Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Youssef Benlemlih

State Management in Component Based User Interfaces

A React Case Study

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Youssef Benlemlih

State Management in Component Based User Interfaces
A React Case Study

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Lars Hamann
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 31. August 2022

Youssef Benlemlih

Thema der Arbeit
State Management in Component Based User Interfaces. A React Case Study

Stichworte
State Management, Software Quality Models, React, EFFORT

Kurzzusammenfassung
Heutzutage werden mehr und mehr mobile Anwendungen und Webseiten entwickelt und genutzt. Eines der führenden Frameworks für die Erstellung von Benutzeroberflächen ist React. Obwohl React im Jahr 2021 das meistgenutzte Web-Framework ist, herrscht Uneinigkeit darüber, wie der Zustand der Anwendung verwaltet werden soll. Diese Arbeit vergleicht gängige Ansätze, indem sie eine Auswahl von State-Management-Bibliotheken untersucht. Ein Software-Evaluierungs-Framework wird verwendet, um die Qualität jeder Bibliothek zu analysieren. Schließlich werden die Ergebnisse verglichen und allgemeine Schlussfolgerungen über die Eignung der einzelnen Ansätze gezogen.

Youssef Benlemlih

Title of the paper
State Management in Component Based User Interfaces. A React Case Study

Keywords
State Management, Software Quality Models, React, EFFORT

Abstract
Nowadays mobile applications and websites are developed and used more than ever. One of the leading frameworks for the creation of user interfaces is React. Although React is the most used web framework in 2021, there is a disagreement on how to manage its application state. This thesis compares common approaches by investigating a selection of state management libraries. A software evaluation framework is used to analyze the quality of each library. Finally the results are compared and general conclusions are made about the suitability of each approach.

Contents

List of Figures

1. Introduction

In today's era of technology, where the virtual world complements our daily lives, websites and mobile applications are used and developed more than ever. Throughout the years, software development has evolved in a fast pasted way, that developing such software products from scratch is no longer necessary. Instead, there are multiple frameworks available for this purpose. One of the leading frameworks used to create interactive web applications is React [30]. According to the 2021 Developer Survey done by Stack Overflow [42], it is the most commonly used web framework as of 2021. Its flexible component based architecture simplifies the creation of sophisticated user interfaces. However, it seems challenging for the developers to come to an agreement, which approach is the most suitable for managing the application state in React. Therefore this work aims to provide a thorough objective comparison of the available approaches. This comparison could serve as the basis for choosing the suitable approach for each individual use case.

1.1. Motivation and Hypotheses

Although it seems that React is established as a successful web framework, it is commonly used with different state management libraries. There appears to be a knowledge gap regarding a thorough analysis of the approaches based on the key software criteria. Therefore this works focuses on the approaches that can be used to manage state in component based user interfaces. This objective is achieved by exploring the available state management libraries used with React. It should be highlighted, that the intention of the work is not the identification of the "best state management library". Rather, it is to gain a thorough understanding from an architectural point of view and easing the process of choosing an adequate solution for a given use case by providing general guidelines based on commonly known software quality characteristics. Hence, following hypotheses are formulated:

- **H1:** React Context is best suited for data which is rarely changed, such as the authenticated user or the current UI theme.
- **H2:** State Management Libraries are suitable for complex User Interfaces (UIs) that don't communicate much with a backend server.
- **H3:** Query approaches are better suited for UIs, that need to interact with a backend server.

1.2. Research Design

The research conducted in this thesis comprises multiple steps. Following the introduction, which aims to give the first impression of the matter, the need for state management libraries is explained in the second chapter. This includes taking an elaborate look at how React functions internally, by introducing its syntax (JSX) and its primitive building blocks (class and function components), clarifying the ways data can be stored and shared in different parts of the UI (props and state), and explaining React's rendering algorithm, referred to as Reconciliation [33]. Following that, problems emerging from the component based architecture are presented along with an example use case. Afterwards, a selection of state management libraries (SMLs) is presented, including code examples and the programming paradigms they draw upon. In the third chapter, the term "software quality model" is defined and the existing models are introduced with their respective advantages and challenges. After defining the requirements for these models, a comparison is presented and the selection of the Evaluation Framework for Free/Open souRce projecTs (EFFORT) [57] is justified. The previously formulated use case is implemented using each of the SMLs and used for comparing them within the application of EFFORT. The results are elaborated and general conclusions are drawn regarding the suitability of each library. Lastly, The work is concluded with a reflection on the research process and an outlook for future work.

2. Conceptual Background

This chapter provides the reader with the needed conceptual background to understand the nature of the problem that the state management libraries aim to solve. First, React is briefly introduced with its syntax and the concept of a component tree. Then, a number of problems arising from React's architecture are presented, followed by a selection of state management libraries. Finally, these problems are illustrated with a use case.

2.1. Introduction to React

In React, the application is modeled as a tree of components, each representing a part of the user interface. In the following figure 2.1, a component called `HelloMessage`, that takes a name as an argument is defined. Arguments are called props (short for properties) and can be accessed inside the component in order to create an element, in this case an `HTML` `div` element [10]. The component implements a `render` method that declaratively describes how to construct the UI. The `render` method is then implicitly used when using the component in an `HTML` inspired syntax called `JSX`.

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

// renders <div>Hello React World</div>
root.render(<HelloMessage name="World" />);
```

Figure 2.1.: Example of a component created with JSX

`JSX` is a syntax extension to JavaScript, that simplifies the creation of components [22]. The usage of `JSX` is optional and can be omitted. For example, the `render` method of the component can be defined in plain JavaScript using `React.createElement(component, props, ...children)`, since `JSX` code is compiled to corresponding function calls [22].

An alternative to the usage of classes is to use a function that receives an object and simply returns the content that was previously in the `render` method. This results in simpler and more readable code, as shown in figure 2.2.

```
function HelloMessage ({name}) {
    return <div>Hello {name}</div>;
}
```

Figure 2.2.: Example of a component defined as function

Components can be nested to create the complete application tree. Figure 2.3 shows a `Parent` component with two children components, `ChildA` and `ChildB`. The created application tree is equivalent to the tree presented in the figure 2.4.

```
function Parent () {
   return <><ChildA/><ChildB/></>;
}
```

Figure 2.3.: Example of a component that includes two children

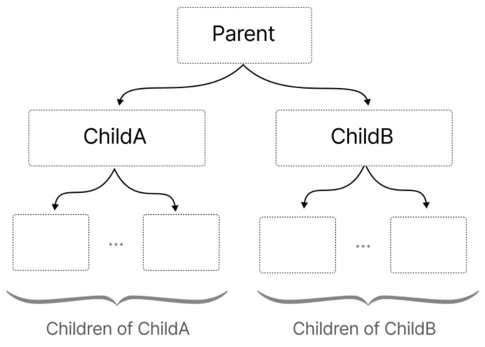


Figure 2.4.: Tree representation of the created components

Up until now, only props have been presented. Props can be thought of as immutable data within the scope of a component. Most applications though, need to have mutable data, also referred to as state, in order to be interactive. In class components, state can be achieved with an initialization in the constructor and the use of the `setState` method for state updates.

Following, the usage of state is illustrated with the example of a simple user interface. Figure 2.5 illustrate the result, where clicking the + button increments the count by one and clicking - decrements it by one. The current count is reflected in real time.

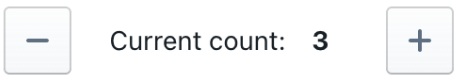


Figure 2.5.: The counter after after clicking + three times.

