaScript Front-end Frameworks

As the first working unit of our PhD thesis, we decided to conduct a study on the adoption of JavaScript frameworks, as our research group had never worked with such frameworks before. Furthermore, as stated in Chapter 2, there are few Software Engineering papers related to the front-end of Web systems, particularly papers on Single-Page Applications and the frameworks used to implement them. Therefore, we start the thesis by conducting an exploratory study to gain a better understanding of the ecosystem of JavaScript front-end frameworks. In this study, **we aimed to comprehend—from a broader but practical perspective—the factors that motivate developers to choose a particular framework**. **We also investigated whether they have plans to migrate from one framework to another in the near future**. The second question was prompted by frequent comments suggesting that the JavaScript ecosystem is highly dynamic, with new frameworks emerging but also being abandoned very rapidly. Thus, we would not be interested in proceeding with the research if this fact turned out to be true.

This chapter is organized as follows. Section 3.1 describes our dataset, which includes JavaScript projects whose developers were surveyed about their motivation to adopt JavaScript front-end frameworks. In Section 3.2 we detail the survey conducted with developers. Section 3.3 reports the results of the survey conducted to reveal the major factors driving the adoption of these frameworks. The implications and lessons learned are discussed in Section 3.4. Section 3.5 describes threats to validity. Lastly, we conclude this chapter in Section 3.6.

3.1 Dataset

As the first step for creating a dataset with clients of JavaScript front-end frameworks, we retrieved the names that identify them in two popular package managers: NPM and BOWER. For this purpose, we randomly selected 10 GitHub projects that are clients of each framework (using GitHub's Used-by feature) and inspected their *package.json* and *bower.json* files. Table 3.1 describes the package names we found after this step.

Table 3.1: Package names used by the studied frameworks

Framework	**Package Names**
VUE	vue
REACT	react
ANGULAR	angular, @angular/core
BACKBONE	backbone
EMBER	ember, ember-cli, ember-cli-app-version

Next, we selected the top-15,000 most popular projects on GitHub ranked by stars, which is a metric commonly used to rank projects by popularity [Borges et al., 2016; Borges and Valente, 2018]. Then, we discarded 541 projects labeled as archived by GitHub. We also searched for forks, but we did not find anyone among the selected projects. Finally, for the remaining 14,459 projects ($15,000 - 541$), we checked out all versions of their *package.json* and *bower.json* files in order to retrieve the project's dependencies. If a project has never depended on any of the five frameworks of interest, it was discarded.

We found 1,515 projects that are (or were) clients of the studied frameworks, since they have (or had) a dependency to one of them. Table 3.2 shows the number of clients by framework.[1] As can be observed, REACT is the most popular framework (988 clients, 65.2%), followed by VUE (315 projects, 20.7%), and ANGULAR (263 clients, 17.3%). By contrast, EMBER has only 11 clients (0.7%).

However, for this initial study, we decided to randomly select at most 10% of these clients for each framework (or ten clients, selecting the greater value). The reason is that we will rely on this dataset to conduct a survey with developers and we do not intend to send a massive number of e-mails in order to avoid our research being perceived as spam [Baltes and Diehl, 2016]. Furthermore, the clients in this dataset are already popular, which minimizes the possibility of randomly selecting irrelevant projects.

After randomly selecting this sample of 10% of the clients, the author of this thesis carefully analyzed each one to discard non-software projects, archived projects,

[1] There are 116 projects that have at least two dependencies to the studied frameworks in their commit history.

Table 3.2: Number of clients by framework (Initial Selection)

Framework	# Clients
React	988
Vue	315
Angular	263
Backbone	68
Ember	11

and projects that use the frameworks only in examples, tutorials, documentation, and tests. After that, new projects were randomly selected to replace the discarded ones. This procedure was repeated until we reached the target number of clients for each framework or had no more projects to select. As a result, we selected a sample of 169 projects, as presented in Table 3.3. React has the highest number of clients (99 projects), followed by Vue (32 projects). By contrast, we found only one eligible Ember project.

Table 3.3: Dataset to study framework's adoption

Framework	# Projects
React	99
Vue	32
Angular	27
Backbone	10
Ember	1

3.2 Survey Design

To reveal the factors that motivate the adoption of front-end frameworks, we surveyed developers of the projects summarized in Table 3.3. To increase the chances of receiving accurate responses, we decided to send the survey only to the project's core front-end developers. To identify such developers, we used a Commit-Based Heuristic but taking into account only commits that change front-end-related source code files, according to the extensions listed in Table 3.4. In other words, we adapted the heuristic commonly used in the literature to identify core developers to the context of front-end files. According to the Commit-Based Heuristic, the core team is responsible for 80% of the overall amount of commits in a project [Joblin et al., 2017; Robles et al., 2009; Coelho et al., 2018]. Therefore, after our adaptation, core front-end developers are the ones responsible for at least 80% of the commits performed in front-end related files.

Table 3.4: Front-end files extensions

`*.js, *.jsx, *.ts, *.tsx, *.es, *.es6, *.mjs, *.vue`
`*.htm, *.html, *.xhtm, *.xhtml, *.css, *.scss, *.sass`

Next, we retrieved the email addresses of the core front-end developers from their GitHub profile. We sent only one email per project, addressed to its top core front-end developer, i.e., the one with the highest number of commits. We also sent at most one email to a given developer, i.e., whenever we found a core front-end developer who was emailed before—for another project—, we selected the next one. Again, our intention was to avoid developers perceiving our messages as spam.

In the email, we asked only two questions:

1. *Why did you choose [framework]?*

2. *Do you have plans to migrate to another framework? Please explain.*

With the first question, our goal is to reveal the factors that drive the adoption of JavaScript front-end frameworks. With the second question, we intend to unveil possible intentions to migrate to another framework in the future.

We sent 169 emails and received 49 responses, achieving a response rate of 29%. After collecting all responses, we analyzed the answers using thematic analysis, a technique for identifying and recording patterns (or "themes") within a collection of documents [Cruzes and Dyba, 2011; Cruzes and Dybå, 2011; Silva et al., 2016]. The analysis involves the following steps: (1) initial reading of the developer responses; (2) generating initial codes for each response; (3) searching for themes among codes; (4) reviewing the themes to find opportunities for merging; and (5) defining and naming the final themes. The author of this thesis performed these five steps. Then, two other researches reviewed the final classification. When quoting the answers, we use labels RA1 to RA49 to indicate the respondents.

3.3 Survey Results

3.3.1 Why did you choose [framework]?

We identified nine key factors that motivate the adoption of front-end frameworks in JavaScript. Table 3.5 summarizes these factors and provides a brief description of each one. Some respondents mentioned more than one factor. Thus, the number of occurrences in Table 3.5 is greater than 49.

Table 3.5: Factors motivating the adoption of front-end frameworks

Factor	**Description**	Occurences
Popularity	This framework is widely known and used	19
Learnability	This framework is easy to learn and use	17
Architecture	This framework forces clients to follow a solid architecture	15
Expertise	I have previous expertise in this framework	10
Community	This framework is maintained by a large community	9
Performance	This framework has excellent performance	8
To gain experience	I wanted to gain experience in this framework	6
Documentation	This framework's documentation is very good	5
Sponsorship	This framework is supported by well-known companies	4

To better understand our results, we also divided the factors by framework, as presented in Table 3.6. Then, in the following paragraphs, we describe and give examples of the top factors that influence the adoption of each framework.

REACT*:* We received 20 answers from REACT's developers. Popularity (8 answers), architecture (7 answers), and community (6 answers) are the most common adoption factors cited by them. As examples, we have the following answers:

> *I chose React because it was emerging as the most popular UI. Framework popularity doesn't always mean the best, but it does mean you have a larger talent pool to draw from, a larger pool of supporting libraries in the ecosystem, more tutorials, etc. (RA26 - Popularity)*

> *React allowed us to have a constant velocity whatever the size of the application, thanks to an architecture based on composition and the one-way data binding encouraged by Flux. (RA35 - Architecture)*

Table 3.6: Factors that motivate the adoption of JavaScript front-end frameworks (answers per individual frameworks).

Factor	Total	Ember	Backbone	Angular	Vue	React
Popularity	19	0	2	5	4	8
Learnability	17	0	1	0	11	5
Architecture	15	0	1	3	3	7
Expertise	10	0	1	2	4	3
Community	9	0	0	1	2	6
Performance	8	1	1	1	5	1
To gain experience	6	0	0	2	0	4
Documentation	5	0	1	0	3	1
Sponsorship	4	0	0	1	0	3

I/we are very happy with the stability and community that React offers (no major bugs we've seen and people are creating and maintaining libraries for all sorts of things). (RA44 - Community)

VUE*:* The framework stands out for its simplicity and ease of use. 11 (out of 14 respondents, or 78.6%) decided to use VUE because it is easy to learn. Other factors are performance with five answers, popularity, and previous expertise, each with four answers. As examples, we have the following responses:

Because Vue.js is much easier to start working with, one can easily transfer the team from existing technology to Vue.js. We've seen guys without previous Vue.js experience, building complete e-shops in 2–3 weeks. (RA17 - Learnability)

We chose Vue specifically because it felt simpler, easier to learn, and we already had someone in our team experienced with Vue. (RA18 - Learnability and Expertise)

Vue because the documentation is great, it is easy to learn and it is really performant. (RA24 - Documentation, Learnability and Performance)

Another reason to go for VueJS was its popularity, so developers can contribute more easily. (RA23 - Popularity)

ANGULAR*:* Popularity—not nowadays, but at the time of the adoption— was cited by five ANGULAR's respondents (50%), as in the following quote:

At the time Angular 1 was the hottest framework in town. This was before React, before Vue. (RA5 - Popularity)

BACKBONE*:* Similar to ANGULAR, developers adopted BACKBONE due to its popularity

at the time:

It was the most popular library at the time, and I had a lot of experience with it. (RA2 - Popularity and Expertise)

Ember*:* We received the following answer about Ember:

Ember was a fully featured framework, but If I had the option, I would probably have chosen React at the time or Angular if we were to build it now. (RA1 - Ember*)*

> *Summary:* Popularity (39%) and learnability (35%) are the main factors that motivate the adoption of front-end frameworks in JavaScript.

Weakness Factors: Beyond the factors that motivate adopting a JavaScript front-end framework, some respondents mentioned why they did not choose other frameworks. After carefully analyzing such answers, we identified eight factors considered as weaknesses by the respondents. Table 3.7 summarizes these factors for each framework.

Table 3.7: Weakness factors of JavaScript front-end frameworks (answers per individual frameworks).

Factor	Total	Ember	Backbone	Angular	Vue	React
Incompatibility of versions	7	0	0	7	0	0
Complexity	4	0	0	4	0	0
Suboptimal architecture	4	0	1	2	0	1
Lack of experts	4	0	1	0	0	3
Difficult to maintain	3	0	1	1	0	1
Lack of TypeScript Support	3	0	1	0	2	0
Poor performance	1	0	0	1	0	0
Security issues	1	0	0	1	0	0

In the following paragraphs, we briefly describe and give examples of the weak points of each framework.

React: Three developers cited the lack of experts as the key reason for not choosing React as their front-end framework. A clear example is the following answer:

The talent pool for experienced React devs is limited. (RA26)

Vue: At the time of the survey, two developers cited the lack of support to TypeScript as one of the Vue's drawbacks. However, this issue is solved in Vue 3, as also mentioned by the participant:

If the project grows much larger than current implementation in the future, Vue 3.0 might be adopted as it better supports TypeScript. (RA18)

ANGULAR: The incompatibility between Angular 1 and Angular 2 is a key factor for developers not choosing ANGULAR. Seven respondents mentioned this problem in their answers:

We were using AngularJS version 1, and when version 2 came out, with a completely new syntax that would have required rewriting most of the code, we decided to see if there were other options. (RA20)

BACKBONE: Three developers commented on BACKBONE's drawbacks. The difficulty to find developers with expertise in BACKBONE (1 answer), the framework's suboptimal architecture (1 answer), code maintainability problems (1 answer), and the lack of support to TypeScript were pointed as drawbacks of the framework.

