The Journal of Systems and Software 215 (2024) 112105

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Refactoring react-based Web apps☆

Fabio Ferreira [a,b,*], Hudson Silva Borges [c], Marco Tulio Valente [a]

[a] Department of Computer Science, UFMG, Brazil
[b] Center of Informatics, IF Sudeste MG - Campus Barbacena, Brazil
[c] Department of Computer Science, UFMS, Brazil

ARTICLE INFO

Keywords:
Refactoring
React
JavaScript
Software design

ABSTRACT

Refactoring is a well-known technique to improve software quality. However, there are relevant domains where refactoring has not been studied in-depth before, such as JavaScript front-end frameworks. To fill this gap, we empirically study refactorings 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 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. The catalog of refactorings proposed in this article might support practitioners when improving the maintainability of React applications.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

Refactoring is a well-known technique to improve software design and an indispensable practice in modern software development. On the one hand, most studies on refactoring focused on mainstream programming languages, such as Java (Fowler, 1999) and JavaScript (Fowler, 2018). On the other hand, there are also 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, 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, components implemented using JavaScript frameworks, such as React[1] and Vue.[2] Essentially, such frameworks provide abstractions – called components – for structuring and organizing the codebase of modern and responsive Web UIs.

We claim it is important to study refactorings in the domain of front-ends for two major reasons. First, modern front-ends can have hundreds of components and hundreds of thousands of lines of code. Thus, it is natural that suboptimal design decisions will eventually occur in their code (Ferreira and Valente, 2023) and consequently refactoring might be needed to keep this code easy to understand, change, and evolve. Second, it is reasonable to assume that most traditional refactoring operations – as defined for example in Fowler's book (Fowler, 1999) – apply to frontend-based code. However, it is possible that new and specific refactorings are also useful in this domain. Therefore, identifying, documenting, and discussing frontend-specific refactorings is important to promote best software quality practices among frontend developers.

In this article we describe an empirical study on refactorings in React applications. We focus on React because it is currently the most popular JavaScript front-end framework (Hora, 2021).[3] 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.

To answer these questions, we focused on the top-10 React projects by stars in GitHub and manually inspected 320 commits performed in their front-end files. These commits explicitly include keywords related to refactorings in their log messages. As a result, we identified 69 distinct refactoring operations categorized into four categories. Among these, 22 refactorings are traditional transformations (*i.e.,* described in Fowler's book Fowler, 1999), six are specific to JavaScript and CSS

☆ Editor: Gabriele Bavota.

* Corresponding author at: Center of Informatics, IF Sudeste MG - Campus Barbacena, Brazil.
E-mail address: fabio.ferreira@ifsudestemg.edu.br (F. Ferreira).

[1] https://react.dev.
[2] https://vuejs.org.
[3] https://survey.stackoverflow.co/2023/#most-popular-technologies-webframe.

https://doi.org/10.1016/j.jss.2024.112105
Received 20 June 2023; Received in revised form 15 February 2024; Accepted 20 May 2024
Available online 22 May 2024

code, 25 are specific to React code, and 17 are React-adapted refactorings (*i.e.,* although related to the React context, they are adaptations of traditional refactorings).

Therefore, our contributions can be summarized as follows:

- Firstly, through a careful manual analysis of ten open-source projects, we present a catalog comprising 69 refactorings employed by developers when maintaining and evolving React-based applications. Additionally, we offer access to the dataset encompassing these refactorings, thus facilitating further studies and analysis.
- Secondly, we document 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.
- Lastly, our catalog provides a foundation for studies on recommendations, best practices, and anti-patterns to guide the evolution of React projects. For each refactoring operation, we provide a clear and concise description and code examples, where applicable, elucidating the benefits and circumstances under which they should be applied.

The remainder of this paper is organized as follows. In Section 2, we present background information on React. In Section 3, we present the methodology we used to define our catalog of refactorings. In Section 4, we present this catalog. In Section 5, we discuss and put our findings and insights in perspective. In Section 6, we detail threats to validity. Finally, we present related work in Section 7 and conclude in Section 8.

2. Background

React is a powerful JavaScript library developed and maintained by Facebook for building the user interface (UIs) of web applications. With React, developers can adopt a declarative and efficient approach to creating UI elements that can be dynamically updated based on changes in data or user interactions.

React utilizes a *virtual DOM* (Document Object Model) to render and update UI elements efficiently. It allows developers to describe the structure and behavior of UI elements using JSX (JavaScript XML), a domain-specific language combining JavaScript and XML syntax. Thus, instead of separating markup and logic into separate files, JSX allows developers to adopt a modular approach through reusable units called *components* (Ferreira et al., 2022). These components encapsulate both the logic and markup, resulting in code that is easier to maintain.