```
function Counter() {
    const [count, setCount] = useState(0);
    return <Container>
        <Button
            icon={"minus"}
            onClick={() => setCount(count - 1)}
        />
        <span>Current count: <b>{count}</b></span>
        <Button
            icon={"plus"}
            onClick={() => setCount(count + 1)}
        />
    </Container>
}
```

Figure 2.6.: Implementation of the counter using a function component

The figure 2.6 demonstrates the implementation of the counter using a function, in which `Counter` is a component that accepts no props and holds a state. It uses a `Container` element to apply a layout and nests in two button components and an `HTML span`[40]. The button receives an icon name and a callback function to be called when clicked. The current count is embedded in the text inside the span using curly braces.

Although state is mutable, it is not to be assigned directly, but rather by using the method `setState`, so that React is notified by the change and can orchestrate the needed UI updates[43].

The function `useState` is one of the many special functions available by the React API, so called hooks [18]. Hooks are special functions that start with `use` and can only be used in function components or within other hooks. There are other hooks available for function components and lifecycle methods for class components. These are left out of this introductory chapter, since they have a low relevance within the scope of the work. In the following, function components are mainly used over class components due to their simplicity and their support by all the selected state management libraries.

2.2. Problem Definition

Before diving into the different approaches, it is important to identify what problems SMLs try to solve in the first place. The review of the developer documentation of the SMLs indicates that the fundamental problem lies in sharing state between multiple components, which can be split into four problems:

- **P1:** Shared state. Although react offers great flexibility on the structure of the application tree, it prescribes a top-down data flow. Meaning, parent nodes pass data to their children. This introduces the challenge of how to share data between nodes that are far away from one another in the tree. Figure 2.7 illustrates the challenge of sharing state between the components C_1 and D_5.

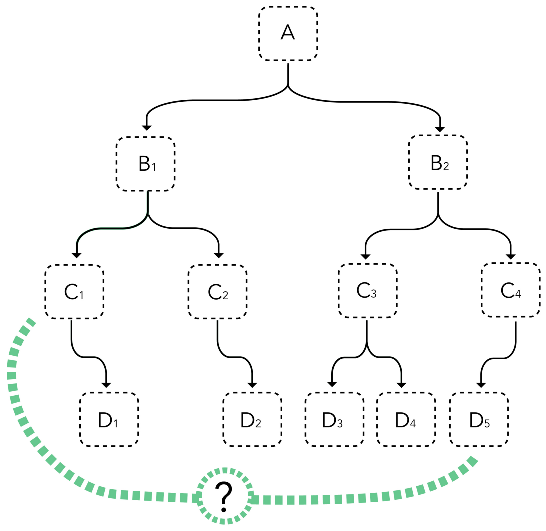


Figure 2.7.: Example situation illustrating the challenge of shared state

- **P2:** Prop drilling: State needs to be forwarded from higher nodes to lower ones that require it, even when intermediate nodes do not utilize it. In figure 2.8, D_3 and D_6 both require the state S, therefore it is imported in the closest common ancestor node A and propagated throughout the tree. As can be seen in the figure, the node B_1, B_2, C_2 and C_4 do not use the state, nevertheless they need to forward it. Typically the lower nodes are **presentational components** [29]: They represent concrete implementations of User Interface elements such as text inputs, buttons and checkboxes. On the other hand, higher nodes are **container components:** they

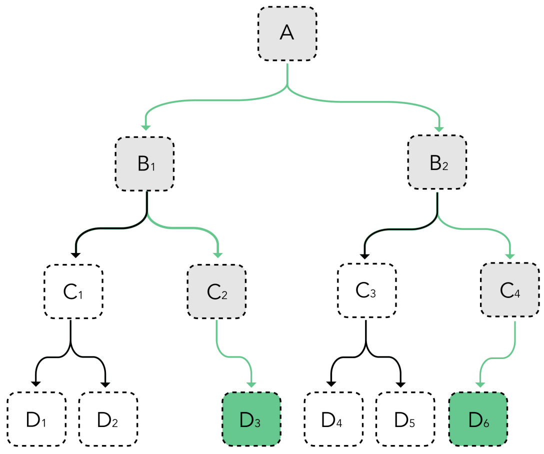


Figure 2.8.: Example situation illustrating prop drilling

contain business logic and use presentational components while being agnostic about implementation details [29]. For instance, prop drilling becomes apparent, when a current application theme that should be passed from the root node throughout the application tree, so that the presentational components can access it and render accordingly.

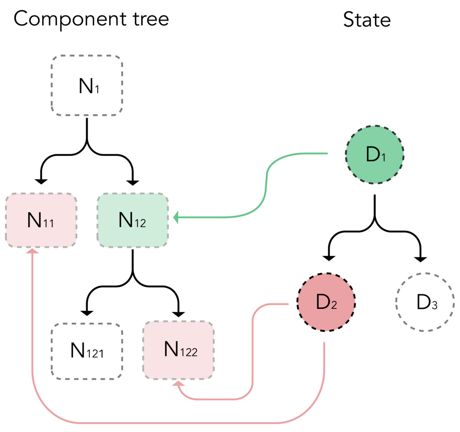


Figure 2.9.: Example situation illustrating derived state

- **P3:** State interdependencies/derived state. It is usual in an application that multiple states are dependent on others forming a state dependency tree. An example of a state dependency tree is depicted in figure 2.9, where D_2 and D_3 depend on D_1. An example for this could be: D_1 is the user selection of an element in a list. D_2 could be a field in the selected object and D_3 is some additional data that needs to be fetched for this specific object.
- **P4:** Unnecessary re-renderings. When the data of a component changes, React re-renders that component and its belonging subtree. In a nutshell, React recursively calls `render` on the component and its children and update the Document Object Model (DOM) [11] accordingly (more about this in chapter chapter 2.4). This process can be exhaustive as it can cause performance problems when one of the affected nodes is high in the application tree. Thereby three re-rendering performance levels are differentiated: the first as **optimal re-renderings**, where only the nodes whose state changed are re-rendered, the second as **suboptimal re-renderings**, where nodes are re-rendered due to their possible change of their derived state, and the third as **redundant re-renderings**, where nodes are re-rendered even when their state has not changed.

2.3. Problem Use Case

In the following, a use case incorporating the above mentioned problems is formulated, that is applied in later steps of this work in the evaluation of the selected approaches.

This use case is an online shop website, which is depicted in Figure 2.10. The displayed articles are fetched from a backend server and can be added to the cart. The count of the articles currently in the cart is shown by a badge on the cart icon. This embodies the problem of derived state (**P3**). The cart state, which is also persisted on the backend side can be viewed by clicking on the cart icon. Since adding items in the cart and the displayed items happen in different sections of the UI, the problem of shared state (**P1**) is therefore included. By adding the possibility to switch between dark and light mode, which is a common pattern nowadays, the theme needs to be shared throughout the application, effectively showing the prop drilling problem (**P2**). In order to test whether the application efficiently handles re-renderings (**P4**), another state is added: the authenticated user. By clicking on the profile icon, a popup prompts the user to log in or out. The authenticated user is shown in the profile icon and popup. Finally, in

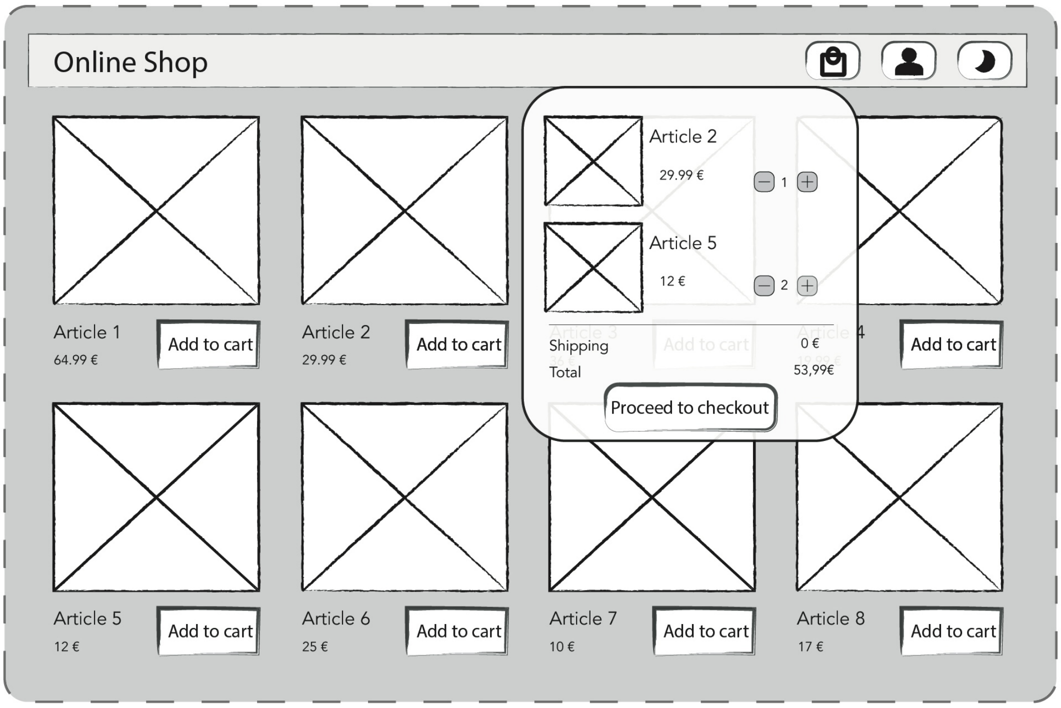


Figure 2.10.: The wireframe of the use case where the cart popup is open

order to add some business logic, a free shipping is offered when the total of the cart surpasses a specific amount.

In order to implement the backend functionality, the application needs to communicate with a backend server. Luckily, this can be easily mocked by creating a service class in the frontend that offers the same interface. Mainly, all methods should include a delay and be asynchronous in order to simulate an implementation using the browsers' built-in `fetch` function [14]. The functionality offered by the service is described in the figure 2.11 below and the used types are shown in figure 2.12.

Method	Returns	Description
`getAllArticles()`	`Promise<Article[]>`	Get all the articles available
`getCart()`	`Promise<Cart>`	Get the current cart, including the articles, their respective count, shipping costs and total
`addArticleToCart (articleId: string)`	`Promise<Cart>`	Add an article to the cart and return the new cart.
`removeArticleFromCart (articleId: string)`	`Promise<Cart>`	Removes an article from the cart and return the new cart.
`logIn()`	`Promise<Profile>`	Logs the user in and returns the user details including the first name, the last name and the user's email address.
`logOut()`	`Promise<void>`	Logs the user out.

Figure 2.11.: API definition of the needed service.

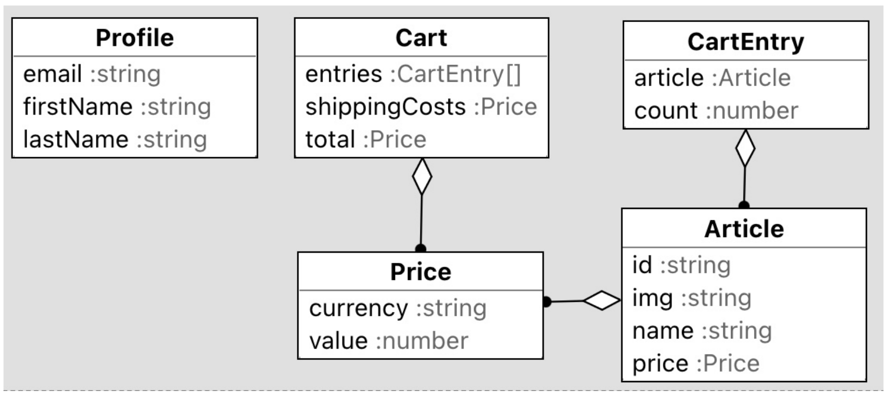


Figure 2.12.: The types used in the API definition of the service

2.4. React's Inner Working

Thanks to React's declarative API, it is simple to create reactive user interfaces, which renders correctly when the underlying data changes. This process of updating the DOM elements is called Reconciliation [33]. Reconciliation is based on the concept of the virtual DOM, where an ideal representation of the UI is kept in memory and synced with the browser DOM. After each change of props or state, the DOM needs to be updated to reflect the changes. To do so, React compares the previous tree with the target tree in a process referred to as "diffing" [33]. Whilst calculating the needed operations to transform a tree to another one has a complexity of $O(n^3)$ [33], React uses a heuristic algorithm that achieves a lower complexity of $O(n)$. The algorithm makes two basic assumptions that prove to be valid in almost all practical use cases. The first assumption is that two nodes of different types produce different trees. And the second implies, the prop `key` can be used to define the identity of a node.

The process of Diffing works as follow: Starting from the root node, the `render` method is called and compared with the previous content. If elements of different types are returned, then the old tree is removed and replaced with the new one. Otherwise, the attributes are compared and only the needed changes are made. The algorithm is then recursively applied to the child components. For example in figure 2.13, only the prop `className` needs to be modified. In other words, when a component is re-rendered, it is not removed from the application tree and re-mounted. Rather, its `render` method is called and the needed changes are calculated.