EMBER: Only one developer comment about Ember:

I chose React for [my-project] in 2016 because it was emerging as the most popular UI framework. At the time, it was clear Ember would not reach the popularity that React and Angular were reaching. (RA26)

3.3.2 Do you have plans to migrate to another framework?

We received 46 answers for this question. Five (10.9%) out of 46 respondents explicitly expressed the intention to migrate to another framework: one respondent from EMBER to ANGULAR, one from BACKBONE to REACT, two from ANGULAR to REACT, and one from ANGULAR to VUE. The main reasons are difficulty in hiring developers and difficulty to maintain the codebase, as in this answer:

Yes, we already migrated our paid products to React and are going to update the open source codebase as well. The main reason for migration was (1) the simplicity of React and much more possibilities to create reusable component libraries; (2) It was hard to hire an engineer that knows Backbone; (3) It was a pain to maintain a huge codebase written using Backbone. (RA3)

Furthermore, 6 (13.0%) out of 46 respondents stated they already completed a migration before (one from ANGULAR to BACKBONE, two from JQUERY to ANGULAR, two from ANGULAR to VUE, and one from BACKBONE to ANGULAR and next to REACT). As example, we received this answer:

We were using AngularJS version 1, and when version 2 came out, with a completely new syntax that would have required rewriting most of the code, we decided to migrate to Vue. (RA20)

On the other hand, 35 respondents (76.1%) did not express intention to migrate to another framework. The main reasons are as follows: our system is working well (16 respondents), our project is in maintenance mode (3 respondents), and migrating to another framework requires a huge effort (4 answers). As examples, we have the following answers:

It would be nice to move to something TypeScript based, however as [my-project] is in maintenance only, there are no plans, only hopes. (RA2)

Not at all. The main goal is still to offer the best tool that solves a problem, and the roadmap is huge. Users do not really care which technology is used as long as it solves their problems and saves them some time. (RA6)

Rewriting it in either framework represents a huge work that nobody is willing to take at the moment. (RA10)

However, 7 (20%) out of 35 respondents that did not express the intention to migrate to another framework stated they would not use the same framework in a new project. As example, we have the following quote:

No plans to migrate [my-project]. However, for new projects that I have picked up since then and the ones I will pick up, my default choice is now Vue. The overall development experience of Vue feels way better to me and easier to wrap my head around than React. (RA33)

Interestingly, one respondent mentioned PReact, and one mentioned Svelte as an alternative in the future, for replacing React. We did not include these frameworks in our study because they are very recent, and thus they are not good candidates for the un-adoption study. For example, the respondent that commented on Svelte also indicated that five years are needed to make a decision:

In 5 years, if Svelte becomes extremely popular and the React ecosystem begins to shrink or become deprecated, we plan to consider a migration. (RA26)

> *Summary:* On one hand, it is true that there is a migration between frameworks in the ecosystem of Single-Page Applications implemented in JavaScript. For instance, 23.9% of the participants stated their intention to migrate to a new framework in the future. On the other hand, this rate is not as alarming and impactful, especially if we consider that all respondents were using first-generation frameworks like jQuery, Ember, Angular, and Backbone.

3.4 Implications

This section presents the implications of our study for practitioners, framework developers, and researchers.

For practitioners: our study provides interesting insights for practitioners that are interested in adopting a front-end framework in their JavaScript projects, as summarized in the following box:

Guidelines for adoption:

- Key factors: framework's popularity, learnability, and architecture.
- Other factors: previous expertise, community, performance, and support.
- Evaluate the quality of the framework's documentation.
- Analyze whether the framework's architecture fits the application's one.
- Analyze the team's expertise.

For framework developers: our study provides key lessons also to the developers of modern JavaScript frameworks. First, we show that the framework's adoption is guided by factors they do not directly control. For example, framework developers do not have direct influence on the popularity of their frameworks [Borges and Valente, 2018]. On the other hand, we also listed several factors that developers might improve, such as quality of the documentation, performance, and architecture.

For researchers: We also found a limited number of scientific papers on JavaScript front-end frameworks. Since they are fundamental components in modern JavaScript applications, we claim that researchers should also invest more time and effort on aspects such as architecture, design, performance, and documentation of such systems.

3.5 Threats to Validity

The first threat is related to the generalization of our results. In this study, we presented an analysis of the adoption of JavaScript front-end frameworks. As is typical in empirical software engineering studies, our dataset might not represent the entire population of projects that depend on JavaScript front-end frameworks. Although we considered five of the most popular and well-known frameworks, we cannot generalize our results to

other frameworks. To mitigate this threat, we employed a selection criterion of choosing the most popular projects ranked by stars for our initial dataset. However, future studies could also consider closed projects.

Another threat relates to facets that may affect our empirical results. We relied on information stored in *package.json* and *bower.json* files to identify the projects using the analyzed frameworks. These files include a list of required dependencies. However, we acknowledge it is possible to use the frameworks without a package manager, although this is not a recommended practice when building professional applications.[2] Additionally, detecting dependencies without package manager support is not trivial and error-prone.

A final threat relates to the selection of candidates for our survey. In our study, we rely on the maintainers' answers to characterize the adoption of front-end frameworks. However, some of the studied projects have hundreds of contributors with distinct responsibilities. Thus, we adopted a Commit-Based Heuristic over changes in front-end files to increase the chances of receiving accurate responses. However, we may have missed maintainers who do not contribute with code.

3.6 Final Remarks

JavaScript is the language that runs the Web. In the large, complex, and dynamic ecosystem created around the language, a remarkable class of applications are the frameworks widely used by front-end JavaScript developers to architecture and implement rich Web apps, usually called *Single-Page Applications*. With the increasing complexity of *Single-Page Applications* and the abundance of available frameworks, developers often face difficulties making informed decisions regarding framework selection, which can have significant implications for application maintenance and evolution. To address this gap, in this chapter, we studied the factors driving the adoption of such frameworks. Through a survey conducted with front-end developers, we gathered valuable insights into the ecosystem of JavaScript front-end frameworks. Our findings revealed a list of nine key factors that developers consider when selecting such frameworks. We found that the two main factors are popularity and learnability.

Furthermore, beyond the factors that motivate adopting a JavaScript front-end framework, we also found factors that negatively influence the choice of front-end frameworks, such as incompatibility of versions, complexity, sub-optimal architecture, difficult to maintain, and lack of experts. Our results not only contribute to the understanding of JavaScript front-end frameworks adoption but also provide practical guidance for de-

[2] `https://vuejs.org/v2/guide/installation.html`

velopers seeking to make informed decisions in their projects. Furthermore, framework developers can leverage these findings to better position their projects in this competitive software market.

Our datasets—including the survey responses in an anonymized format—are available at: `https://doi.org/10.5281/zenodo.4148591`

Chapter 4

Detecting Code Smells in React-based Web Apps

In this chapter, **we propose a list of code smells for React-based JavaScript applications**. We focus on React because it is currently the most popular JavaScript front-end framework [Hora, 2021].[1] Specifically, we aim to answer 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?

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 tool, called ReactSniffer, that detects the smells in JavaScript React-based applications. Then, we use ReactSniffer to unveil the most common code smells in React-based Web systems. Finally, we conducted a historical analysis to check how often developers remove the proposed smells.

This chapter is organized as follows. In Section 4.1, we present the procedure we used to define our catalog of React code smells. In Section 4.2, 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 4.3. Section 4.4 presents a field study by using ReactSniffer in ten GitHub projects. In Section 4.5 we present the results of a historical analysis and show how often developers remove these smells. In Section 4.6, we validate ReactSniffer's results with an experienced React Developer. In Section 4.7, we discuss the novelty of the proposed code smells. In Section 4.8, we detail threats to validity. Finally, we conclude this chapter in Section 4.9.

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

4.1 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 (4.1.1) and semi-structured interviews with professional software developers (4.1.2).

4.1.1 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) [Garousi et al., 2016; Barik et al., 2015; Kamei et al., 2021; Zhang et al., 2020].

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

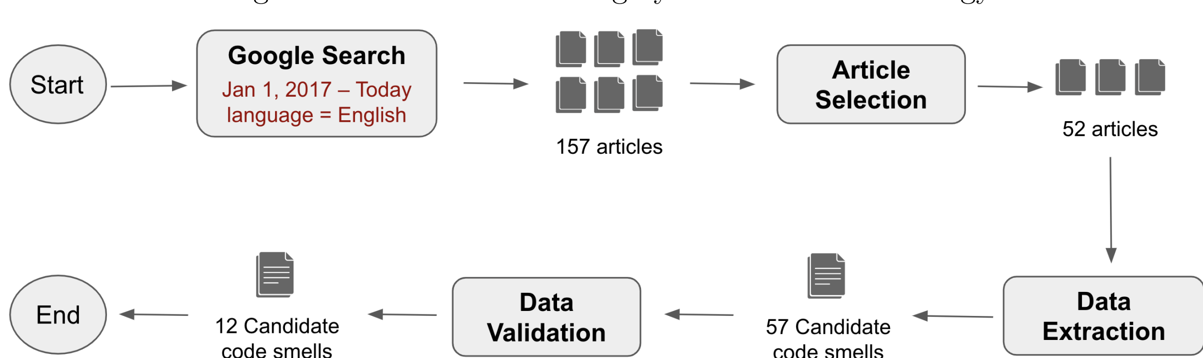
RQ1: What are the most common code smells when using REACT*?*

Figure 4.1 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 provide additional details on 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:

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

Figure 4.1: Overview of the grey literature methodology



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. [2019]. 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.[2] 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 author of this thesis 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 [Cruzes and Dyba, 2011; Cruzes and Dybå, 2011]. 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.

Data Validation: In the data validation step, a different research analyzed each article to validate each candidate code smell retrieved during the data extraction step. Initially,

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

Table 4.1: 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 unrelated elements and multiple responsibilities	A5, A32	2

we identified 57 candidate smells. However, in this validation step, we decided to consider only smells cited in more than one article. Consequently, we removed 35 candidates that did not meet this criterion, 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 4.1. The complete list of the selected articles and candidate smells is available at `https://doi.org/10.5281/zenodo.6985604`.

4.1.2 Semi-Structured Interviews

We interviewed six professional REACT developers recruited through the 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 author of this thesis conducted the interviews, taking from 27 minutes (minimum) to 43 minutes (maximum). We report data about each participant in Table 4.2.

Table 4.2: 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 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 4.3. 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.*

Table 4.3 highlights the smells validated with the developers as well some of the participants' comments. The JSX OUTSIDE THE RENDER METHOD and INHERITANCE

instead of Composition smells are only related to class components. Since this type of component is in disuse, they are found more in legacy code.

Table 4.3: 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.*
Large Component	*When a component grows, it is a sign that it needs to be broken.*
Inheritance Instead of Composition	*Using inheritance makes it difficult to reuse components elsewhere.*
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.*
JSX outside the render method	*It is a bad practice. Class methods with JSX should be components, also for reuse purposes.*
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.*
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.*
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.*
Uncontrolled Components	*It is common among inexperienced developers or when migrating to REACT to use uncontrolled components.*
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.*

New Smells: We also identified three new smells in the interviews: Large Bundlers, Lack of Accessibility, and Large Files, as summarized in Table 4.4. 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 4.2, we provide examples of each one.

Table 4.4: 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

4.1.3 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.[3] They refer to features very specific to React, including View updates, components that manipulate DOM directly, components that refer to HTML elements, components' state, and render methods.

Partially Novel Smell. We classified a single smell in our catalog as partially novel: Prop Drilling. For example, Ousterhout [2018] 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 [Fowler and Beck, 1999]. 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 [Gamma et al., 1994]), 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 [Fowler and Beck, 1999]. 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-known Long Parameter List smell, proposed by Fowler and Beck [1999]. 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.

[3] 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 4.4) — occur in Large Components.

4.2 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`.

4.2.1 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 4.2). 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 4.2), there are five different files to check for property updates, 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.*

Figure 4.2: Example of Prop Drilling (code smell).

App

<Gallery user={user} avatar={avatar} />

Gallery

<Image user={user} avatar={avatar} />

Image

<Comment user={user} avatar={avatar} />

Comment

<Author user={user} avatar={avatar} />

Author

4.2.2 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 4.1, `Comment` and `Opinion` have the same code.