A component represents a reusable and self-contained UI piece, which can be composed and nested to build complex user interfaces. React components can accept parameters called *props* (short for *properties*) and return a React element via the `render` method, which determines 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 recommends 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.

We extracted the following example from a previous study on React code smells (Ferreira and Valente, 2023) to demonstrate the implementation of the same component using both *class* component (see Listing 1) and *function* component (see Listing 2). In both cases, the component accepts a single object argument (`name`) via `props` and returns a React element that displays a welcoming message.

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

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

Listing 2: Example of Function Component.

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

Listing 1: Example of Class Component.

React elements are immutable. Once created, their data cannot be changed. Furthermore, to render a React element, it is necessary to establish a root within a browser's Document Object Model (DOM) node where React components can be displayed. Subsequently, the React element is passed to the `root.render()` method. For instance, consider the HTML page presented in Listing 3, wherein the HTML element with the identifier `root` (line 3) is employed to render the previous React element responsible for exhibiting a welcoming message.

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

Listing 3: Example of HTML page used to render a React component.

Listing 4 shows how to render a React element using the `Welcome` component. Initially, the DOM element must be passed to the `ReactDOM.createRoot()` function (lines 1–3). Subsequently, the `Welcome` component (line 4) is instantiated (line 4). Finally, the React element is provided as an argument to `root.render()`.

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

Listing 4: Rendering a React element

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 attributes 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 only renders what it receives via *props* or always renders the same content. On the other hand, a stateful component in addition to taking input data (via `this.props`) can maintain internal data (via `this.state`). Furthermore, when a component's state changes, the `render` method is automatically re-invoked 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. In the domain of JavaScript front-end frameworks,

Table 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	3476	4319	80,271	1917	320

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.

3. Study design

Our goal is to study React-specific refactorings that developers perform when maintaining and evolving Web apps. For that, we used a dataset of a previous study on the (un-)adoption of JavaScript front-end frameworks (Ferreira et al., 2022). From this dataset, we selected the top-10 React-based projects by stars after manually discarding non-software projects and projects that use React only in examples, tutorials, documentation, and tests. For each selected project, Table 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 keywords refactor*, in their log messages. However, we also selected commits containing the keywords reengineer*, restructur*, and reorgani*. We decided to follow this strategy for three reasons. First, we still do not have tools for mining React-based refactorings in version histories (as the ones we have for Java and other languages Silva et al., 2020; Silva and Valente, 2017; Tsantalis et al., 2018). Indeed, we view our catalog of React-specific and React-adapted refactorings as a first step in the direction of having these tools in the future. Second, the same approach has been recently used by works that mine domain-specific refactorings, such as refactorings performed in machine learning systems (Tang et al., 2021) and Docker configuration files (Ksontini et al., 2021). Third, to achieve our key goal (a catalog of React refactorings) we do not need to mine all refactorings performed in our dataset. Instead, we can rely on a sample of representative refactoring operations, which could allow us to propose a first catalog of refactorings for React. The "Ref." column in Table 1 shows the number of commits containing at least one of the mentioned keywords in their messages (1917 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 (Serdar et al., 2021). Using this methodology, we selected 320 commits out of a total of 1917 to examine manually, striking a balance between minimizing costs and time while ensuring statistical confidence. The "Analyzed" column in Table 1 shows the number of commits selected by project.

Table 2
Example of refactoring operations labeled with different names (the final selected version is underlined).

First author classification	Second author classification
Merge Components	Combine components into one
Convert JS code in TS	Migrate to Typescript
Replace class component with 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

Due to the large and complex nature of many commits, the first author of this article 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. Fig. 1 shows a second example of an unclear log message related to a discarded commit.

After the first author's initial examination, 65 commits were discarded. To ensure accuracy, the third author of this article 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 these authors, 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 Fig. 2. Consequently, the final classification resulted in 261 commits whose descriptions clearly indicate refactorings.

In the following analysis stage, the first and second authors undertook a detailed and independent examination of the 261 selected commits using thematic analysis. 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 authors 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 (Diniz et al., 2020). Then, each author generated initial labels for every commit, considering that a single commit could encompass multiple refactoring operations. Moreover, at any time, each classifier could suggest a new label (i.e., we did not use a closed set of labels).

After completing the analysis, the authors 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 2. In this table, we underline the final name choice.