```
// actual
<div className="before" title="my-title" />

// target
<div className="after" title="my-title" />
```

Figure 2.13.: Two nodes to compare that have a different attribute

When recursing on the child components, both lists are simply iterated over and the needed change operations are computed. For example, when adding an element to a collection such as in figure 2.14, the comparison of the first two entry sets result in no change needed. Then, upon arriving at the third element, it becomes apparent that the element `<li>third</li>` need to be added.

```
// actual
<ul>
  <li>first</li>
  <li>second</li>
</ul>

// target
<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

Figure 2.14.: Two trees to be compared, where the second one has an extra element in the end

This implementation leads to redundant operations when elements are added in other places than at the end. For instance, when an element is added in the beginning the so far presented procedure will result in changing the content of the two first nodes and adding a third one.

This issue is solved by using the unique `key` attribute to identify the nodes. Figure 2.15 gives an examples that improves the previous code.

```
// actual
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

// target
<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

Figure 2.15.: Two trees to be compared. The usage of keys indentifies that an element has been added

2.5. State Management Libraries

Following, a selection of state management libraries is presented. Two factors has been taking into consideration when choosing the libraries: the popularity of the library and the programming paradigm it is based on. The selected libraries are React Context [3] which falls into the global state category, Redux Toolkit [35] which follows a flux based approach [17], MobX [31] which follows the observable pattern, Recoil [32] which is based on the concept of atomic state, and React Query [47], a query based approach.

2.5.1. Global State

Built into React, React Context provides the possibility of sharing a piece of state in a defined context. The usage of React Context requires the creation of a context object and a context provider. As the name suggests, the context provider holds the actual state that is shared with the consumers. Its usage in the component tree defines the scope of the context by wrapping a part of the component tree. All the descendent components of the provider have access to the shared state through the use of the hook `useContext`. Figure 2.16 shows the creation and usage of a context globally. It is to be noted, that there is usually a one-to-many relation between context providers and consumers, although multiple providers can be used to overwrite values deeper in the components tree [3].

When using React Context, the consumer components become functions that contain implicit parameters that are not declared in their signatures. This means that they behave differently depending on the context they are called from. React Context is therefore considered as a form of dynamic scoping, where the variable binding (the association of a variable to its actual value) happens dynamically [82]. According to [82], “this allows for a number of well-known benefits [...], like conciseness, modularity and adaptability”. When having a unique provider shared throughout the application however, React Context can be interpreted as a form of global state.

```
// Context creation
const MyContext = createContext();

// Creating the context provider
export const MyContextProvider = ({
    children,
}) => {
    const [state, setState] = useState(/* initial state */);
    return (
        <MyContext.Provider value={{ state, setState }}>
            {children}
        </MyContext.Provider>
    );
};

// usage in a consumer component
export const { state, setState } = useContext(MyContext)
```

Figure 2.16.: Example usage of React Context.

2.5.2. Flux-Based State Management Libraries

One of the popular state management libraries is Redux [34]. It has gained popularity since it drew from Flux pattern, a pattern that has been recommended for React [16]. An application using the flux pattern is made up of three main components shown in figure 2.17:

- The **state**, the single source of truth that drives the application,
- The **view**, a set of UI components that renders based on the current state, and
- The **actions**, the events that occur in the application based on user interaction, and trigger updates in the state.

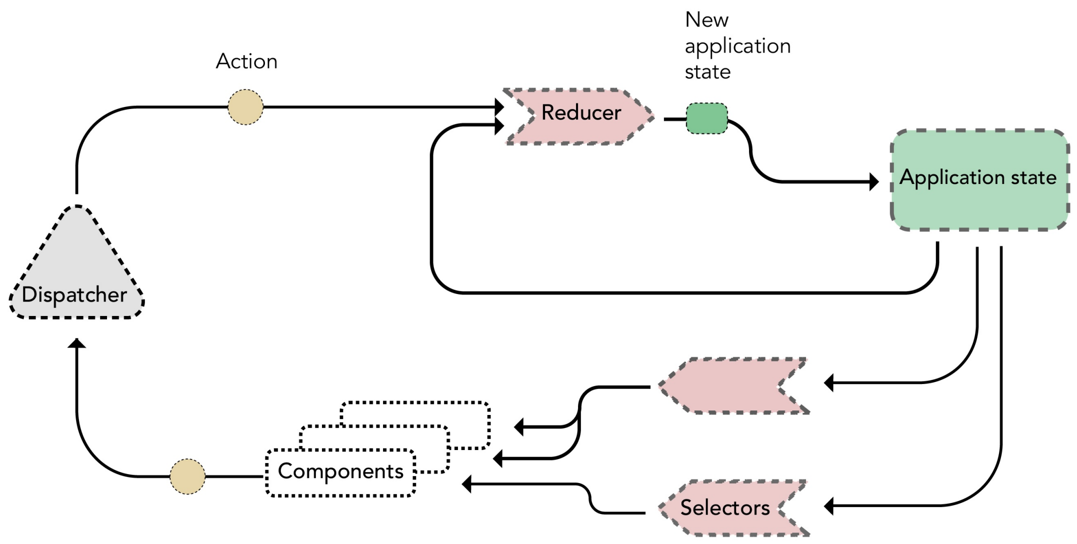


Figure 2.17.: The components of an application using Redux

This definition of the paradigm enforces a one way data flow, meaning that the application state cannot be changed directly, but through actions, which leads to more predictability. Actions are nothing more that a JavaScript object containing a type that typically follows the pattern `"domain/eventName"` along with a `payload` field containing other parameters. Figure 2.18 shows a simple action and a dynamic action creator.

When an action is dispatched, it is passed to a **reducer** function along with the old state and returns the new one. Figure 2.19 shows an example of a todos application. The list of todos that is initially empty is extended when the action with the type `todos/add` is

```
// a constant action
const addTodoAction = {
  type: 'todos/add',
  payload: 'Buy milk'
}

// a dynamic action creator where the payload is parameterized
const createAddTodoAction = (payload) => ({
  type: 'todos/add',
  payload
})
```

Figure 2.18.: Examples of an action and an action creator

dispatched. It is to be noted, that reducers are pure functions, meaning that the output of the function depends strictly in the provided parameters.