Listing 4.1: Example of Duplicated Component (code smell).

```
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>
  );
}
```

4.2.3 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, mainly after the disuse of class components.

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` (see Listing 4.2).

Listing 4.2: Example of the Code Smell Inheritance instead of Composition.

```
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>
      )
   }
}
```

However, inheritance usually results a tight coupling between components [Gamma et al., 1994]. 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 4.3 shows the `Developer` component using composition instead of inheritance, which produces the same result.

Listing 4.3: Example of composition instead of inheritance.

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

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.*

4.2.4 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 4.4), 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`.

Listing 4.4: Example of JSX outside the Render Method (code smell).

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

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

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

4.2.5 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 4.5). This component has many properties, including the four properties presented in the Listing 4.5).

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 4.6.

Listing 4.5: Example of Too Many Props (code smell).

```
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 4.6: Example used to reduce the number of props.

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

4.2.6 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().*

4.2.7 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 4.7) is considered uncontrolled, since it uses a `ref` to get the comment value.

Listing 4.7: Example of Uncontrolled Component (code smell).

```
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>
    );
  }
}
```

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.

4.2.8 Low Cohesion

As usual in software design, the implementation of components must be cohesive and follow the Single Responsibility Principle [Martin et al., 2018]. In other words, the component's data and behavior must be related to make the component reusable and easy to understand.

4.2.9 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.*

4.2.10 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.*

4.2.11 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.

4.2.12 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.*

4.3 ReactSniffer: Code Smell Detection Tool

We implemented a prototype tool—called ReactSniffer—to detect the proposed smells. In this section, we describe its architecture (Section 4.3.1), thresholds selection policy (Section 4.3.2), and limitations (Section 4.3.3).

4.3.1 Architecture

As described in Figure 4.3, ReactSniffer architecture has three key components: a parser for generating the AST of React files, an AST Analyzer for collecting metrics and analyzing rules, and a Smells Detector module for identifying the smells.

Figure 4.3: 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 4.5.

The principal element of this parser is Babel,[4] a JavaScript compiler commonly used to convert new 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,[5] which generates an AST for JSX code.

Table 4.5: Front-end files extensions

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

The AST Analyzer module relies on the AST to search and inspect React elements. It recursively traverses the AST using a preorder algorithm. Finnaly, the Smells Detector module compares the component metrics against predefined thresholds to identify any potential code smells. To better understand ReactSniffer architecture, Listing 4.8 shows the pseudocode of the ReactSniffer tool and Listing 4.9 details the major steps and computations performed by ReactSniffer to detect smells in components.

Listing 4.8: ReactSniffer algorithm.

```
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 = astAnalyzer(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
```

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.

[4] https://babeljs.io
[5] https://babeljs.io/docs/en/babel-parser

Listing 4.9: Smell detection algorithm.