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 authors held differing opinions. The disagreements were often attributed to instances where one author failed to identify a refactoring activity that the other author

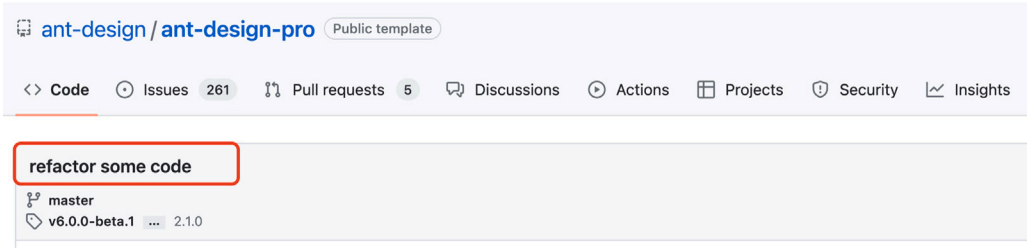

Fig. 1. Example of an unclear commit.

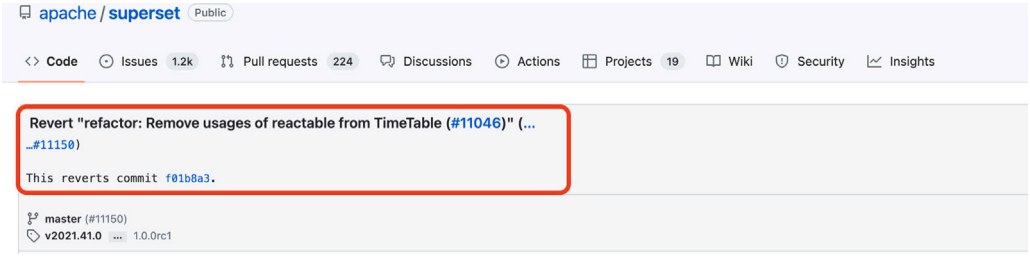

Fig. 2. Example of a discarded commit because it is revoking a previous refactoring operation.

had recognized. Another case of disagreement emerged when one author identified an activity as a refactoring while the other author did not recognize it as such. For instance, one author attributed the labels "Add parameter" and "Change parameter" to certain commits. However, the other author 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 first and second authors 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, Fig. 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 authors classified the refactoring instances into four major categories:

- React-specific refactorings (163 instances, 28.8%), which are novel refactorings that only occur in front-end code.
- React-adapted refactorings (185 instances, 32.7%), 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 Fig. 4. In the subsequent section, we provide a comprehensive overview of the proposed catalog, delving into the specifics of more frequent refactoring operations.

4. A catalog of refactorings for react-based web apps

In the following sections, we discuss the most frequent React-specific (Table 3) and React-adapted (Table 4) refactorings. In addition, we briefly comment on the JavaScript-specific (Table 5) and on the traditional (Table 6) refactorings.

In order to prevent this article from becoming too long, our presentation of the refactorings is concise. However, a more detailed presentation of the main refactorings (with at least 5 occurrences) can be found in a supplementary repository: https://github.com/fabiosferreira/refactoring-react-web-apps.

4.1. 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 3, we have identified a total of 163 instances involving 25 unique React refactoring operations. These operations are exclusive to front-end code and have been recurrently observed across multiple projects, indicating their relevance and practical applicability in React development. As can be seen in Table 3, we also classified the React-specific refactorings into five categories, as follows:

- **Hook Modularization:** refactorings that improve the structure or logic of hooks, such as Replace Logic to Hook or Split Hook.
- **Component Modularization:** refactorings that improve the structure and organization of components, such as Extract High-Order Component or Migrate class component to function component.
- **State Improvements:** refactorings that improve the organization or structure of state handling code, such as Remove Props in Initial State and Replace Direct Mutation of State with SetState().
- **Rendering Improvements:** refactorings that improve the organization or structure of rendering logic, such as Extract Conditional in Render or Remove ForceUpdate().
- **Other:** Other refactorings related to a variety of concerns, such as data manipulation, routing, navigation, and event handling.

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

4.1.1. *Extract logic to a custom hook*

By default, React provides hooks to use state and other React features without writing a class. For example, the `useState()` hook allows tracking state in function components. However, the logic that deals with state might also become duplicated in components. For example, in a chat application, more than one component may store data