```
function todosReducer(state = { todos: [] }, action) {
  // Check to see if this reducer cares about this action
  if (action.type === 'todos/add') {
    // If so, add a new todo item
    return {
      todos: [
                ...state.todos,
              {
                    name: action.payload,
                    checked: false,
                    id: getNewId()
                }
    }
  }
  return state
}
```

Figure 2.19.: Example of a Redux reducer used in a todos application

Since the team behind Redux strongly recommends the usage of the library Redux Toolkit [44], it is selected in this work. Redux Toolkit provides an improved API that addresses a few disadvantages of Redux, mainly the big amount of boilerplate code and the difficulty of configuring a Redux store [36]. Most notably, the actions and action creators can be generated by the library. Redux Toolkit also introduces the concept of slices [7] which are units of state with a specific domain that can be combined to create the root state [2]. Each slice contains the state, reducers, and action creators. Additionally, Redux Toolkit simplifies the creation of async actions [52], that include a loading state and a success or failure result with the provided function `createAsyncThunk` [6].

Before moving to observable state management libraries, it is worth mentioning that Redux was inspired not only by the original flux paradigm [17], but also Command Query Responsibility Segregation (CQRS) and Event Sourcing [25]. In a nutshell, CQRS is a pattern that encourages splitting the application models into separate models for update and display [5]. To quote [5]:

“At its heart is the notion that you can use a different model to update information than the model you use to read information”.

This is analogous to Redux Toolkit’s selectors and action creators. Figure 2.20 shows an application not using CQRS and figure 2.21 one that does.

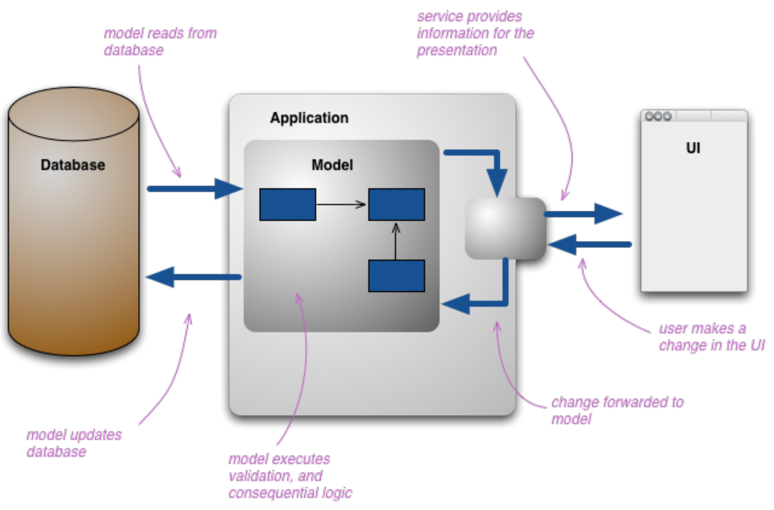

model reads from database
service provides information for the presentation
Application
Database
Model
UI
user makes a change in the UI
change forwarded to model
model updates database
model executes validation, and consequential logic

Figure 2.20.: Example of an application not using Command Query Responsibility Segregation (CQRS). Reproduced from [5]

Event sourcing on the other hand, aims to make the changes of the application state transparent. This is achieved by having events (or commands) for each change to the application state [12]. In order to obtain the actual application state, the events must be applied. The main benefits of this pattern lie in the possibility to trace the changes of the application state and to reconstruct past states.

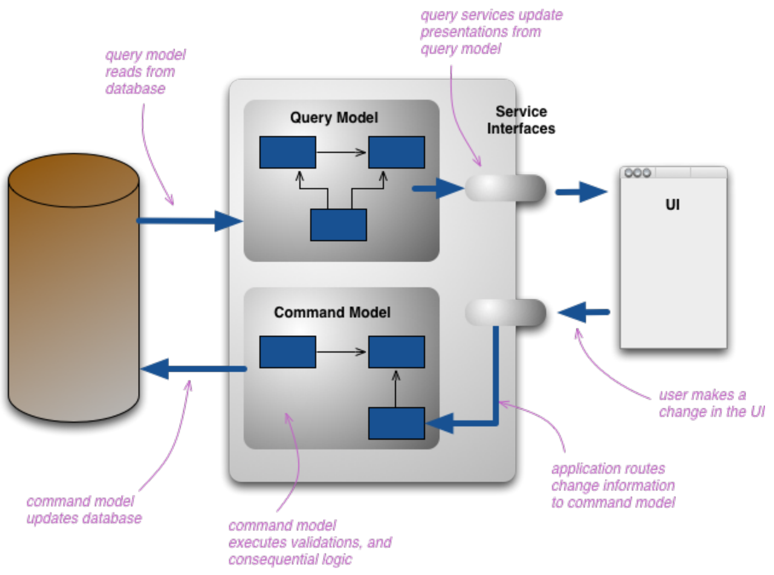

query services update presentations from query model
query model reads from database
Query Model
Service Interfaces
UI
Command Model
user makes a change in the UI
application routes change information to command model
command model updates database
command model executes validations, and consequential logic

Figure 2.21.: Example of an application using Command Query Responsibility Segregation (CQRS). Reproduced from [5]

2.5.3. Observable-Based State Management Libraries

MobX is a simple library based on the observer pattern [68], that hides multiple implementation details. Using the terminology of the observer pattern, the state of the application constitutes the "Subject" whereas the components that use the state are the "Observers" [68]. In order to create Observables, the provided function `makeAutoObservable()` needs to be called in the constructor of a class, then an instance can be created and used anywhere needed. Each component that uses the created state is wrapped with `observer()`. Consequently, each change made to the state object implicitly causes the notification of all subscribers.

Compared to the original observer pattern depicted in figure 2.22, MobX offers a minimalistic interface. The logic behind emitting and listening to events is hidden behind a general interface.

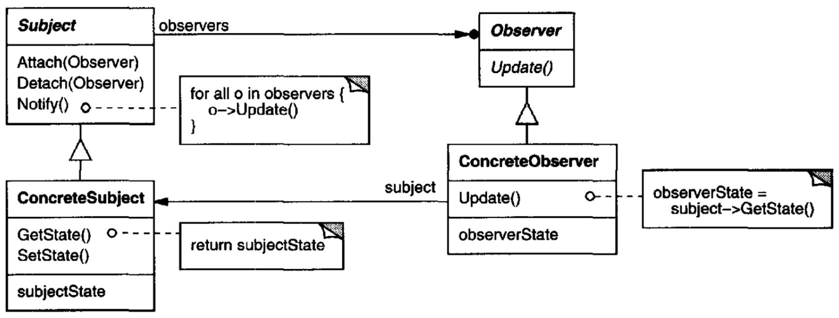


Figure 2.22.: Structure of the Observer Pattern. Reproduced from [68]

MobX's user documentation refers to this as "transparently applying functional reactive programming"(TFRP) [31]. Transparent in TFRP refers to the fact, that connecting the observables to the observers requires "no explicit wiring" [51]. MobX is considered Reactive because changes in the observables emit the correct events efficiently, without excessive events or polling [51]. The functional programming aspect comes from the possibility to "apply transformations on input [observable] data and produce output values"[51].

The usage of Functional Reactive Programming, according to [70], addresses many of the software engineering principles that the classical observer pattern violates, which are "Uniformity and abstraction", "Encapsulation", "Resource management", "Side-effects", "Composability", "Separation of concerns", "Scalability" and "Semantic distance" [70].

2.5.4. Atom-Based State Management Libraries

Recoil [32] is a state management library that takes a different approach on modeling the application state. In contrast to the previously presented approaches that are based on a centralized application state, Recoil offers a distributed state approach by introducing the concept of atoms [1] and selectors [39]. An atom is simply a unit of state, whereas a selector is a special function that derives its states from one or more atoms or selectors. This allows the creation of a complex application state tree, where a changing atom leads to the re-evaluation of all the dependents [4].

Figure 2.23 illustrates the API provided by Recoil by using an example. In the example, an atom called counter is defined along with a selector that returns the value of the counter times ten. The usage in React components is straightforward, since the method signature resembles React's built-in `useState` hook [53].

```
// atom definition
const counter = atom({
  key: 'counter', // gloabally unique identifier
  default: 0, // the initial value
});

// a selector that computes the value of the counter times ten
const counterTimesTen = selector({
  key: 'counterTimesTen',
  get: ({get}) =>  get(counter) * 10
});

// usage in a React function component
const [count, setCount] = useRecoilState(counter);
```

Figure 2.23.: Defining and using state by means of an atom

2.5.5. Query-Based State Management Libraries

The last selected approach is the query based state management libraries represented by the library React Query[47]. Although including this approach as an SML is questionable, it is still taken into account, since it serves the objectives of this work nonetheless.

React Query's purpose is to manage server state on the client [28] by solving common problems including caching, grouping multiple requests for the same data into a single request, detecting and updating out-of-date data in the background, managing memory and garbage collection of server state, pagination, and lastly lazy loading data [28]. Although the library has various features to offer, caching seems to be the most relevant one since it allows sharing state across different components. Figures 2.24 and 2.25 show a simple caching situation where two components need to access the same information that is stored on a server. When the first component requires the data, it is fetched from the server, but when consecutive components require the same data, it is taken directly from the cache.

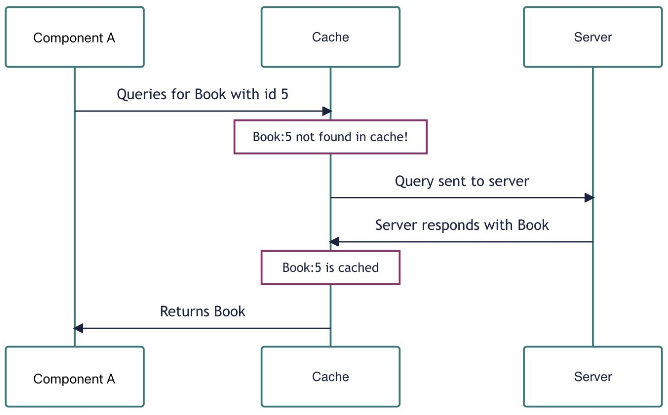


Figure 2.24.: When the first component requires the data, it is queried from the server and saved in the cache

The possibility of interacting with the cache also introduces practicable advantages. For example, when the client performs a mutation that leads to a changed state, the respective cache can be invalidated. This means, that by the next time the data is required, it is fetched from the server. Another use-case which is relevant nowadays is the topic of "optimistic updates" [27]. "Optimistic updates" contribute to a more responsive user experience by displaying applied changes to the user before they are

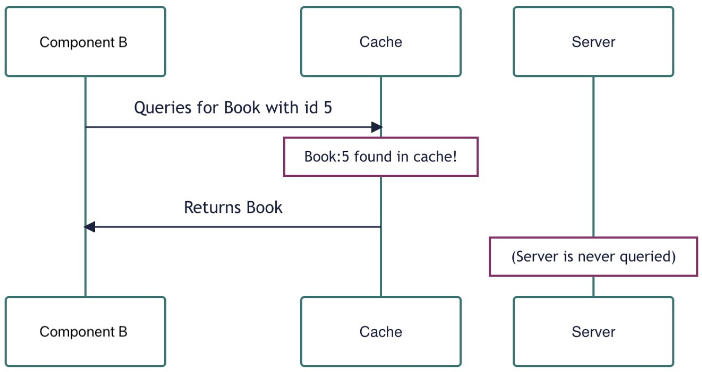


Figure 2.25.: When a second component requires the data it is taken from the cache

accepted by the backend server. This can be easily achieved by overwriting the relevant cache entry manually, since changes in the cache are reflected instantaneously in the components. Figure 2.26 displays an example usage of the library in a component, where a selection of fields is deconstructed [9].