```
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
```

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 `innerHTML`, `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 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.

4.3.2 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 [Alves et al., 2010; Palomba et al., 2013], when the thresholds are derived from a dataset of systems. In this case, we need a dataset of systems (more details in Section 4.4.1). Our dataset comprises heterogeneous and relevant GitHub projects (the top-10 React projects ranked by stars). According to Mori et al. [2018], benchmarks composed of heterogeneous systems in terms of size and domains tend to have similar thresholds. Specifically, we compute the respective metrics for all systems and then used the 90th percentile value of each metric as a threshold. For example, the same approach is used by Alves et al. [2010] to characterize very-high risk code. On the other hand, Aniche et al. [2018] 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 conservative in our thresholds selection policy.

4.3.3 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 complex implementation or metric selection. For example, usually, there is a lack of consensus on recommended cohesion metrics [Pantiuchina et al., 2018; Cinnéide et al., 2012]. Regarding Duplicated Component and Prop Drilling, their detection depends on a static analysis of multiple files, a task that falls beyond the scope of this thesis. We plan to support these three smells in future versions of our tool.

4.3.4 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`.

4.4 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 4.4.1) and by using ReactSniffer to search for smells in these projects (Section 4.4.2).

4.4.1 Dataset

We intend to validate our catalog of smells with relevant GitHub projects. For that, we used the dataset of the study on the adoption of JavaScript front-end frameworks presented in Chapter 3, which contains 988 projects that depend on React. 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 4.6 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 4.6: 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/ANT-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 4.3.2. As a result, we obtained the values in Table 4.7, which are then used to obtain the results presented in the following section.

Table 4.7: 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

4.4.2 Results

Table 4.8 details the number of instances of each smell, as we found in our dataset. We briefly discuss the results for each one.

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.

Table 4.8: 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

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

Listing 4.10: Example of Uncontrolled Component found by the ReactSniffer Tool.

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

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

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 4.11:

Listing 4.11: Example of Force Update found by the ReactSniffer Tool.

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

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.

4.5 Historical Analysis

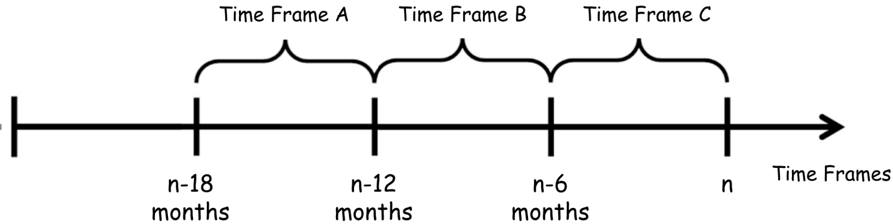
In the first RQ, we relied on a grey literature review and on interviews to come up with a catalog of 12 code smells for React-based systems. In the second RQ, we provided quantitative data on the frequency of each smell. This is important, for example, to provide guidance to developers on how often 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?*

To answer this research question, we deliberately decided to work with timeframes of six months. 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 4.4. 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.

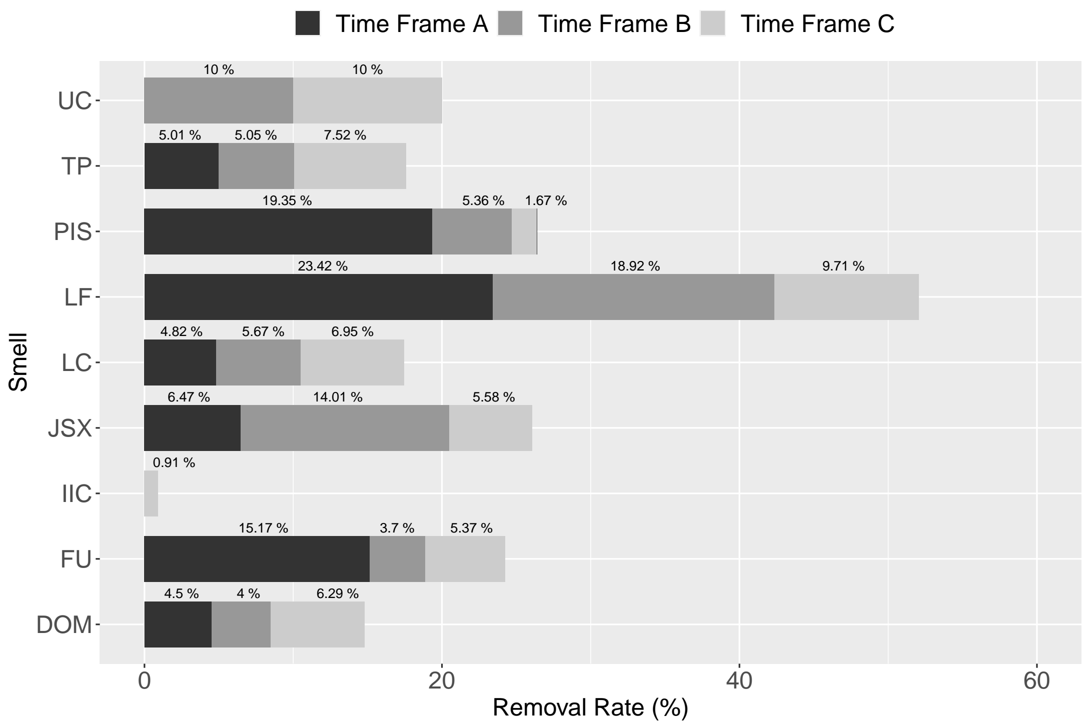
Figure 4.4: Time frames used in the analysis



To facilitate the visualization and analysis, we group the results of all projects by time frame. Figure 4.5 shows the overall removal rates.

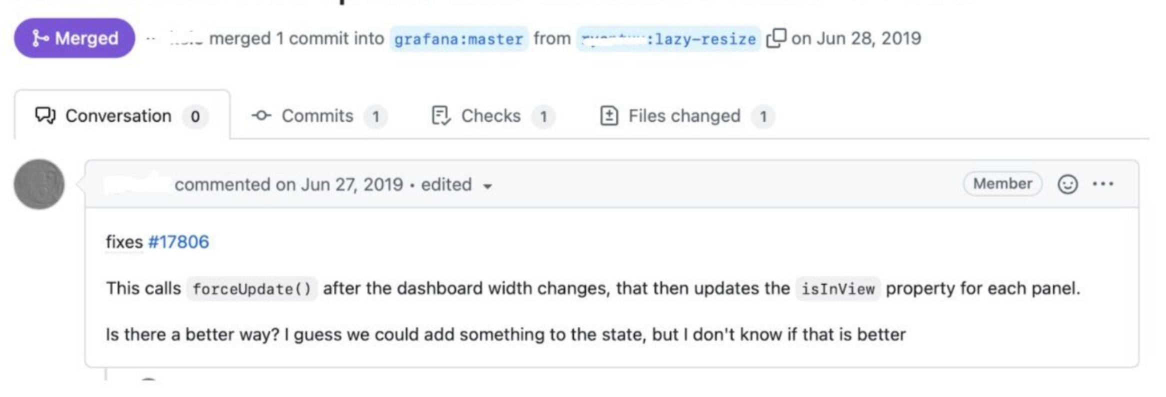
The removal rates range from 0.9% to 50.5%; the smell with the greatest removal rate is Large File (50.5%). In fact, all projects contain refactorings for large files. On the other hand, the Inheritance instead of Composition smell has the lowest removal rate (0.91%), and only one project contains a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

Figure 4.5: 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)



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 4.6 shows a developer from grafana project questioning the use of Force Update before it was refactored.

Figure 4.6: Issue questioning the use of Force Update



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 4.7 shows a refactoring removing an occurrence of this last smell with a comment self-admitting a technical debt.

Figure 4.7: Direct DOM Manipulation Refactoring

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

> *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%).

4.6 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 4.7. We also asked him to evaluate the tool results. Table 4.9 shows the number of front-end files (FF), and number of components of the selected project.

Table 4.9: 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 4.8 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 chart, there are no occurrences of the INHERITANCE INSTEAD OF COMPOSITION (IIC) and UNCONTROLLED COMPONENT (UC) smells.

Figure 4.8: 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)

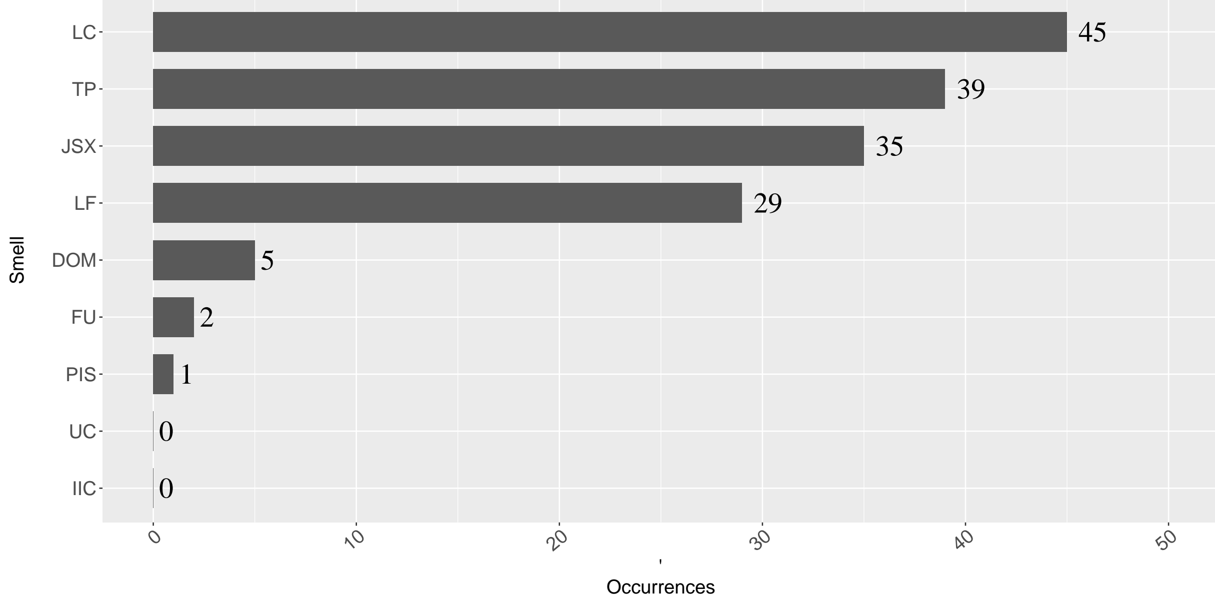


Table 4.10 shows the average relevance scores for each smell, as answered by the contacted developer. All smells have an average relevance score greater than 3.3. The exception is JSX Outside the render method (JSX), with an average score of 2.2. The developer justified that most JSX occurrences are 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.*

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.*

Table 4.10: Relevance scores of ReactSniffer Evaluation

	LC	TP	DOM	JSX	LF
Average	3.4	3.3	3.5	2.2	3.7
Standard Dev.	1.1	1.1	1.1	0.4	1.1

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

4.7 Discussion

Code smells are a widely studied concept, not only in object-oriented designs but also in particular domains, such as MVC-based applications [Aniche et al., 2018], JavaScript [Fard and Mesbah, 2013], HTML [Nederlof et al., 2014], and CSS [Mazinanian et al., 2014]. Therefore, a key discussion relates to the novelty of smells we identified for React systems. In this regard, in Section 4.1.3 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.

4.8 Threats to Validity

In this section we discuss threats to the validity of our results [Wohlin, 2012]. 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. [2019]. 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 empirical 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.

4.9 Final Remarks

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 [Fowler and Beck, 1999]. 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 25.9% 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 library. 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 chapter, 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 this chapter. 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 tool to handle all identified smells.

Chapter 5

Refactoring React-based Web Apps

In Chapter 4, we proposed a list of 12 common code smells describing design issues in React applications. Since refactoring is a well-known technique to improve software design and an indispensable practice in modern software development, in this chapter, **we propose a catalog of refactorings that can be applied to eliminate these smells and consequently improve the source code quality of front-end components**. Specifically, we set out to discover (i) the most important refactoring operations performed in React applications, (ii) how often these refactorings occur in a representative sample of open-source projects, and (iii) whether they are indeed frontend-specific program transformations or whether they are variations of traditional refactorings.

This chapter is organized as follows. In Section 5.1, we present the methodology we used to define our catalog of refactorings. In Section 5.2, we present this catalog. In Section 5.3, we discuss and put our findings and insights in perspective. In Section 5.4, we detail threats to validity. Finally, we conclude this chapter in Section 5.5.

5.1 Study Design

Our goal is to study React-specific refactorings that developers perform when maintaining and evolving Web apps. For that, we selected the top-10 React-based projects by stars from the dataset of the study on the adoption of JavaScript front-end frameworks presented in Chapter 3. For each selected project, Table 5.1 shows the number of front-end files (FF) and the number of components.

To identify commits that include refactorings, we initially collected all commits with changes in front-end files, *i.e.,* files with the extensions `*html`, `*htm`, `*.js`, `*.jsx`, `*.ts`, and `*.tsx`. Then, similar to Ksontini et al. [2021] and Tang et al. [2021], we selected, via `git log`, the commits containing the keyword REFACTOR* in their log messages. However, we also selected commits containing the keywords REENGINEER*, RESTRUCTUR*, and REORGANI*. The "Ref." column in Table 5.1 shows the number of

Table 5.1: Dataset of React clients (FF: number of front-end files; Comp: number of components; Ref: Number of refactoring commits)

Project	**FF**	**Comp.**	**Commits**	**Ref.**	**Analyzed**
grafana/grafana	914	1116	24,018	939	106
apache/superset	387	438	6,852	810	60
prometheus/prometheus	34	46	2,826	8	8
RocketChat/Rocket.Chat	532	815	4,530	70	17
ant-design/ant-design-pro	19	20	2,381	37	5
carbon-app/carbon	62	103	1,785	28	21
joplin/joplin	52	52	7,636	58	20
mitmproxy/mitmproxy	22	33	6,740	11	11
metabase/metabase	1159	1344	21,659	314	47
getredash/redash	295	352	1,844	42	25
Total	3,476	4,319	80,271	1,917	320

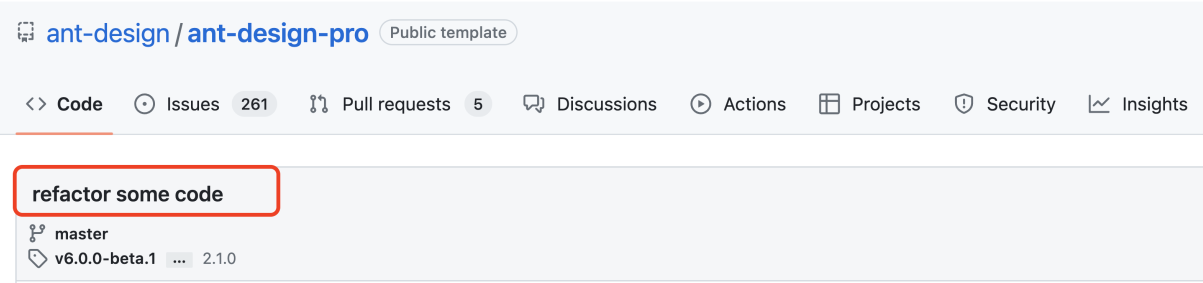
commits containing at least one of these keywords in their messages (1,917 commits in total).

Next, we randomly selected a subset of these commits to examine manually. However, as this analysis is manual, it is crucial to select a sample size that is neither too small nor too large. A small sample may not provide a representative picture of the commits and does not give a satisfactory level of accuracy, while a large sample demands increased costs and time to conduct a manual analysis. Therefore, we used a sample size calculator with a 95% confidence level and a 5% margin of error to determine the optimal subset size. Using this methodology, we selected 320 commits out of a total of 1,917 to examine manually, striking a balance between minimizing costs and time while ensuring statistical confidence. The "Analyzed" column in Table 5.1 shows the number of commits selected by project.

Due to the large and complex nature of many commits, the author of this thesis carefully reviewed each commit message to identify only those commits that were clear instances of refactoring. This process involved analyzing each commit message to determine whether it contained unambiguous descriptions of the refactoring operations being performed. As a result, only those commits with clear and explicit descriptions of their refactoring activities were selected for further analysis. In other words, some commits were discarded because their logs contained refactoring-related keywords but the exact refactorings were not precisely described. For example, some logs contained statements such as "should refactor the UI code," which is not a specific refactoring operation. Other commits could potentially represent refactorings. However, due to the lack of clarity regarding the nature and location of the refactoring, they were also discarded. Figure 5.1 shows a second example of an unclear log message related to a discarded commit.

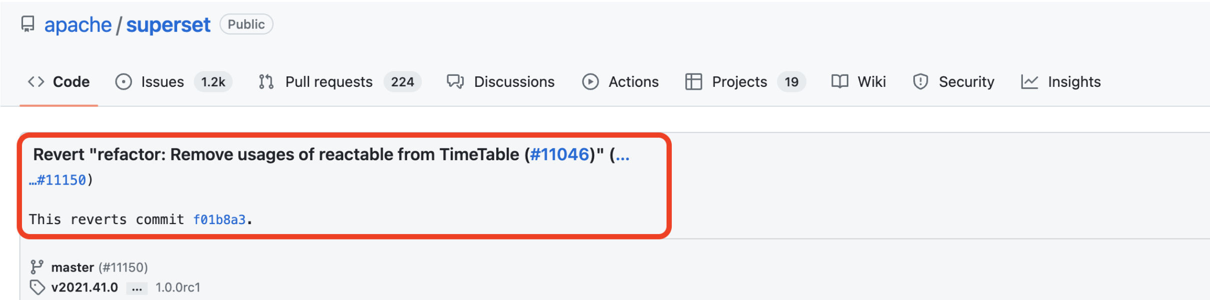
After the author's initial examination, 65 commits were discarded. To ensure accu-

Figure 5.1: Example of an unclear commit.



racy, a second researcher conducted a thorough review of these commits to confirm their elimination. Upon review, he agreed with discarding 56 commits (86%) but claimed nine commits could potentially be considered in the analysis. Then, after discussions between the author of this thesis and the second researcher, six of the nine cases were ultimately reclassified as valid commits. Out of the three remaining commits, one introduces a change in functionality that does not preserve behavior, another commit description was considered unclear, and the third revoked a previous refactoring operation, as illustrated in Figure 5.2. Consequently, the final classification resulted in 261 commits whose descriptions clearly indicate refactorings.

Figure 5.2: Example of a discarded commit because it is revoking a previous refactoring operation.



In the subsequent stage of the analysis, the author of this thesis and a third researcher undertook a detailed and independent examination of the 261 selected commits. This process involved conducting an independent review of each commit to identify the specific refactoring operations and their underlying rationale. To determine the nature of the operations, the researches primarily studied the commit diff, carefully analyzing the changes made to the codebase. They also examined issues linked to the commits (if any), as they often provide valuable context for the commit activities.

After completing the analysis, the researchers discussed their classification and addressed the disagreements that emerged during the process. Notably, in nine instances the authors assigned different labels to the commits despite them representing the same refactoring operation. Through collaborative discussions, a consensus was reached to use

the same names, as summarized in Table 5.2. In this table, we underline the final name choice.

Table 5.2: Example of refactoring operations labeled with different names (the final selected version is underlined).

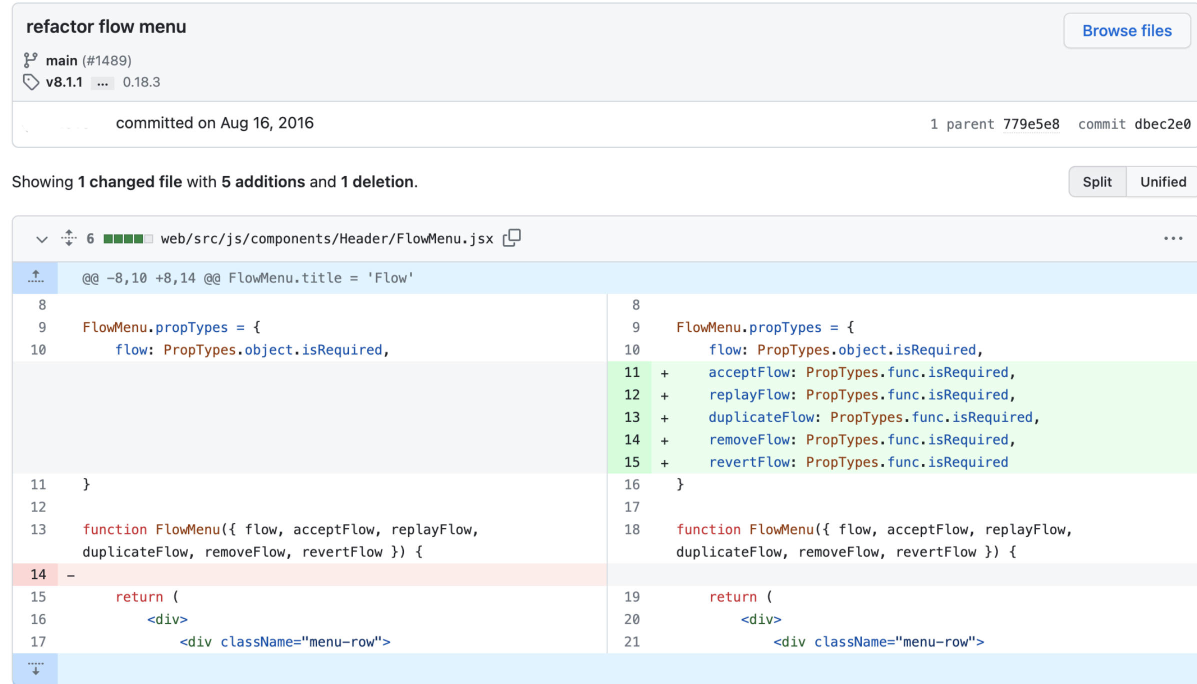
First researcher classification	Second researcher classification
Merge Components	Combine components into one
Convert JS code in TS	Migrate to Typescript
Convert class component into function component	Migrate to functional component
Extract HTML to Component	Migrate HTML to react
Remove directly state updates	Remove state usage
Dead Code Elimination	Remove dead code
Replace EOL to semi-colon format	Change style format
Convert function component into class component	Migrate to class style
Extract Logic to a Custom Hook	Extract Custom Hook

After this names' standardization, the authors reached a consensus on 234 commits, indicating a substantial level of agreement (89.6%). However, there were 27 commits in which the researchers held differing opinions. The disagreements were often attributed to instances where one researcher failed to identify a refactoring activity that the other researcher had recognized. Another case of disagreement emerged when one researcher identified an activity as a refactoring while the other researcher did not recognize it as such. For instance, one researcher attributed the labels "Add parameter" and "Change parameter" to certain commits. However, the other researcher argued that these operations do not preserve behavior. They reviewed these conflicting cases, discussed their reasons for disagreement, and sought a consensus on the correct classification for each one.

Furthermore, in this second classification stage, the researchers classified ten commits as false positives (3.8%). As example, five commits describe a refactoring operation but the changed code indeed adds a new feature or change an existing one. For instance, Figure 5.3 illustrates a commit that introduces a restriction, resulting in a change of behavior. This change mandates that all these properties will be mandatory. This modification alters the behavior. Other five commits include refactorings performed only in the back-end code. After eliminating these cases, we found 565 refactoring instances in the remaining 251 commits.

Finally, the researchers classified the refactoring instances into four major categories:

Figure 5.3: Example of a false positive commit that does not preserve behavior.



- React-specific refactorings (134 instances, 23.7%), which are novel refactorings that only occur in front-end code.

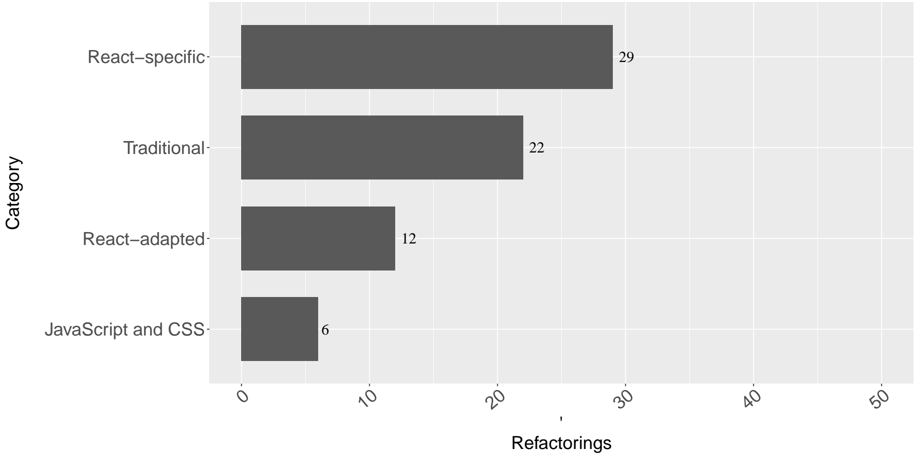
- React-adapted refactorings (214 instances, 37.8%), which are refactorings that although related to the React context are adaptations of traditional refactorings.

- Traditional refactorings (192 instances, 33.9%), *i.e.*, refactorings documented in Fowler's catalog.

- JavaScript-specific refactorings (22 instances, 3.8%) and CSS-specific refactorings (3 instances, 0.5%), which are refactorings related to JavaScript and CSS code and structures and that were not previously classified as React-specific or React-adapted

We identified a total of 69 distinct refactoring operations. The distribution of these refactoring operations by category is illustrated in Figure 5.4. In the subsequent section, we provide a comprehensive overview of the proposed catalog, delving into the specifics of more frequent refactoring operations.

Figure 5.4: Number of refactoring operations by category



5.2 A Catalog of Refactorings for React-Based Web Apps

In the following sections, we discuss the most frequent React-specific (Table 5.3) and React-adapted (Table 5.4) refactorings. In addition, we briefly comment on the JavaScript-specific (Table 5.5) and on the traditional (Table 5.6) refactorings.

5.2.1 React-specific Refactorings

In this section, we focus on the refactoring operations that are specific to React code. Based on our analysis, which is detailed in Table 5.3, we have identified a total of 134 instances involving 25 unique React refactoring operations. These operations are exclusive to front-end code and have been observed multiple times, indicating their relevance and practical applicability in React development. Since React documentation advises against using class components in new codebases, among the 25 React-specific refactoring operations, three directly relate to the abandonment of class components: Migrate Class Component to Function Component, Generalize interface to accept class and functional component types, and Extract JSX outside render method component.

In the following subsections, we will present the React refactorings that have been observed in more than one instance during our analysis.

Table 5.3: React-specific refactorings

Refactoring	**Occur.**
Extract Stateful logic to a custom hook	47
Migrate class component to function component	33
Migrate Angular to React Component	7
Replace third-party component with own component	5
Extract conditional in render	4
Remove props in initial state	4
Migrate to styled component	4
Memoize component	4
Extract higher-order component (HOC)	3
Remove direct DOM manipulation	3
Replace access state in setState with callbacks	3
Replace direct mutation of state with setState()	2
Remove forceUpdatte()	2
Replace Stateful logic to hook	2
Extract logic to a custom context	1
Split hook	1
Generalize hook	1
Replace callback bind in constructor with bind in render	1
Replace HTML/JS code with third-party components	1
Migrate function component to class component	1
Migrate React.FC to functional component syntax	1
Move reducer	1
Replace id with Routers	1
Generalize interface to accept class and functional component types	1
Replace dynamic keys with stable ids	1
Total	134

Extract stateful logic to a custom hook

React hooks were introduced in React 16.8 to use state and other features without needing class components. For example, the `useState()` hook allows tracking state in function components. However, the logic that deals with state might become duplicated in components. For example, in a chat application, more than one component may store data about the status of the users. Moreover, these components may also replicate the logic that checks whether a user is online or not. React Hooks allows developers to eliminate this duplicated logic by extracting it to a custom hook, which is a function whose name starts with "`use`" (*e.g.,* `useFriendStatus`). This hook usually returns the state and the function to update it (*e.g.,* `const [friendStatus, setFriendStatus] = useFriendStatus()`). Custom hooks improve reusability since the same code that appears in multiple components can be implemented in a single function. We classify this refactoring as specific to React because it is tightly connected to React's key

abstractions, such as hooks, function components, and states.

For example, several components in the REDASH project need to load geolocation data and add it to the `geoJson` components state. Initially, each component had its geolocation state and the loading. Then, a refactoring was performed to extract the state and the associated logic to a single custom hook, called `useLoadGeoJson`, as illustrated in Figure 5.5. We found 47 occurrences of this refactoring.

Figure 5.5: Refactoring that extracts a stateful logic to a custom hook