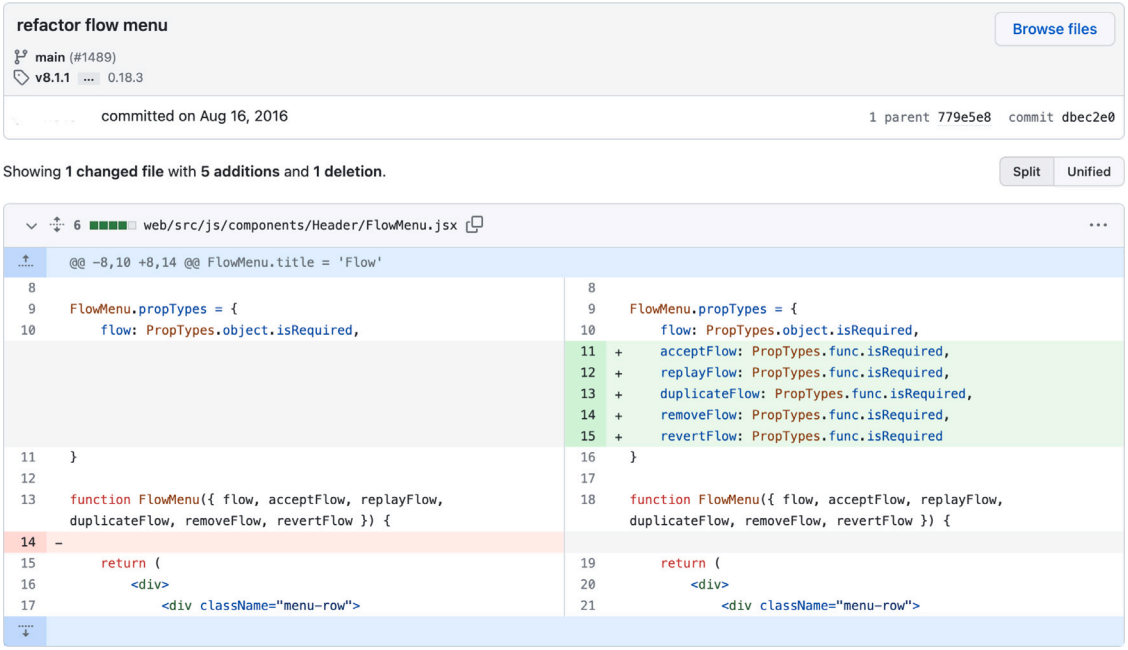

Fig. 3. Example of a false positive commit that does not preserve behavior.

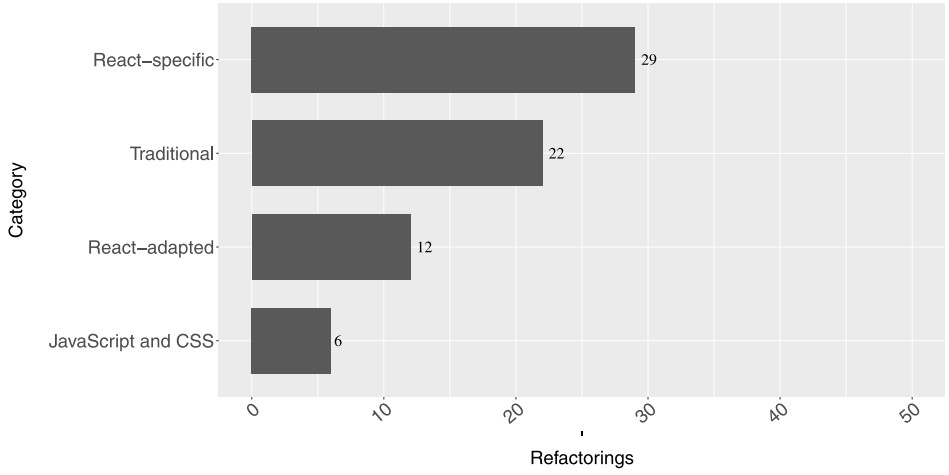

Fig. 4. Number of refactoring operations by category.

on 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 eliminating 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 across multiple components is implemented in a single function.

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 Fig. 5. We found 47 occurrences of this refactoring distributed over eight projects.

4.1.2. *Migrate class component to function component*

React supports class and function components. A class component is an ES6 class with local state, lifecycle control methods (*e.g.,* `componentDidMount()` and `componentDidUpdate()`) 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 distributed over seven projects. Fig. 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)

Table 3
React-specific refactorings.