```
const {
        // the data that has been fetched, originally undefined
        data,
        // if applicable, the error thrown by fetchTodoList
        error,
        failureCount,
        // the status of the query: one of 'loading', 'error' and 'success'
        status,
        // a function to manually refetch the query
        refetch,
        // a function to remove the query from the cache
        remove,
} = useQuery(
                        ['todos'], // query keys used for caching
                        fetchTodoList // the function used to fetch the data
)
```

Figure 2.26.: Example usage of React Query with deconstructed relevant fields

3. Software quality models

In order to evaluate and compare the different software solutions mentioned in the previous chapter, it is important to determine the requirements they should fulfill i.e. an adequate “software quality model”. The following chapter introduces software quality models, including the definition of the term and an overview on different models available along with their strengths and weaknesses, and a special focus is put on the ISO/IEC 25010 Model. Furthermore, requirements for software quality models are presented based on the Definition-Assessment-Prediction classification [63] and Open Source Software Quality Models are compared.

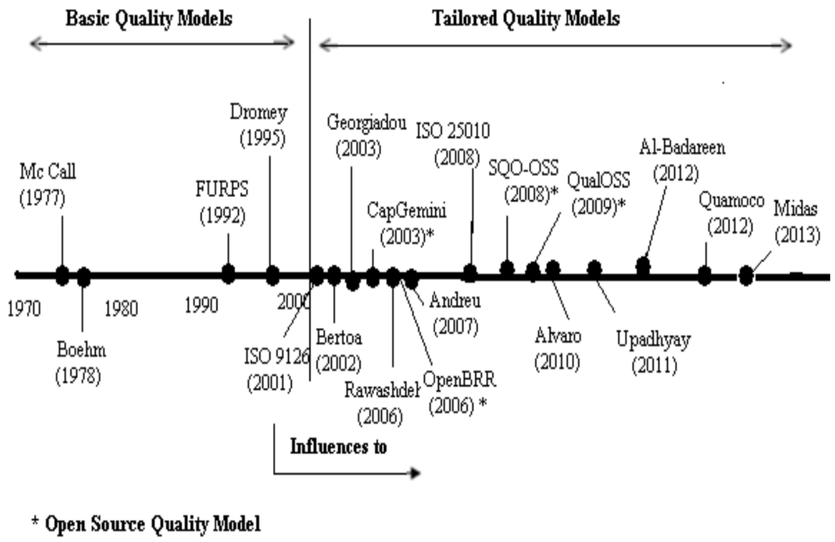


Figure 3.1.: Quality Models. Reproduced from [74]

ISO/IEC 9126-1 defines the term “quality model” as “the set of characteristics, and the relationships between them that provides the basis for specifying quality requirements and evaluation” [62] (cited in [74]). Since different quality models can result in different evaluation results, the available quality models have to be taken into consideration before adopting one. [74] have studied the different quality models (displayed in figure 3.1) and compared their completeness regarding the characteristics that the models consider.

To give an overview, a few models are presented along with their portrayed characteristics in [74]. The McCall Model [73], introduced in 1977, proposes many characteristics to define software quality, grouped into three higher order groups. **Product Review** includes *Maintenance*, *Flexibility*, and *Testing*, **Product Operation** contains the qualities *Correct*, *Reliable*, *Efficient*, *Integrity* and *usability*, and finally **Product Transition** includes *Portability*, *Reusability* and *Interoperability* [74]. Figure 3.2 shows a representation of the model.

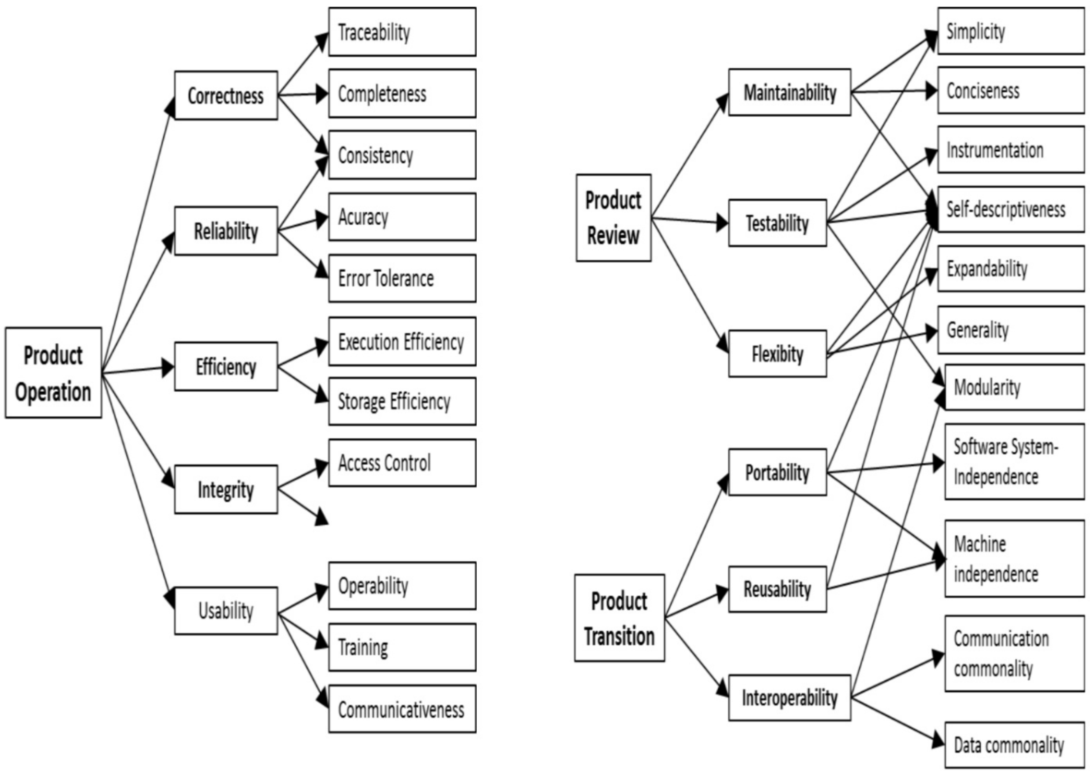


Figure 3.2.: McCall Quality Model. Reproduced from [74]

Although the McCall model considers the relation between quality characteristics and metrics, it lacks in accuracy, since the metrics can only have a boolean value [74]. Moreover, it does not include the characteristic **Functionality** or the possibility to add use case specific characteristics, which can be a drawback for the user [74].

The Boehm model [59] presents improvements over the McCall model by adding a few factors [74], as can be seen in figure 3.3.

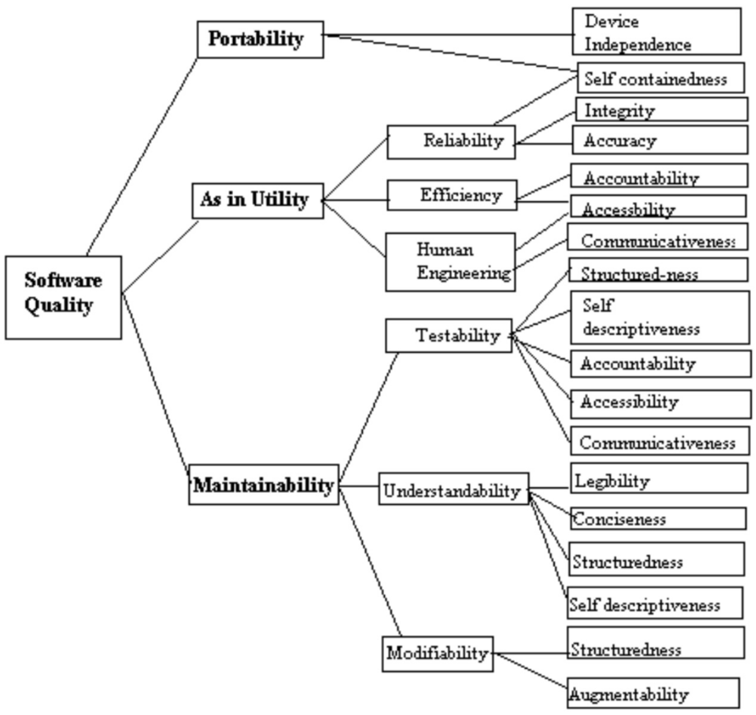


Figure 3.3.: Boehm Model. Reproduced from [74]

The Dromey model [64], which is depicted in figure 3.4 below, clusters the characteristics into four groups: *Correctness*, *Internal*, *Conceptual*, and *Descriptive* [74]. Although this model allows for a more dynamic evaluation to be done, its practical application is unclear. It is therefore used as the basis for other models [74].

The FURPS model established in 1992, is a composition of the characteristics **F**unctionality, **U**sability, **R**eliability, **P**erformance and **S**upportability (or Product Support) [66], as displayed in figure 3.5. Although the model differentiates between functional and non functional requirements, it is missing a few main characteristics, such as portability [74].

The ISO/IEC 9126 model bases on the McCall and Boehm models and differentiates between external and internal qualities [63]. Internal quality attributes, which can be considered as a form of intrinsic quality attributes, can be evaluated without the need of executing the software. In contrast, the external quality attributes can be assessed