```
9  + export default function useLoadGeoJson(mapType) {
10 +   const [geoJson, setGeoJson] = useState(null);
11 +   const [isLoading, setIsLoading] = useState(false);
12 +
13 +   useEffect(() => {

36 +   }, [mapType]);
37 +
38 +   return [geoJson, isLoading];
39 + }
```

(a) `useLoadGeoJson` custom hook

```
8  + import useLoadGeoJson from "../hooks/useLoadGeoJson";
9  + import { getGeoJsonFields } from "./utils";
10
11    export default function GeneralSettings({ options, data, onOptionsChange }) {
12 +   const [geoJson, isLoadingGeoJson] = useLoadGeoJson(options.mapType);
```

(b) The component GENERALSETTINGS using the `useLoadGeoJson` hook

Migrate class component to function component

REACT supports class and function components. A class component is an ES6 class with lifecycle control methods (*e.g.*, `componentDidMount()` and `componentDidUpdate()`), local state, and a render method that returns what must appear in the UI. On the other hand, a function component is just a JavaScript function that accepts props (or inputs) as arguments and returns a REACT element representing the UI. For this reason, function components are simpler to understand than class components. Moreover, by using hooks, function components can access state and other REACT features.

For these reasons, replacing class components with function components is a common REACT-specific refactoring, with 33 occurrences in our dataset. Figure 5.6 shows an example that replaces the `CreateUserDialog` class with a function component. Specifically, the refactoring (1) changes the class to a function, (2) removes the render method, (3) removes references to `this`, (4) removes the constructor and replaces the state with a `useState` hook, (5) replaces the `componentDidMount()` lifecycle method with a `useEffect` hook.

Figure 5.6: Refactoring `CreateUserDialog` class component to function component

9	`- class CreateUserDialog extends React.Component {`	8	`+ function CreateUserDialog({ dialog }) {`
10	`-   static propTypes = {`	9	`+   const [error, setError] = useState(null);`
11	`-     dialog: DialogPropType.isRequired,`	10	`+   const formRef = useRef();`
12	`-     onCreate: PropTypes.func.isRequired,`		
13	`-   };`		
14		11	
15	`-   constructor(props) {`	12	`+   useEffect(() => {`
16	`-     super(props);`		
17	`-     this.state = { savingUser: false, errorMessage: null };`		
18	`-     this.form = React.createRef();`		
19	`-   }`		
20	`-`		
21	`-   componentDidMount() {`		
22	`      recordEvent("view", "page", "users/new");`	13	`      recordEvent("view", "page", "users/new");`
23	`-   }`	14	`+   }, []);`
24		15	
25	`-   createUser = () => {`	16	`+   const createUser = useCallback(() => {`

Migrate Angular to React Component

This refactoring operation transforms a component developed using the Angular framework into a React component. Therefore, it is commonly performed when developers migrate an Angular-based application to a React-based one. In our analysis, we identified seven instances of this refactoring operation in two projects that underwent a migration from Angular to React.

Replace Third-party Component with Own Component

While developing a React Web App, developers can use components from a component library. This refactoring replaces a third-party component with an in-house component developed by the team. We found five occurrences of this refactoring.

Extract Conditional in Render

React allows conditional rendering of UI elements, depending on the application's state—for example, a set of UI elements is rendered only when the user is logged in. However, mixing JSX code with nested conditional rendering makes the code hard to read and maintain. In our dataset, we found four refactorings that simplify such conditionals by extracting the JSX code to other components or helper methods.

Remove Props in Initial State

Initializing the state with props makes the component ignore all props updates. If the props values change, the component renders its initial values. We found four refactorings that eliminate the initialization of state with props.

Migrated to Styled Component

Styled-components is a React-specific CSS-in-JS styling solution that allows developers to write CSS code to style React components. The required dependencies from the styled-components package must be imported to create a styled component, which involves defining a new component using the styled object or utility functions. [1] For example, a button HTML tag and its original CSS class names can be replaced by a styled component.

The following code defines a styled component to create a button.