Refactoring	Category	Occur.	Projects
Extract logic to a custom hook	Hook Modularization	47	8
Migrate class component to function component	Component Modularization	33	7
Migrate Angular to React Component	Component Modularization	7	2
Replace third-party component with own component	Component Modularization	5	2
Extract conditional in render	Rendering Improvements	4	2
Remove props in initial state	State Improvements	4	4
Migrated to styled component	Component Modularization	4	2
Memoize component	Rendering Improvements	4	3
Extract higher-order component (HOC)	Component Modularization	3	2
Remove direct DOM manipulation	Rendering Improvements	3	2
Replace access state in setState with callbacks	State Improvements	3	2
Replace direct mutation of state with setState()	State Improvements	2	1
Replace logic to hook	Hook Modularization	2	1
Remove forceUpdatte()	Rendering Improvements	2	1
Extract logic to a custom context	Other	1	1
Split hook	Hook Modularization	1	1
Generalize hook	Hook Modularization	1	1
Replace callback bind in constructor with bind in render	Other	1	1
Replace HTML/JS code with third-party components	Component Modularization	1	1
Migrate function component to class component	Component Modularization	1	1
Migrate React.FC to functional component syntax	Component Modularization	1	1
Move reducer	State Improvements	1	1
Replace id with Routers	Other	1	1
Generalize interface to accept class and functional component types	Component Modularization	1	1
Replace dynamic keys with stable ids	Other	1	1
Total		134	–

```
 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

Fig. 5. Refactoring that extracts a logic to a custom hook.

removes the constructor and replaces the state with a `useState` hook, (5) replaces the `componentDidMount()` lifecycle method with a `useEffect` hook.

4.1.3. 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.

4.1.4. 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 distributed over two projects.

4.1.5. 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 distributed over seven projects

```
- class CreateUserDialog extends React.Component {
-   static propTypes = {
-     dialog: DialogPropType.isRequired,
-     onCreate: PropTypes.func.isRequired,
-   };

-   constructor(props) {
-     super(props);
-     this.state = { savingUser: false, errorMessage: null };
-     this.form = React.createRef();
-   }
-
-   componentDidMount() {
      recordEvent("view", "page", "users/new");
-   }

-   createUser = () => {
```

```
+ function CreateUserDialog({ dialog }) {
+   const [error, setError] = useState(null);
+   const formRef = useRef();

+   useEffect(() => {
      recordEvent("view", "page", "users/new");
+   }, []);

+   const createUser = useCallback(() => {
```

Fig. 6. Refactoring `CreateUserDialog` class component to function component.

that simplify such conditionals by extracting the JSX code to other components or helper methods.

4.1.6. 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, distributed over four projects, that eliminate the initialization of state with props.

4.1.7. 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.[5] 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 distributed over two projects.

4.1.8. 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 distributed over three projects.

4.1.9. 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 as Redux's connect.[6] For example, the following code uses 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. Fig. 7 shows a commit message indicanting the extract of an HOC called LiveItemsList, which wraps common logic for the DashboardList, QueriesList, and UsersList components.

We found three occurrences of this refactoring distributed over two projects.

4.1.10. 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 distributed over two projects.

4.1.11. 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 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:

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

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

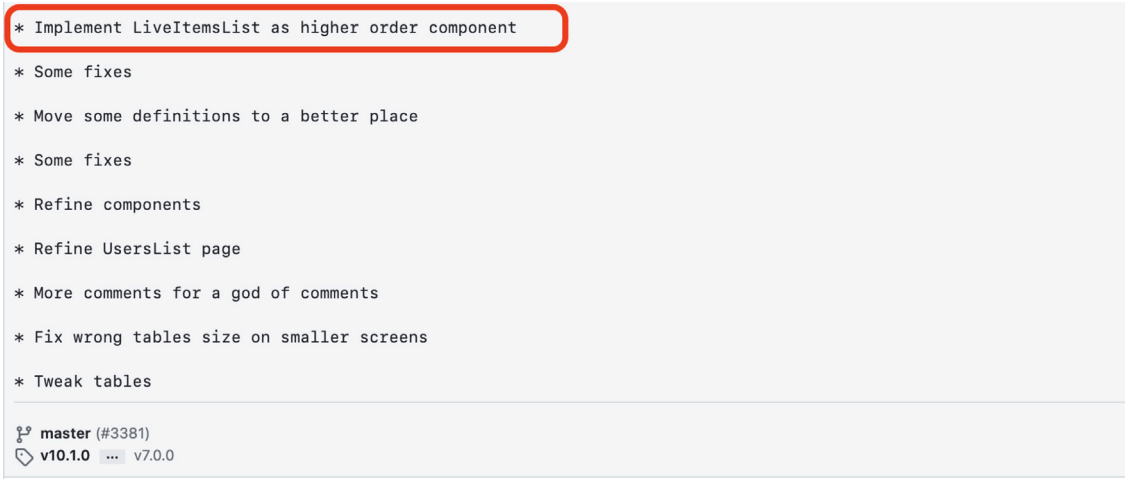

Fig. 7. Commit message indicanting an extract HOC refactoring.

```
1 this.setState({counter: this.state.counter +
      1}) // 2
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:

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

We found three occurrences of this refactoring distributed over two projects.

4.1.12. 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, in a single project, two occurrences of direct mutation of the state being replaced with mutation using the `setState()` method.

4.1.13. Replace 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 in a single project.

4.1.14. 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()` in a single project.

Table 4
React-adapted refactorings.