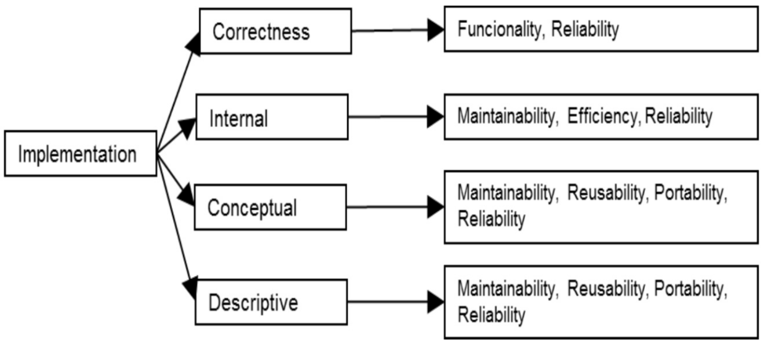

Implementation
Correctness
Funcionality, Reliability
Internal
Maintainability, Efficiency, Reliability
Conceptual
Maintainability, Reusability, Portability, Reliability
Descriptive
Maintainability, Reusability, Portability, Reliability

Figure 3.4.: Dromey Model. Reproduced from [74]

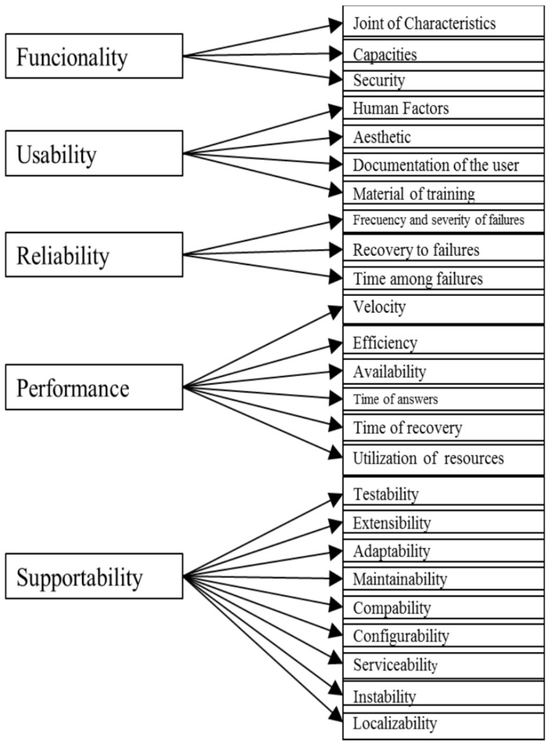


Figure 3.5.: FURPS Model. Reproduced from [74]

during the execution [63]. The model introduces another group of qualities, Quality in use, which includes the *effectiveness of the product*, *productivity*, *security offered to the applications*, and *satisfaction of users*.

Lastly, the ISO/IEC 25010 Model is an extension of the ISO/IEC 9126 [80]. The main improvement lies in the addition of the characteristics **Security** and **Compatibility** [55].

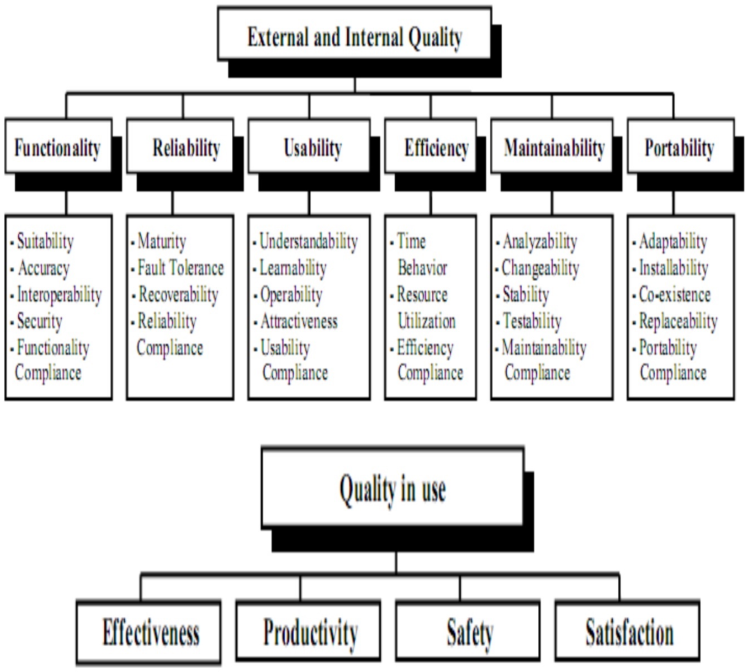


Figure 3.6.: ISO/IEC 9126 Quality Model. Reproduced from [74]

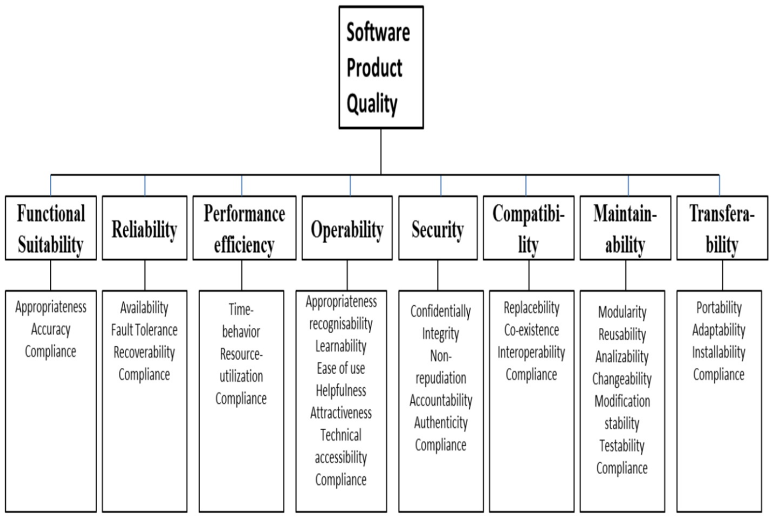
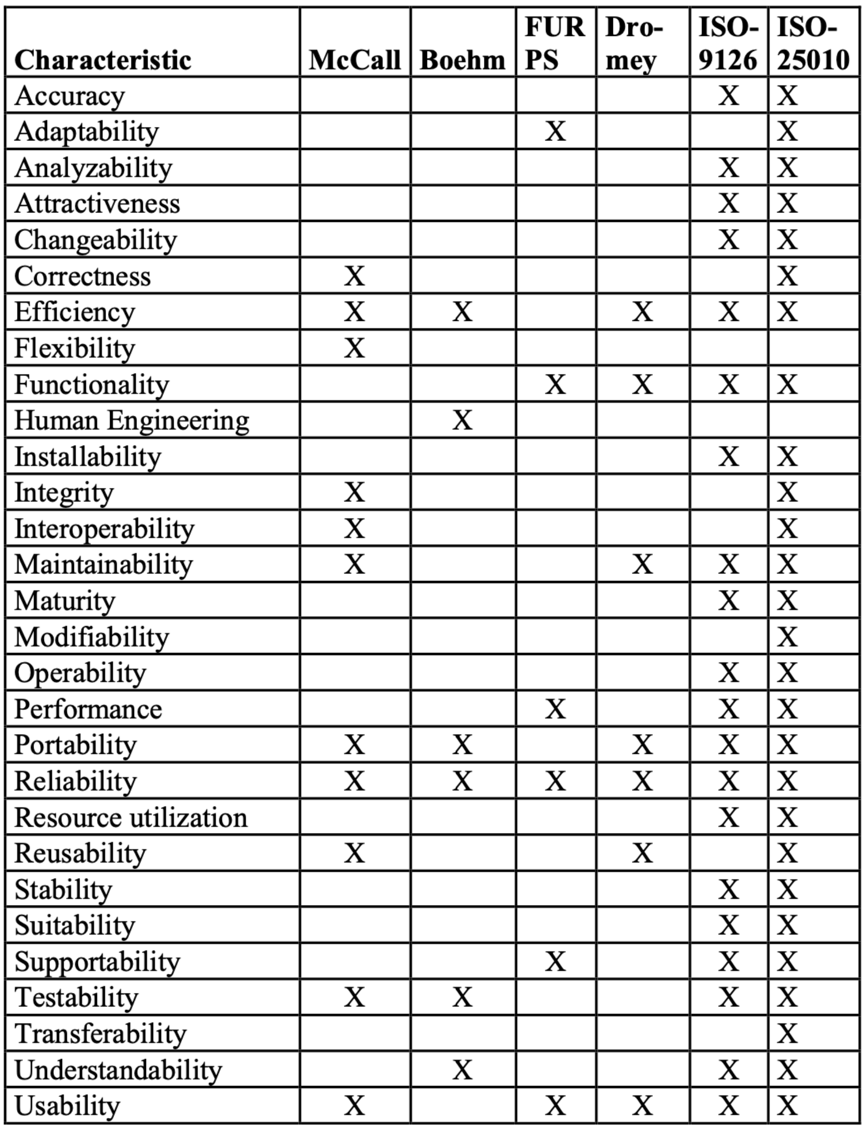


Figure 3.7.: ISO/IEC 25010 Quality Model. Reproduced from [74]

The result of the comparison made by [74] can be seen in figure 3.8, where ISO/IEC 25010 is regarded as most complete.



Characteristic	McCall	Boehm	FUR PS	Dro-mey	ISO-9126	ISO-25010
Accuracy					X	X
Adaptability			X			X
Analyzability					X	X
Attractiveness					X	X
Changeability					X	X
Correctness	X					X
Efficiency	X	X		X	X	X
Flexibility	X					
Functionality			X	X	X	X
Human Engineering		X				
Installability					X	X
Integrity	X					X
Interoperability	X					X
Maintainability	X			X	X	X
Maturity					X	X
Modifiability						X
Operability					X	X
Performance			X		X	X
Portability	X	X		X	X	X
Reliability	X	X	X	X	X	X
Resource utilization					X	X
Reusability	X			X		X
Stability					X	X
Suitability					X	X
Supportability			X		X	X
Testability	X	X			X	X
Transferability						X
Understandability		X			X	X
Usability	X		X	X	X	X

Figure 3.8.: Comparision of quality models. Reproduced from [74]

3.1. ISO/IEC 25010 Model

ISO/IEC 25010 is a quality model that defines which "quality characteristics will be taken into account when evaluating the properties of a software product" [19]. This model is used in different areas of software engineering and architecture, for example in the architecture documentation template arc42 [49]. The quality of a system or software product is thereby defined as the degree to which it satisfies the needs of the stakeholders, that according to [19] are grouped in eight characteristics: **Functional Suitability**, **Performance Efficiency**, **Compatibility**, **Usability**, **Reliability**, **Security**, **Maintainability**, and **Portability** as portrayed in figure 3.9. Requirements are defined across these characteristics to clarify the expectations that need to be fulfilled by the presented software product [19].

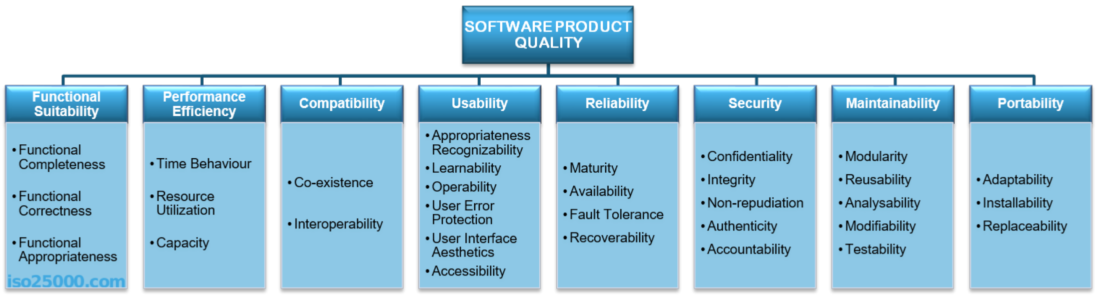


Figure 3.9.: Quality characteristics of ISO/IEC 25010 model. Reproduced from [19]

Functional Suitability is the first characteristic of ISO/IEC 25010 and is defined as the “degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions”[19]. This includes prominently *Functional Completeness*, which refers to the extend to which the tasks/objectives of the user are covered, *Functional Correctness*, meaning to which degree the system provides the “correct results with the needed degree of precision”[19] and *Functional appropriateness*, and how easy the tasks and objectives are accomplished [19].

The second characteristic is **Performance Efficiency**, which considers the performance of the software product in regard to its *Time Behavior*, *Resource Utilization*, and *Capacity* (maximum limits)[19].

The third characteristic is **Compatibility** and contains two pillars: *Co-existence*, the extend to which it can function efficiently “while sharing a common environment or resources with other products”[20], and *Interoperability*, referring to “the ability of two or more software components to cooperate despite differences in language, interface, and execution platform” [84].