```
import styled from 'styled-components';

const StyledButton = styled.button`
  background-color: blue;
  color: white;
`;
```

Then, we can replace the corresponding HTML element tag with the newly defined styled component name, as showed next:

```
// Before
<button className="originalButtonClass">Click Me</button>

// After
<StyledButton>Click Me</StyledButton>
```

We found four occurrences of migration to styled components.

Memoize Component

This refactoring focuses on enhancing the performance of a React component through the application of memoization. This technique is crucial in optimizing the rendering process, particularly for components involving computationally intensive tasks and complex rendering logic. By caching callbacks and the results of expensive computations, memoization prevents unnecessary renderization of components. In addition, this caching mechanism enables the reuse of cached values when the inputs to those computations remain unchanged, resulting in improved overall performance. We found four refactoring operations for memoize component.

Extract Higher-Order Component (HOC)

A higher-order component (HOC) is an advanced technique in React for reusing component logic. It takes a component as an input and returns a new enhanced component. React allows writing custom HOC or reusing HOCs from third-party React libraries, such

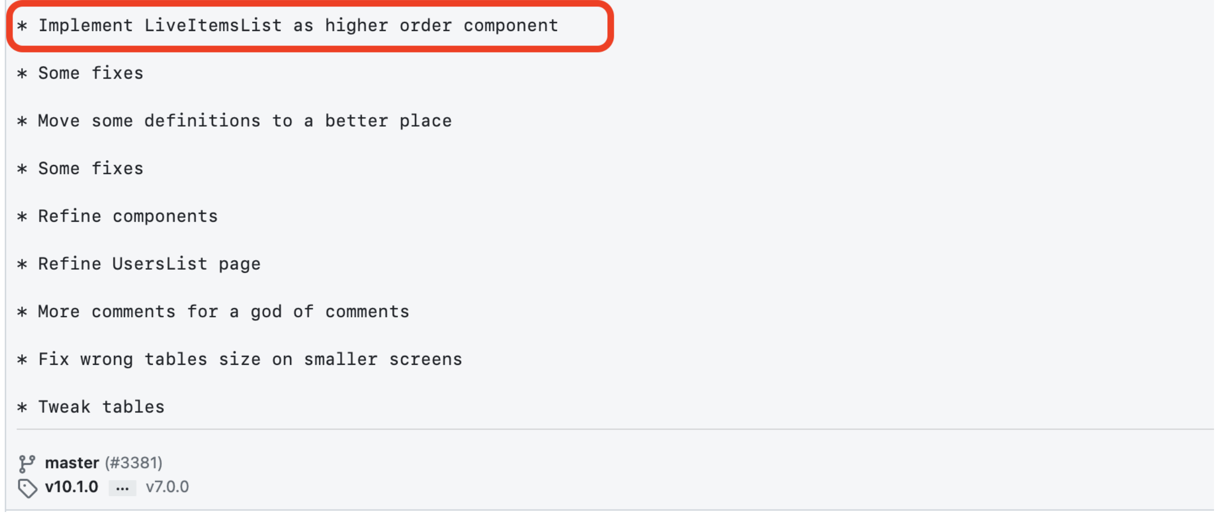
[1] https://styled-components.com

as Redux's connect.[2] For example, the following code use the `higherOrderComponent()` function that takes a component as an input (`WrappedComponent`) and returns a new enhanced component.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

Thus, HOC can be used to wrap around other components and provide additional functionality or data to them. Thus, the refactoring Extract higher-order component involves extracting common functionality from multiple components into a HOC, avoiding code duplication and making it easier to manage and update the shared functionality. Figure 5.7 shows a commit message indicanting the extract of an HOC called LiveItemsList, which wraps common logic for the DashboardList, QueriesList, and UsersList components

Figure 5.7: Commit message indicanting an Extract HOC refactoring



Remove Direct DOM Manipulation

React uses its own representation of the DOM, called virtual DOM. When the state changes, React updates the virtual DOM and propagates the changes to the real DOM. However, manipulating the DOM using standard JavaScript code can cause inconsistencies between React's virtual DOM and the real DOM. We found three refactoring that removes direct DOM manipulation.

Replace Access State in setState with callbacks

In React, accessing state in the `setState()` method can lead to inconsistencies because such updates are asynchronous, *i.e.*, React can batch multiple `setState()` calls into a

[2]https://react-redux.js.org/api/connect

single update to increase performance. Therefore, if two `setState` operations are grouped they both can access the old state. For example, the following code may fail to update the `counter` state:

```
this.setState({counter: this.state.counter + 1}) // 2
this.setState({counter: this.state.counter + 1}) // 2, not 3
```

The recommended refactoring uses a variation of `setState()` that accepts a function rather than an object. This function receives the previous state as an argument:

```
this.setState(prevState => ({counter: prevState.counter + 1}));
```

We found three occurrences of this refactoring.

Replace Direct Mutation of State with setState()

Class components provide the `setState()` method, which updates the component state and indicates to the framework what needs to be re-rendered. To support this process, React keeps the previous state and compares it with the updated state to decide whether or not the component needs to be re-rendered. The problem occurs when the state is changed directly *i.e.,* without calling `setState()`. As a result, the component may not reflect the state updates.

Therefore, the solution is always use `setState()` to update the state. We found two occurrences of direct mutation of the state being replaced with mutation using the `setState()` method.

Remove `forceUpdatte()`

In order to automatically reflect model changes in the view, React re-renders a component only if its state or the props passed to it changed. However, developers can force the update of components or even reload the entire page, which may cause inconsistencies between the model and view. React documentation recommends to avoid all uses of `forceUpdate()` and to only access `this.props` and `this.state` in `render()`. We found two refactorings that eliminate calls to `forceUpdate()` or `reload()`.

Replace Stateful Logic to Hook

This refactoring involves replacing a custom logic or functionality implemented within a React component with an existing hook. By employing a hook, the component can utilize pre-existing functionality provided by third-party libraries or implemented by the team, which helps in reducing redundancy and promoting code reusability. We found two occurrences of this refactoring.

5.2.2 React-adapted Refactorings

In our analysis, we found 214 instances of 17 refactoring operations that, although related to the React context, are adaptations of traditional refactorings. For this reason, we call them as React-adapted refactorings. For example, React structures and organizes the UI through the abstraction of component composition. Consequently, the Extract Component refactoring remembers the Extract Class operation. However, a component is not exactly a class. Table 5.4 shows such refactorings and highlights the traditional refactoring they are similar to.

Table 5.4: React-adapted refactorings

Refactoring	**Similar to**	**Occur.**
Extract component	Extract Class	76
Rename component	Rename Class	30
Remove unused props	Remove unused parameter	24
Move Component	Move Class	24
Rename prop	Rename parameter	12
Split component	Extract Class	9
Move Hook	Move Method	8
Extract HTML/JS code to component	Extract Class	7
Extract JSX outside render method to component	Extract Class	6
Rename hook	Rename Method	6
Combine Components into one	Combine Functions into Class	3
Remove unused state	Remove unused field	3
Rename state	Rename field	2
Remove unused useEffect	Dead Code Elimination	1
Remove unused hook	Dead Code Elimination	1
Change prop type	Change variable type	1
Replace value with props	Change Value to Reference	1
Total		214

Extract Component

This refactoring occurs when parts of a component appear in multiple places. Therefore, extracting these parts into a new component allows their reuse in other places. We found 76 occurrences of this refactoring in our dataset.

Rename Component

We found 30 refactorings that rename components. This refactoring usually occurs when the name of a component does not represent the component well, either because the component was poorly named or because its purpose evolved and the original name finished being a good choice.

Remove Unused props

Passing props to child components is common when developing and maintaining React applications. However, as the application evolves, some props may become unused due to changes in the component's logic or requirements. These unused props can clutter the codebase, making it harder to understand and maintain the component. We found 24 occurrences of a refactoring that eliminates unused props.

Move Component

This refactoring is recommended when a component is used in multiple files. In such cases, we should consider moving the component to the location where it is most used. We found 24 occurrences of this refactoring.

Rename Props

This refactoring is similar to a traditional rename refactoring. It occurs when the name of a prop does not represent its purpose very well. We found 12 occurrences of this refactoring.

Split Component

This refactoring occurs when a component starts getting too large, with many responsibilities, making it hard to maintain. We found nine occurrences of this refactoring in our dataset.

Move Hook

This refactoring is recommended when a hook is used in multiple components. In such cases, we should consider moving the hook to the location where it is most used. We found eight occurrences of this refactoring.

Extract HTML/JS Code to Component

In Web apps, duplicated UI elements are also a common design problem. For example, some buttons in an app might be very similar, changing only details such as text and

image. The problem happens when the HTML/JS code that implements these buttons is duplicated, making it more difficult to maintain, reuse, and evolve. Thus, refactoring duplicated UI code to components fosters reuse and encapsulation. We found seven refactorings extracting HTML/JS code to reusable components.

Extract JSX Outside render Method to Component

This refactoring enables the reuse of helper methods with JSX code in other components. In React, the render method—which is the only method required in a class— returns a JSX template describing what should appear on the UI. However, when this method becomes large, developers sometimes move part of its code to separate methods, which prevents reuse decoupled from the render. Therefore, extracting these methods to new components improves reusability and allows their reuse in other pages. We found six occurrences of this refactoring in our dataset.

Rename Hook

This refactoring is also similar to a traditional rename refactoring. It occurs when the name of a hook does not represent its purpose well. We found six occurrences of this refactoring.

Combine Components into One

This refactoring is recommended when two or more components share UI elements and logic, *i.e.,* when we have code duplication. The solution is to create a new common component and move the duplicated UI elements and logic to it. We found three occurrences of this refactoring.

Remove Unused State

We found three refactoring operations that remove unused states, *i.e.,* state variables that have been declared but are no longer used or referenced within the component's logic.

Rename state

This refactoring occurs when the name of a state does not represent its purpose well. We found two occurrences of rename state.

5.2.3 JavaScript and CSS Refactorings

As described in Table 5.5, we found 22 instances of four refactoring operations specific to JavaScript and two refactoring operations specific to CSS, *i.e.,* they are not directly related to React-specific code. For example, the most common JS refactoring is Convert JavaScript Code into TypeScript, with 13 occurrences.

Table 5.5: JavaScript and CSS refactorings

Refactoring	**Type**	**Occur.**
Convert JS code in TS	JS	13
Migrate function to arrow function syntax	JS	4
Replace promises with useCallBack	JS	1
Replace EOL to semi-colon format	JS	1
Rename CSS Class	CSS	2
Extract Stylesheet	CSS	1
Total		22

5.2.4 Traditional Refactorings

As summarized in Table 5.6, we also found 192 instances of 22 refactoring operations documented in Fowler's catalog Fowler and Beck [1999]. Dead Code Elimination is the most common one, with 83 occurrences.

5.3 Discussion

In this section, we discuss our results under two main dimensions. First, we map the proposed refactorings to a set of React-based code smells. Next, we discuss how our findings can be generalized to other frameworks, such as Vue.

Table 5.6: Traditional refactorings

Refactoring	**Occur.**
Dead Code Elimination	83
Move Function	20
Extract Function	16
Rename Function	13
Consolidate Conditional Expression	11
Duplicated Code Elimination	7
Rename Method	6
Rename Variable	6
Extract Method	4
Rename Type	4
Rename Parameter	4
Move File	3
Rename File	3
Rename Object Fields	2
Move Type Definition	2
Replace magic literal	2
Merge Methods	1
Move Method	1
Rename Interface	1
Encapsulate fields in Object	1
Replace custom logic with external lib	1
Use Composition instead of Inheritance	1
Total	192

5.3.1 React Code Smells and Refactorings

In chapter 4, we proposed a catalog of code smells for React-based Web applications. We also implemented a tool to detect these code smells. The catalog and detection tool serve as valuable resources for front-end developers, alerting them to potential design issues in the source code of React applications. However, only identifying code smells, even with the aid of an automated tool, does not resolve the underlying design issues. In other words, addressing and eliminating these code smells is equally crucial by applying appropriate refactorings [Sharma et al., 2015]. To facilitate this process, in this section, we establish a mapping between prevalent React code smells and the refactorings identified in this chapter, which could be used to remove them. By associating each code smell with a possible "fixing" refactoring, we aim to provide developers with practical guidance

Table 5.7: React smells and the refactorings that eliminate them

Smell	Possible Refactoring
Large Component	Split Component Extract Component Extract Logic to Custom Hook Extract JSX outside render method to component
Duplicated code	Extract logic to a custom hook Extract Component Extract Higher Order Component Extract HTML/JS code to compoment Replace logic with Hook
Poor names	Rename Component Rename Prop Rename State Move Hook
Dead Code	Remove unused props Remove unused state
Feature Envy	Move Method Move Component
Too Many Props	Remove unused props Split Component
Poor performance	Migrate class component to function component Memoize component
Props in initial state	Remove props in initial state
Direct DOM Manipulation	Remove direct DOM manipulation
Force Update	Remove forceUpdatte()
Inheritance Instead of Composition	Use Composition instead of Inheritance
JSX outside the render method	Extract JSX outside render method to component Extract component
Low Cohesion	Extract component
Conditional rendering	Extract conditional in render
No access state in setState()	Replace access state in setState with callbacks
Direct Mutation of state	Replace direct mutation of state with setState()
Dependency smell	Replace third-party component with own component
Prop Drilling	Extract logic to a custom context

on efficiently addressing and resolving these design issues.

Table 5.7 describes the relationship between specific React smells and the corresponding refactorings that potentially eliminate them. Among the various refactorings available, Extract Component is the most versatile and can address design issues such as Large Components, Duplicated Code, JSX outside the render method, and Low Cohesion. Other refactoring operations also eliminate more than one code smell, such as Split Component, Extract Logic to Custom Hook, Extract JSX outside the render method, and Remove unused props. It is also worth noting that a code smell may be removed by different types of refactoring. For instance, we have identified five refactoring operations that can be used to remove duplicated code. Therefore, it is the developer's responsibility to select the most suitable refactoring

approach that aligns with the specific problem at hand.

5.3.2 Generalization to Other Frameworks

Another key discussion is the possibility of generalizing our refactorings to other front-end frameworks, such as Vue. Since these frameworks usually rely on components for structuring and organizing Web UIs, refactorings such as Rename component, Extract HTML/JS code to component, Replace HTML/JS code with third-party components, and Replace third-party component with own component apply to these frameworks. In addition, the components of these frameworks also have props, making it possible to generalize the Rename props and Remove Unused props refactorings. It is also possible to find bad practices that force the update of components or manipulate the DOM directly in other frameworks.

On the other hand, the `setState()` method is part of the React architecture, which presents challenges when attempting to generalize certain refactorings such as Replace access state in setState with callbacks, Remove unused state and Replace direct mutation of state with setState(). The Migratet Class component to function component and Extract Logic to a Custom Hook refactorings became common after React hooks' release. On the other hand, it is worth mentioning that there are also initiatives to define and share logic in Vue.[3]

Furthermore, the Memoize component refactoring can be applied to other frameworks as well. For instance, Vue provides the `v-memo` directive, which enables the memoization technique. By utilizing `v-memo` on a computed property or method, Vue automatically caches the result and recalculates it only when the dependencies change. This feature significantly enhances the performance of reactive components by avoiding unnecessary recomputations.

5.4 Threats to Validity

In this section we discuss threats to the validity of our results [Wohlin, 2012]. The first threat is related to decisions that may affect our empirical results. As is typical in empirical software engineering studies, our dataset might not represent the entire popu-

[3] `https://www.npmjs.com/package/@u3u/vue-hooks`

lation of React-based projects. For instance, we selected only ten open-source projects, which may not be representative of the extensive number of React projects. To mitigate this threat, we employed a selection criterion of choosing the top-10 projects ranked by stars, a common metric for selecting widely popular and relevant GitHub projects [Borges et al., 2016; Borges and Valente, 2018]. Typically, these projects are continuously maintained and have been utilized in our previous studies [Ferreira et al., 2022; Ferreira and Valente, 2023]. However, future studies could also consider closed projects. Additionally, as a complementary selection criterion, future studies might consider the proportion of React-based code in a repository. This criterion could aid in selecting projects that heavily rely on React to build their front-end components.

Another threat concerns identifying and classifying refactoring operations, which can be subjective due to the manual nature of this step. To mitigate this threat, our analysis involved two stages. Firstly, two researchers independently reviewed each commit message to identify only those commits that unambiguously represented instances of refactoring. In cases of disagreement, the experts engaged in discussions to reach a consensus. Consequently, only commits with clear and explicit descriptions of their refactoring activities were selected for further analysis. However, with this manual approach we may have excluded commits with relevant but non-documented refactorings.

In the second stage, our study involved substantial manual validation and analysis to categorize the refactorings, which can be subjective. To address these threats, the author and a second researcher thoroughly examined the commit diffs carefully analyzing the changes made to the codebase. They also examined associated issues (if any), pull-request descriptions, and comments in the code to gain a better understanding of the contextual changes. The agreement score between them was consistently high for all identification and classification results, as discussed in Section 5.1. In instances of disagreement, the experts discussed the commit(s) to achieve a consensus. They marked the refactorings and their rationale only when they had a high level of confidence in their identification.

Another threat relates to larger commits, which may encompass multiple activities, making it challenging to categorize the tasks accurately. To mitigate this, we focused on commit messages with clear and explicit descriptions of refactoring activity, which were selected during the initial stage of our analysis. Consequently, it is possible that developers performed additional refactorings within the same commit but did not explicitly mention them in the log message, potentially resulting in missed refactorings. Nonetheless, our study still involved manual identification and classification of a representative sample of 320 commits.

A final threat pertains to the heuristics employed to determine whether refactorings were related to React code. Particularly, there could be cases where a commit includes changes both in the front-end and in the back-end, but the refactoring occurs only in the latter. During our manual analysis, we marked such commits as false positives to address

this concern.

5.5 Final Remarks

Refactoring is a well-known technique to improve software quality. However, most studies on refactoring focused on mainstream programming languages, such as Java and JavaScript. This chapter proposed a catalog of common refactorings in React applications. By manually inspecting 320 refactoring instances performed in front-end files, we identified 25 distinct refactoring operations specific to React code, 17 adaptations of traditional refactorings, six specific to JavaScript and CSS code, and 22 traditional refactorings.

Our study offers a range of refactoring options to address these specific design problems in React applications. By employing the appropriate refactoring techniques, developers can improve the quality, maintainability, and overall design of their React-based Web applications. In future work, we plan to validate our findings with professional developers and consider other frameworks, such as Vue and Angular. We also plan to provide tool support to detect the refactorings identified in this chapter. For example, we plan to extend tools such as RefDiff [Silva and Valente, 2017; Silva et al., 2020], or Refactoring Miner [Tsantalis et al., 2013] to detect React-based refactorings. Finally, we also aim to expand our catalog by including composite refactorings, which are sequences of individual refactorings performed on a specific program element (e.g., component decomposition)[Brito et al., 2023].

Chapter 6

Conclusion

This chapter describes the three major works conducted throughout this thesis (Section 6.1). Next, in Section 6.2 we list the main contributions of these efforts. Finally, Section 6.3 outlines future work that we find interesting for follow-up research.

6.1 Thesis Recapitulation

JavaScript-based front-end frameworks, such as REACT and VUE, are key tools for implementing modern Web applications. However, limited research has focused on these frameworks' design and maintenance challenges. In this thesis, we presented a set of three work units where we examined common problems encountered when designing Web systems using JavaScript-based front-end frameworks.

Firstly, we provided an overview of modern JavaScript front-end frameworks in Chapter 2, emphasizing two popular frameworks: REACT and VUE. We also described the state-of-the-art concerning studies related to the design and maintenance issues that occur when implementing Web systems.

In Chapter 3, we presented our initial exploratory study, which involved surveying the key factors that motivate developers to adopt JavaScript front-end frameworks. We also investigated whether they have plans to migrate from one framework to another in the near future. Since the JavaScript ecosystem is highly dynamic, with new frameworks emerging but also being abandoned very rapidly, this information holds practical value for software evolution and practitioners.