Refactoring	Similar to	Occur.	Projects
Extract component	Extract Class	76	10
Rename component	Rename Class	30	7
Move Component	Move Class	24	7
Remove unused props	Remove unused parameter	24	6
Rename prop	Rename field	12	4
Split component	Extract Class	9	4
Move Hook	Move Method	8	2
Extract HTML/JS code to component	Extract Class	7	4
Extract JSX outside render method to component	Extract Class	6	5
Rename hook	Rename Method	6	3
Combine Components into one	Combine Functions into Class	3	3
Remove unused state	Remove unused field	3	2
Rename state	Rename Variable	2	1
Remove unused useEffect	Dead Code Elimination	1	1
Remove unused hook	Dead Code Elimination	1	1
Change prop type	Change variable type	1	1
Replace value with props	Change Value to Reference	1	1
Total		214	–

4.2. React-adapted refactorings

In our analysis, we found 185 instances of 13 refactoring operations that, although related to the React context, are adaptations of traditional refactorings. For this reason, we call them as React-adapted refactorings. Table 6 shows such refactorings and highlights the traditional refactoring they are similar to.

4.2.1. 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.

4.2.2. 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.

4.2.3. 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.

4.2.4. 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.

4.2.5. 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.

4.2.6. 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.

4.2.7. 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.

4.2.8. 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.

4.2.9. 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.

4.2.10. 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.

4.2.11. 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.

4.2.12. 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.

Table 5
JavaScript and CSS refactorings.

Refactoring	Type	Occur.	Projects
Convert JS code in TS	JS	13	5
Migrate function to arrow function syntax	JS	4	2
Replace promises with useCallBack	JS	1	1
Replace EOL to semi-colon format	JS	1	1
Rename CSS Class	CSS	2	2
Extract Stylesheet	CSS	1	1
Total		22	–

Table 6
Traditional refactorings.

Refactoring	Occur.	Projects
Dead Code Elimination	83	9
Move Function	20	6
Extract Function	16	5
Rename Function	13	3
Consolidate Conditional Expression	11	5
Duplicated Code Elimination	7	4
Rename Method	6	4
Rename Variable	6	3
Extract Method	4	3
Rename Type	4	2
Rename Parameter	4	2
Move File	3	3
Rename File	3	2
Rename Object Fields	2	2
Move Type Definition	2	1
Replace magic literal	2	2
Merge Methods	1	1
Move Method	1	1
Rename Interface	1	1
Encapsulate fields in Object	1	1
Replace custom logic with external lib	1	1
Use Composition instead of Inheritance	1	1
Total	192	–

4.2.13. Rename state

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

4.3. JavaScript and CSS refactorings

As described in Table 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.

4.4. Traditional refactorings

As summarized in Table 6, we also found 192 instances of 22 refactoring operations documented in Fowler's catalog (Fowler, 1999). Dead Code Elimination is the most common one, with 27 occurrences.

5. 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.

5.1. React code smells and refactorings

In a previous study (Ferreira and Valente, 2023), 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

Table 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 component
	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

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 paper, which could be used to remove them. Our ultimate goal is to provide developers with practical guidance on addressing and resolving the design issues associated with React-based smells.

Table 7 describes the relationship between specific React smells and the corresponding refactorings that eliminate them. This mapping was first created by the first author. Essentially, for each code smell proposed in our previous work (Ferreira and Valente, 2023), he checked whether there are refactorings that could remove the smell. After that, the resulting mapping was discussed with the two other authors, who confirmed and agreed with the proposed relationships. Thus, despite the first author being an expert in both sides of the mapping (smells vs refactorings) we acknowledge that the current mapping should be viewed as recommendations. In future work, we plan to collect feedback from developers about these initial recommendations, evaluate their accuracy and possibly complement and improve the current mapping.

As can be checked in Table 7, 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.

5.2. 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.[7]

Furthermore, the Memoize component refactoring can be applied to other frameworks as well. For instance, Vue provides offers 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.

6. 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 experimental results. As is typical in empirical software engineering studies, our dataset might not represent the entire population 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; Silva and Valente, 2018). Typically, these projects are continuously maintained and have been utilized in previous studies (Ferreira et al., 2022; Ferreira and Valente, 2023). However, we also acknowledge that other criteria – including license, documentation, number of issues, and community engagement – are also useful to select high-quality repositories. In other words, we do not claim that only the top-starred GitHub projects follow high software quality standards.

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, the first and the third authors 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 first and second authors thoroughly

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

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 3. 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.

Another 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. Moreover, our initial selection of commits using keywords may have missed commit messages including the name of the refactoring (e.g., *Migrate class component to function component*) as well as messages that do not mention refactorings at all.