The fourth characteristic is **Usability**, which focuses on the interaction between the user and the software product and includes the sub-characteristics: *Operability*, *Learnability*, *User Error Protection*, *Appropriateness Recognizability*, *User Interface Aesthetic*, and *Accessibility* [20].

The fifth characteristic is **Reliability**, which refers to the degree to which the software product performs “specified functions under specified conditions for a specified period of time” [20]. This characteristic comprises the sub-characteristics *Maturity* (the degree to which the needs for reliability are met under normal operation), *Fault Tolerance* (to which extend the software product works correctly in the presence of software or hardware faults), and *Recoverability* (to which degree the system can recover its normal state and affected data after an interruption or a failure) and *Accessibility* [20].

The sixth characteristic is **Security** and is defined by the sub-characteristics *Confidentiality* (the degree to which data is accessible to only authorized entities), *Integrity* (the protection of data from unauthorized changes), *Non-repudiation* (committed actions or events can be proved and cannot be repudiated afterwards), *Accountability* (where actions can be traced back to the committing entity), and *Authenticity* (where “the identity of a subject or resource can be proved to be the one claimed”) [21].

The next characteristic is **Maintainability** and includes different sub-characteristics ranging from *Modularity* (the degree to which the product is split into components whose changes don’t affect others) to *Reusability*, *Testability*, *Modifiability* (how easy it is to modify the product without introducing bugs), and finally *Analyzability* (how easy it is to assess the impact of changes as well as the ease of diagnosis) [21].

The last characteristic is **Portability** and refers to the ability of the product to be transferred to another environment. It comprises *Adaptability* (the degree to which the product can evolve or adapt to a changing environment), *Installability* (how easy it is to be installed or removed from a specified environment), and *Replaceability* (how easy it is to be replaced by another product with the same purpose in the same environment) [21].

3.2. DAP classification

When it comes to practically applying the above mentioned ISO/IEC 25010 standard, many ambiguities arise. Namely, no instructions are provided on how to measure the defined software criteria. [63] addresses this problem by highlighting that the umbrella term “quality model” includes models that aim to fulfill different purposes. For example, the goal of ISO/IEC 9126 (the predecessor of ISO/IEC 25010) is to **define** quality, whereas the maintainability index [61], a metric-based approach, is used to **assess** quality, and finally stochastic models such a reliability growth models (RGMs) [69] can be used to **predict** quality [63]. Although the specified goals are different, these models depend on each other. Quality can’t be assessed without first being defined. Similarly, it is inconceivable to predict quality without knowing how to measure it [63]. Figure 3.10 illustrates the relation between the three model categories: Definition, Assessment and Prediction (DAP), as well as the positioning of ISO/IEC 9126, MI and RGM, and an ideal model which covers all three aspects.

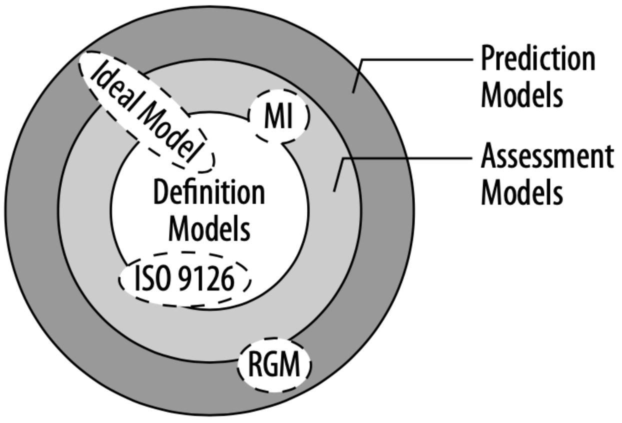


Figure 3.10.: DAP classification for Quality Models: Reproduced from [63]

Another critique on the existing quality models is the lack of precision that leads to different possible interpretations [63]. Since software systems can vary largely, the quality models should offer additionally the possibility of customization [63].

In [55], a comparison of Open Source Software (OSS) Quality Models, that are based on ISO/IEC 25010 and ISO/IEC 9126, was conducted. The compared OSS Quality Models include *Open Source Maturity Model (OSMM)* [65], *QSOS* [77], *Open Business*

Readiness Rating (Open BRR) [83], *Sung et al. Model* [81], *QualOSS* [78], *OMM* [75], *SQO-OSS* [76, 79] and *Evaluation Framework for Free/Open souRce projecTs (EFFORT)* [57].