In Chapter 4, **we propose a list of code smells for React-based JavaScript applications**. We identified these smells by conducting a grey literature review and interviewing six professional REACT developers. Additionally, we implemented a tool called REACTSNIFFER to detect these smells. By utilizing this tool, **we discovered the most prevalent code smells in React-based Web systems**. Ultimately, we conducted a historical analysis to check how often developers remove the proposed smells.

Finally, in Chapter 5, we reported the results of an empirical study on refactoring operations that developers perform when maintaining and evolving React-based Web applications. Through manual inspection of 320 refactoring commits in open-source projects, we proposed a catalog of refactorings that can be applied to eliminate the smells presented in Chapter 4 and consequently improve the source code quality of front-end components. Additionally we reportetd how often these refactorings occur in a representative sample of open-source projects, and whether they are indeed frontend-specific program transformations or whether they are variations of traditional refactorings.

6.2 Contributions

In the context of the research conducted in this thesis, we highlight the following contributions:

1. We revealed a list of nine key factors developers consider when selecting contemporary JavaScript front-end frameworks. This list is useful to JavaScript developers that plan to adopt a frontend framework in their projects. It can also be used by framework developers, helping them better position their projects in a competitive software market. We also provide a public dataset with popular projects that are clients of front-end frameworks. Therefore, future research on JavaScript front-end frameworks can use this dataset.

2. We proposed 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 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.

3. We implemented a 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.[1]

4. We used 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

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

(10 systems, in total). To complement, we also showed how the removal rate varies among the smells in our catalog (since it is not important to show the number of occurrences of the proposed smells, but also how frequently they are removed). We also provide a public dataset with these smells for future research.

5. Through a careful manual analysis of ten open-source projects, we proposed a catalog comprising 69 refactorings employed by developers when maintaining and evolving React-based applications. We documented 25 new refactorings specific to React and 17 adaptations of traditional refactorings tailored to the React context, therefore acknowledging the distinctive nature of React applications. Additionally, we offer access to the dataset encompassing these refactorings, thus facilitating further studies and analysis.[2]

6. Lastly, we provided a relationship between specific React smells and the corresponding refactorings that eliminate them. By associating each React smell with a "fixing" refactoring, we provided developers with practical guidance on efficiently addressing and resolving these design issues.

6.3 Future Work

During the works conducted in this thesis, we identified some unexplored questions that can result in relevant studies. We detailed these future works in the following sections.

6.3.1 Design Pattern Catalog

A Design Pattern is a typical solution to a recurring design problem. However, despite the popularity of JavaScript front-end frameworks, we still lack a design pattern catalog documenting repeatable solutions to common problems when designing Web systems using these frameworks. Indeed, the classical book on design patterns [Gamma et al., 1994] was published before JavaScript became popular. Therefore, as future work, a study can be conducted to explore front-end frameworks' internal architecture, aiming to document the key design and architectural patterns used by them. We envision that

[2] `https://doi.org/10.5281/zenodo.8044249`

this study can help practitioners gain an in-depth understanding of the internals of such frameworks and, therefore, better assess the trade-offs involved in their adoption.

6.3.2 Code Smells: Tool Improvements and New Future Works

In Chapter 4, we proposed a list of code smells for React-based JavaScript applications. We also implemented a prototype tool, called ReactSniffer, to detect the proposed smells. However, ReactSniffer does not support all proposed smells. Specifically, it does not detect Prop Drilling and Duplicated Component. Thus, a future work can extend our current implementation to handle all identified smells. Chapter 4 also reports the results of a field study conducted to investigate whether the proposed React smells are common in open-source systems. However, we agree that is important 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. Furthermore, our catalog paves the way for future research avenues, including investigating the impact of refactoring these smells, exploring the concurrent presence of these smells, and enhancing metrics for identifying the code smells. Lastly, we provide a public dataset of code smells found in ten GitHub projects that use React. This dataset can also be used for future research endeavors like conducting surveys with developers regarding the identified smells.

6.3.3 Refactorings: Tool Improvements and New Future Works

In Chapter 5, we proposed a catalog of common refactorings in React applications. Our study offers a range of refactoring options to address specific design problems in React applications. By employing the appropriate refactoring techniques, developers can improve the quality, maintainability, and overall design of their React-based Web applications. In future work, we plan to validate our findings with professional developers and consider other frameworks, such as Vue and Angular. We also plan to provide tool support to detect the refactorings identified in Chapter 5. For example, we plan to extend tools such as RefDiff [Silva and Valente, 2017; Silva et al., 2020], or Refactoring Miner [Tsantalis et al., 2013] to detect React-based refactorings. We also provide a public dataset with refactorings performed by developers on React applications. This

dataset can be used for other studies in the future, for example, for surveying developers about the motivations for performing specific refactorings and on how these refactorings affect quality attributes. Finally, we also aim to expand our catalog by including composite refactorings, which are sequences of individual refactorings performed on a specific program element (e.g., component decomposition)[Brito et al., 2023].

Bibliography

Alizadeh, V., Kessentini, M., Mkaouer, M. W., Cinnéide, M. Ó., Ouni, A., and Cai, Y. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, 46(9):932–961, 2018.

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

Aniche, M., Bavota, G., Treude, C., Gerosa, M. A., and van Deursen, A. Code smells for model-view-controller architectures. *Empirical Software Engineering*, 23(4):2121–2157, 2018.

Araújo, C. P. and Filho, A. M. Evolution of web systems architectures: A roadmap. *Special Topics in Multimedia, IoT and Web Technologies*, page 1, 2020.

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

Baltes, S. and Diehl, S. Worse than spam: Issues in sampling software developers. In *10th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2016.

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

Beck, K. *Extreme programming explained: embrace change*. addison-wesley professional, 1 edition, 2000.

Boldyreff, C. and Kewish, R. Reverse engineering to achieve maintainable WWW sites. In *Eighth Working Conference on Reverse Engineering*, pages 249–257. IEEE, 2001.

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

Borges, H. S. and Valente, M. T. What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146 (1):112–129, 2018.

Brito, A., Hora, A., and Valente, M. T. Towards a catalog of composite refactorings. *Journal of Software: Evolution and Process*, 1:1–22, 2023.

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

Coelho, J., Valente, M. T., Silva, L. L., and Hora, A. Why we engage in FLOSS: Answers from core developers. In *11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 14–21, 2018.

Cruzes, D. S. and Dybå, T. Research synthesis in software engineering: A tertiary study. *Information and Software Technology*, 53(5):440–455, 2011.

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

Davis, F. D. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):319–340, 1989.

Diniz, J. P., Cruz, D., Ferreira, F., Tavares, C., and Figueiredo, E. GitHub Label Embeddings. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 249–253, 2020.

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

Fard, A. M. and Mesbah, A. Jsnose: Detecting JavaScript code smells. In *13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125, 2013.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. A review-based comparative study of bad smell detection tools. In *20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–12, 2016.

Ferreira, F. and Valente, M. T. Detecting Code Smells in React-based Web Apps. *Information and Software Technology*, 155:1–35, 2023.

Ferreira, F., Silva, L. L., and Valente, M. T. Turnover in Open-Source Projects: The Case of Core Developers. In *34th Brazilian Symposium on Software Engineering (SBES)*, pages 447–456, 2020.

Ferreira, F., Silva, L. L., and Valente, M. T. Software engineering meets deep learning: a mapping study. In *36th Annual ACM Symposium on Applied Computing (SAC)*, pages 1542–1549, 2021.

Ferreira, F., Borges, H. S., and Valente, M. T. On the (Un-) Adoption of JavaScript Front-end frameworks . *Software: Practice and Experience*, 52(4):947–966, 2022.

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

Fowler, M. and Beck, K. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

Fowler, M. and Beck, K. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, second edition, 2018.

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

Garousi, V., Felderer, M., and Mäntylä, M. V. 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)*, pages 1–6, 2016.

Garousi, V., Felderer, M., and Mäntylä, M. V. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106(1):101–121, 2019.

Geneves, P., Layaïda, N., and Quint, V. On the analysis of cascading style sheets. In *21st International Conference on World Wide Web (WWW)*, pages 809–818, 2012.

Gizas, A., Christodoulou, S., and Papatheodorou, T. Comparative evaluation of javascript frameworks. In *21st International Conference on World Wide Web*, pages 513–514, 2012.

Graziotin, D. and Abrahamsson, P. Making Sense Out of a Jungle of JavaScript Frameworks. In *Product-Focused Software Process Improvement*, pages 334–337, 2013.

Harold, E. R. *Refactoring html: improving the design of existing Web applications.* Addison-Wesley Professional, 1 edition, 2012.

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

Hora, A. and Robbes, R. Characteristics of method extractions in java: A large scale empirical study. *Empirical Software Engineering*, 25:1798–1833, 2020.

Joblin, M., Apel, S., Hunsen, C., and Mauerer, W. Classifying developers into core and peripheral: An empirical study on count and network metrics. In *39th International Conference on Software Engineering (ICSE)*, pages 164–174, 2017.

Johannes, D., Khomh, F., and Antoniol, G. A large-scale empirical study of code smells in javascript projects. *Software Quality Journal*, 27(3):1271–1314, 2019.

Kamei, F., Wiese, I., Lima, C., Polato, I., Nepomuceno, V., Ferreira, W., Ribeiro, M., Pena, C., Cartaxo, B., Pinto, G., et al. Grey literature in software engineering: A critical review. *Information and Software Technology*, 1(1):1–26, 2021.

Kim, M., Zimmermann, T., and Nagappan, N. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7): 633–649, 2014.

Krasner, G. E. and Pope, S. T. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80, ch. 31 (3). *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

Ksontini, E., Kessentini, M., do N. Ferreira, T., and Hassan, F. Refactorings and technical debt in docker projects: An empirical study. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 781–791, 2021.

Lacerda, G., Petrillo, F., Pimenta, M., and Guéhéneuc, Y. G. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610, 2020.

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

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

Lucca, G. A. D., Penta, M. D., and Fasolino, A. R. An approach to identify duplicated web pages. In *26th Annual International Computer Software and Applications*, pages 481–486. IEEE, 2002.

Lucca, G. A. D., Penta, M. D., Fasolino, A. R., and Granato, P. Clone analysis in the web era: An approach to identify cloned web pages. In *Seventh Workshop on Empirical Studies of Software Maintenance*, volume 107, 2001.

Lucia, A. D., Francese, R., Scanniello, G., and Tortora, G. Understanding cloned patterns in web applications. In *13th International Workshop on Program Comprehension (IWPC)*, pages 333–336. IEEE, 2005.

Mariano, C. Benchmarking JavaScript Frameworks. Master's thesis, Technological University Dublin, 2017.

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

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

McCabe, T. J. A complexity measure. *IEEE Transactions on software Engineering*, 1(4): 308–320, 1976.

Mesbah, A. and Mirshokraie, S. Automated analysis of css rules to support style maintenance. In *34th International Conference on Software Engineering (ICSE)*, pages 408–418, 2012.

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

Moha, N., Guéhéneuc, Y.-G., Duchien, L., and Le Meur, A.-F. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2009.

Montandon, J. E., Politowski, C., Silva, L. L., Valente, M. T., Petrillo, F., and Guéhéneuc, Y.-G. What skills do IT companies look for in new developers? a study with Stack Overflow Jobs. *Information and Software Technology*, 1:1–6, 2020.

Mori, A., Vale, G., Viggiato, M., Oliveira, J., Figueiredo, E., Cirilo, E., Jamshidi, P., and Kastner, C. Evaluating domain-specific metric thresholds: an empirical study. In *Proceedings of the 2018 International Conference on Technical Debt*, pages 41–50, 2018.

Murphy-Hill, E., Parnin, C., and Black, A. P. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2011.

Nakajima, N., Matsumoto, S., and Kusumoto, S. Jact: A playground tool for comparison of JavaScript frameworks. In *26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 474–481, 2019.

Nederlof, A., Mesbah, A., and van Deursen, A. Software engineering for the web: the state of the practice. In *36th International Conference on Software Engineering (ICSE)*, pages 4–13, 2014.

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

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

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

Paiva, T., Damasceno, A., Figueiredo, E., and Sant'Anna, C. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 5(1):1–28, 2017.

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

Pano, A., Graziotin, D., and Abrahamsson, P. Factors and actors leading to the adoption of a JavaScript framework. *Empirical Software Engineering*, 23(6):3503–3534, 2018.

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

Peruma, A., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. An exploratory study on the refactoring of unit test files in Android applications. In *42nd International Conference on Software Engineering Workshops*, pages 350–357, 2020.

Polančič, G., Heričko, M., and Pavlič, L. Developers' perceptions of object-oriented frameworks – An investigation into the impact of technological and individual characteristics. *Computers in Human Behavior*, 27(2):730–740, 2011.

Rajapakse, D. C. and Jarzabek, S. An investigation of cloning in web applications. In *International Conference on Web Engineering*, pages 252–262. Springer, 2005.

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

Robles, G., Gonzalez-Barahona, J. M., and Herraiz, I. Evolution of the core team of developers in libre software projects. In *6th International Working Conference on Mining Software Repositories (MSR)*, pages 167–170, 2009.

Saboury, A., Musavi, P., Khomh, F., and Antoniol, G. An empirical study of code smells in javascript projects. In *24th international conference on software analysis, evolution and reengineering (SANER)*, pages 294–305. IEEE, 2017.

Sharma, T. and Spinellis, D. A survey on software smells. *Journal of Systems and Software*, 138:158–173, 2018.

Sharma, T., Suryanarayana, G., and Samarthyam, G. Challenges to and solutions for refactoring adoption: An industrial perspective. *IEEE Software*, 32(6):44–51, 2015.

Silva, D. and Valente, M. T. RefDiff: Detecting refactorings in version histories. In *14th International Conference on Mining Software Repositories (MSR)*, pages 1–11, 2017.

Silva, D., Tsantalis, N., and Valente, M. T. Why we refactor? confessions of GitHub contributors. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 858–870, 2016.

Silva, D., da Silva, J. P., Santos, G., Terra, R., and Valente, M. T. RefDiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, 1(1): 1–17, 2020.

Silva, L. H., Valente, M. T., Bergel, A., Anquetil, N., and Etien, A. Identifying classes in legacy JavaScript code. *Journal of Software: Evolution and Process*, 29(8):1–20, 2017.

Sobrinho, E. V. P., Lucia, A. D., and de Almeida Maia, M. A systematic literature review on bad smells–5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering*, 47(1):17–66, 2018.

Synytskyy, N., Cordy, J. R., and Dean, T. Resolution of static clones in dynamic web pages. In *Fifth IEEE International Workshop on Web Site Evolution, 2003. Theme: Architecture. Proceedings.*, pages 49–56. IEEE, 2003.

Tang, Y., Khatchadourian, R., Bagherzadeh, M., Singh, R., Stewart, A., and Raja, A. An empirical study of refactorings and technical debt in machine learning systems. In *43rd International Conference on Software Engineering (ICSE)*, pages 238–250. IEEE, 2021.

Tsantalis, N. and Chatzigeorgiou, A. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

Tsantalis, N., Guana, V., Stroulia, E., and Hindle, A. A multidimensional empirical study on refactoring activity. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 132–146, 2013.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. Accurate and efficient refactoring detection in commit history. In *40th International Conference on Software Engineering (ICSE)*, pages 483–494, 2018.

Vale, G., Fernandes, E., and Figueiredo, E. On the proposal and evaluation of a benchmark-based threshold derivation method. *Software Quality Journal*, 27(1):275–306, 2019.

Vegi, L. and Valente, M. T. Towards a catalog of refactorings for Elixir. In *39th International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–5, 2023.

Venkatesh, V., Morris, M. G., Davis, G. B., and Davis, F. D. User acceptance of information technology: Toward a unified view. *MIS Quarterly*, 27(3):425–478, 2003.

Wittern, E., Suter, P., and Rajagopalan, S. A look at the dynamics of the JavaScript package ecosystem. In *13th International Conference on Mining Software Repositories (MSR)*, pages 351–361, 2016.

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

Xavier, L., Ferreira, F., Brito, R., and Valente, M. T. Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems. In *17th International Conference on Mining Software Repositories (MSR)*, pages 137–146, 2020.

Xavier, L., Montandon, J. E., Ferreira, F., Brito, R., and Valente, M. T. On the Documentation of Self-Admitted Technical Debt in Issues . *Empirical Software Engineering*, 27(7):1–34, 2022.

Zerouali, A., Mens, T., Robles, G., and Gonzalez-Barahona, J. M. On the diversity of software package popularity metrics: An empirical study of npm. In *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 589–593, 2019.

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