Last but not least, we acknowledge that generalizability is also a concern in this kind of study. First, we tackled this concern by selecting a representative sample of ten well-known React projects. After that, we randomly selected a sample of commits of such projects that would allow us to derive results with a confidence level of 95% and a margin of error of 5%. The commits were manually analyzed by two authors and by a third author in case of conflicts. Despite that, we agree that our results may not be fully generalizable and we may have missed refactorings. However, we also think this threat is minimized due to two factors. First, in our sample, we were able to identify a representative number of React-specific (25) and React-adapted refactorings (17). Second, we also identified a number of traditional refactorings (22), which confirm that the selected commits indeed include transformations for improving code and design quality. On the other hand, out of the 42 React-specific or React-adapted refactorings proposed in our catalog, 45.2% were identified in a single project. Therefore, we should not make general claims regarding their widespread usage and popularity.

To better understand the threats to the generalization of our results, we decided to assess a small sample of new commits. Thus, we randomly selected 20 commits that were not used for the construction of the catalog proposed in Section 4. The messages of these commits were read and analyzed by the first author to identify the refactorings described in them. Table 8 shows the refactorings detected in this new sample. As we can see, all of them are already present in the catalog proposed in the study. Therefore, this additional result also helps to minimize the concerns related to the generalization of our results.

Table 8
Refactorings detected in a new sample.

Refactoring	Type	Occur.	Projects
Dead Code Elimination	Traditional	9	4
Extract Component	React-adapted	6	3
Convert JS code in TS	JS	3	1
Migrate Class to Functional Component	React-specific	2	2
Remove unused props	React-adapted	2	2
Move Function	Traditional	2	1
Memoize Component	React-specific	1	1
Rename Type	Traditional	1	1
Rename Component	React-adapted	1	1
Split Component	React-adapted	1	1

7. Related work

Refactoring is a fundamental practice to maintain a healthy code base (Fowler, 1999; Beck, 2000). For this reason, a significant amount of empirical research was conducted to extend our knowledge on 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, 1999) and JavaScript (Fowler, 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 3188 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 HTML (Nederlof et al., 2014), CSS (Mesbah and Mirshokraie, 2012), JavaScript (Fard and Mesbah, 2013), 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.

Other recent works also investigate and document refactorings in specific and relevant domains. For instance, 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 particularly 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 technical debt categories.

8. Conclusion

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 article 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 (which could also help us to identify new problems and challenges they face when refactoring React-based code) and consider other frameworks, such as Vue and Angular. We also plan to provide tool support to detect the refactorings identified in this paper. 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).

Replication package

Our data is publicly available at https://doi.org/10.5281/zenodo.8044249.

CRediT authorship contribution statement

Fabio Ferreira: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Hudson Silva Borges:** Software, Investigation, Formal analysis, Data curation. **Marco Tulio Valente:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

Our research is supported by CNPq, FAPEMIG, and CAPES.

References

Alizadeh, Vahid, Kessentini, Marouane, Mkaouer, Mohamed Wiem, Cinnéide, Mel Ó., Ouni, Ali, Cai, Yuanfang, 2018. An interactive and dynamic search-based approach to software refactoring recommendations. IEEE Trans. Softw. Eng. 46 (9), 932–961.

Aniche, Maurício, Bavota, Gabriele, Treude, Christoph, Gerosa, Marco Aurélio, van Deursen, Arie, 2018. Code smells for model-view-controller architectures. Empir. Softw. Eng. 23 (4), 2121–2157.

Beck, Kent, 2000. Extreme Programming Explained: Embrace Change, first ed. addison-wesley professional.

Borges, Hudson, Hora, Andre, Valente, Marco Tulio, 2016. Understanding the factors that impact the popularity of GitHub repositories. In: 32nd IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 334–344.

Brito, Aline, Hora, Andre, Valente, Marco Tulio, 2023. Towards a catalog of composite refactorings. J. Softw.: Evol. Process 1, 1–22.

Diniz, Joao P., Cruz, Daniel, Ferreira, Fabio, Tavares, Cleiton, Figueiredo, Eduardo, 2020. GitHub label embeddings. In: 20th IEEE International Working Conference on Source Code Analysis and Manipulation. SCAM, pp. 249–253.

Fard, Amin Milani, Mesbah, Ali, 2013. Jsnose: Detecting JavaScript code smells. In: 13th International Working Conference on Source Code Analysis and Manipulation. SCAM, pp. 116–125.

Ferreira, Fabio, Borges, Hudson Silva, Valente, Marco Tulio, 2022. On the (un-) adoption of JavaScript front-end frameworks. Softw. - Pract. Exp. 52 (4), 947–966.

Ferreira, Fabio, Valente, Marco Tulio, 2023. Detecting code smells in react-based web apps. Inf. Softw. Technol. 1, 1–35.

Fowler, Martin, 1999. Refactoring: Improving the Design of Existing Code, first ed. Addison-Wesley Professional.

Fowler, Martin, 2018. Refactoring: Improving the Design of Existing Code, second ed. Addison-Wesley Professional.

Harold, Elliotte Rusty, 2012. Refactoring Html: Improving the Design of Existing Web Applications, first ed. Addison-Wesley Professional.

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

Hora, Andre, Robbes, Romain, 2020. Characteristics of method extractions in Java: A large scale empirical study. Empir. Softw. Eng. 25, 1798–1833.

Kim, Miryung, Zimmermann, Thomas, Nagappan, Nachiappan, 2014. An empirical study of refactoring challenges and benefits at Microsoft. IEEE Trans. Softw. Eng. 40 (7), 633–649.

Ksontini, Emna, Kessentini, Marouane, do N. Ferreira, Thiago, Hassan, Foyzul, 2021. Refactorings and technical debt in docker projects: An empirical study. In: 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 781–791.

Mazinanian, Davood, Tsantalis, Nikolaos, Mesbah, Ali, 2014. Discovering refactoring opportunities in cascading style sheets. In: 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE, pp. 496–506.

Mesbah, Ali, Mirshokraie, Shabnam, 2012. Automated analysis of CSS rules to support style maintenance. In: 34th International Conference on Software Engineering. ICSE, pp. 408–418.

Murphy-Hill, Emerson, Parnin, Chris, Black, Andrew P., 2011. How we refactor, and how we know it. IEEE Trans. Softw. Eng. 38 (1), 5–18.

Nederlof, Alex, Mesbah, Ali, van Deursen, Arie, 2014. Software engineering for the web: the state of the practice. In: 36th International Conference on Software Engineering. ICSE, pp. 4–13.

Peruma, Anthony, Newman, Christian D., Mkaouer, Mohamed Wiem, Ouni, Ali, Palomba, Fabio, 2020. An exploratory study on the refactoring of unit test files in Android applications. In: 42nd International Conference on Software Engineering Workshops. pp. 350–357.

Serdar, Ceyhan Ceran, Cihan, Murat, Yücel, Doğan, Serdar, Muhittin A., 2021. Sample size, power and effect size revisited: simplified and practical approaches in pre-clinical, clinical and laboratory studies. Biochem. Medica 31 (1), 27–53.

Sharma, Tushar, Suryanarayana, Girish, Samarthyam, Ganesh, 2015. Challenges to and solutions for refactoring adoption: An industrial perspective. IEEE Softw. 32 (6), 44–51.

Silva, Danilo, da Silva, Joao Paulo, Santos, Gustavo, Terra, Ricardo, Valente, Marco Tulio, 2020. RefDiff 2.0: A multi-language refactoring detection tool. IEEE Trans. Softw. Eng. 1 (1), 1–17.

Silva, Danilo, Tsantalis, Nikolaos, Valente, Marco Tulio, 2016. Why we refactor? confessions of GitHub contributors. In: 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE, pp. 858–870.

Silva, Danilo, Valente, Marco Tulio, 2017. RefDiff: Detecting refactorings in version histories. In: 14th International Conference on Mining Software Repositories. MSR, pp. 1–11.

Silva, Hudson, Valente, Marco Tulio, 2018. What's in a GitHub star? Understanding repository starring practices in a social coding platform. J. Syst. Softw. 146 (1), 112–129.

Tang, Yiming, Khatchadourian, Raffi, Bagherzadeh, Mehdi, Singh, Rhia, Stewart, Ajani, Raja, Anita, 2021. An empirical study of refactorings and technical debt in Machine Learning systems. In: 43rd International Conference on Software Engineering. ICSE, IEEE, pp. 238–250.

Tsantalis, Nikolaos, Chatzigeorgiou, Alexander, 2009. Identification of move method refactoring opportunities. IEEE Trans. Softw. Eng. 35 (3), 347–367.

Tsantalis, Nikolaos, Guana, Victor, Stroulia, Eleni, Hindle, Abram, 2013. A multidimensional empirical study on refactoring activity. In: Conference of the Centre for Advanced Studies on Collaborative Research. CASCON, pp. 132–146.

Tsantalis, Nikolaos, Mansouri, Matin, Eshkevari, Laleh M., Mazinanian, Davood, Dig, Danny, 2018. Accurate and efficient refactoring detection in commit history. In: 40th International Conference on Software Engineering. ICSE, pp. 483–494.

Wohlin, Claes, 2012. Experimentation in Software Engineering. Springer Science & Business Media.