The study [55] resulted in EFFORT being favored over the rest. One of the reasons is that EFFORT is a second generation [67] Free/Libre and Open Source Software (FLOSS) quality model. It was preferred above QualOSS since it included characteristics from both Product Quality and Quality in Use [55]. Additionally, there have been practical applications of the model within the context of Customer Relationship Management (CRM) [56] and Enterprise Resource Planning (ERP) Systems [57]. Another reason for selecting EFFORT, is that it comes close to the ideal model proposed in the DAP model: The definition aspect is provided by specifying the quality characteristics in form of goals and questions, the assessment aspect through the usage of metrics, and the prediction aspect through the consideration of community trustworthiness and product attractiveness, as is presented in further detail in the next chapter.

4. EFFORT

EFFORT, short for **E**valuation **F**ramework for **F**ree/**O**pen sou**R**ce projec**T**s, is a framework which supports not only the evaluation of product quality but also community trustworthiness and product attractiveness [57]. Additionally the framework is defined in a generic manner and can be instantiated to analyze software systems and products within a specific context.

4.1. EFFORT Baseline Version

As an Open Source Software Quality Model, EFFORT leverages the advantages of OSS for generating extensive results.

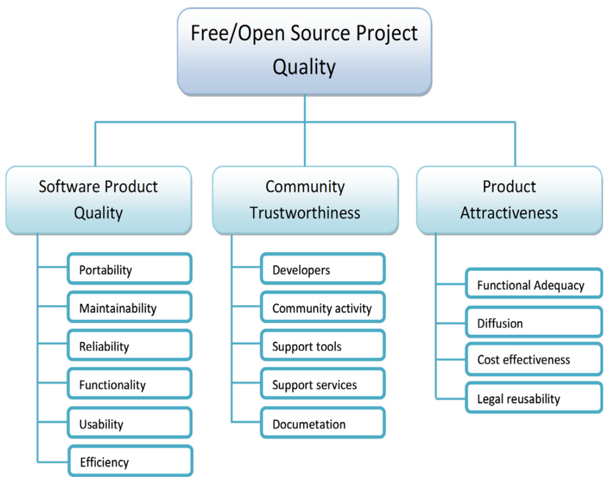


Fig. 2. Quality model defined by EFFORT and associated quality characteristics.

Figure 4.1.: Quality model defined by EFFORT and associated quality characteristics. Reproduced from [57]

.

The quality of a software product according to EFFORT is a combination of three main quality characteristics [57], as shown in Figure 4.1.

- **Quality of the product**, which is evaluated on the basis of the ISO/IEC 9126 standard.
- **Trustworthiness of the community** of developers and contributors, which is an indicator of "the degree of trust that a user has in a community with respect to the support offered" [57]
- **Product attractiveness**, which "considers all the factors that influence the adoption of a product by a potential user, who perceives convenience and usefulness in using it" [57].

In order to specify the proposed characteristics and sub-characteristics, the Goal/Question/Metric (GQM) paradigm [58] is used. The Goal/Question/Metric paradigm is "a mechanism for defining and interpreting software measurement" that "represents a systematic approach for tailoring and integrating goals with models of the software processes, products and quality perspectives of interest based upon the specific needs of the project and the organization" [58].

A critique made by [63] is that most definition models structure their quality attributes in a hierarchy, which causes difficulty while locating elements and can lead to redundancies [63]. Luckily, this issue is addressed in the GQM paradigm where questions can contribute to more than one goal [58]. This can be seen in figure 4.2, in which the goals *Goal2* and *Goaln* contribute to the same question *Question6*.

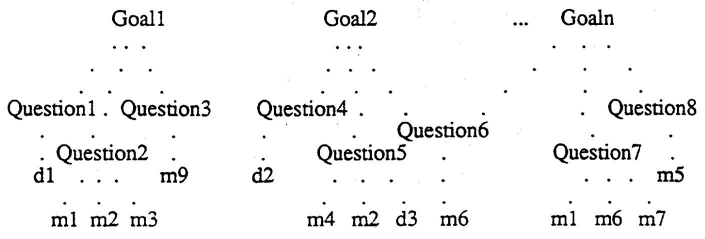
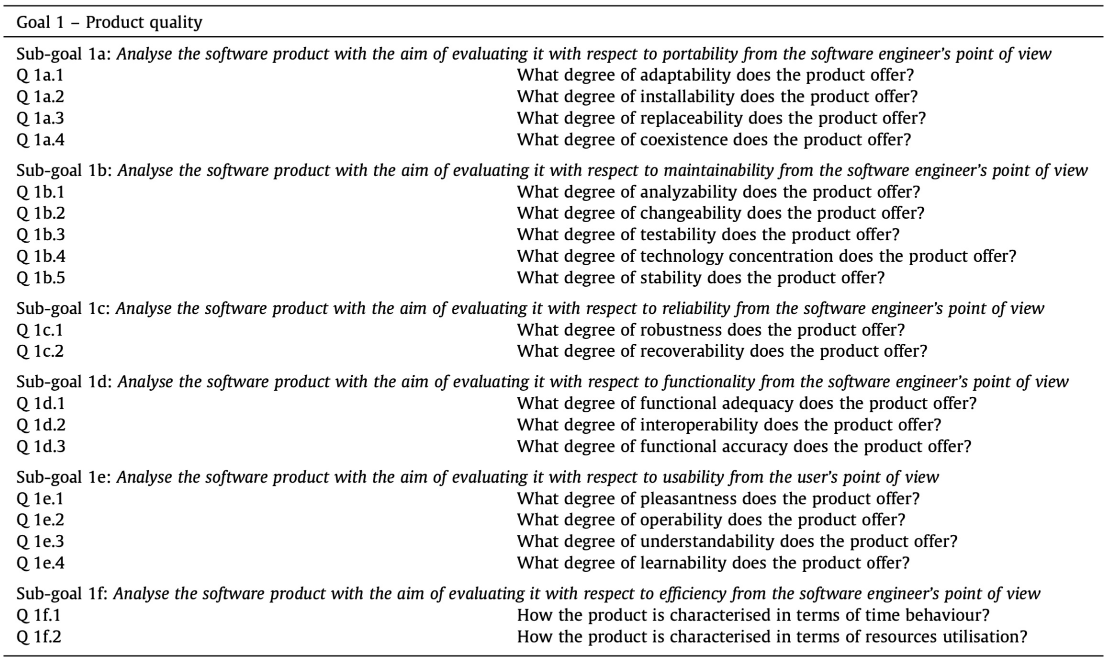


Figure 4.2.: The structure of Goals, Questions and Metrics in the GQM Paradigm [58]

Within GQM, the goals need to be specified, which are refined into a set of quantifiable questions that, in turn, define the metrics used to measure the corresponding outcomes [58]. Because of the large scope of the first goal of EFFORT, *Software Product*

Quality, its subcategories are modeled as sub-goals. Figures 4.3, 4.4, 4.5, 4.6, and 4.7 show the questions related to each goal along with their associated metrics.

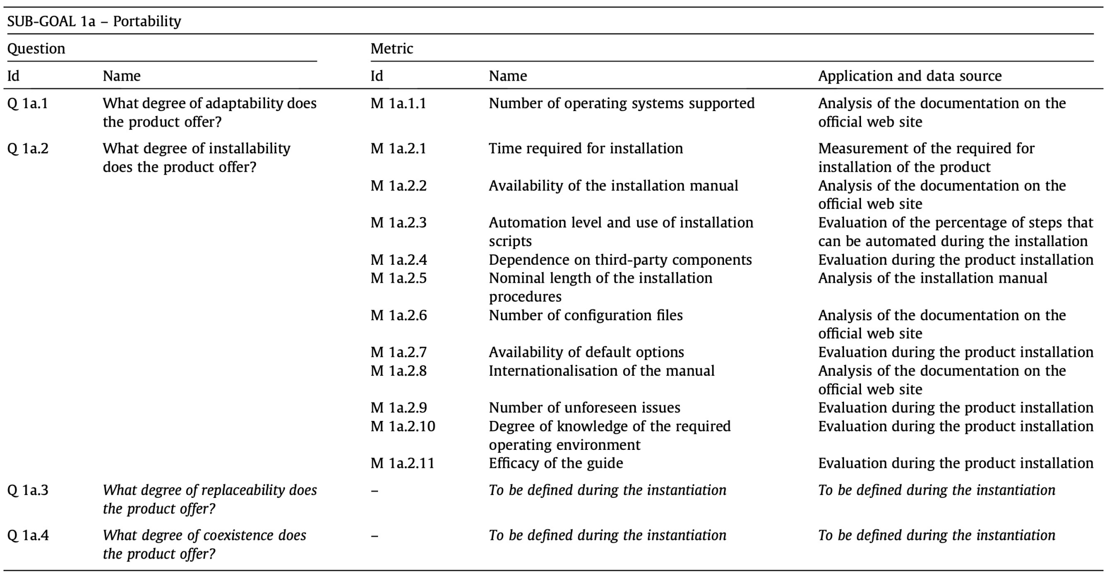


Goal 1 – Product quality	
Sub-goal 1a: *Analyse the software product with the aim of evaluating it with respect to portability from the software engineer's point of view*	
Q 1a.1	What degree of adaptability does the product offer?
Q 1a.2	What degree of installability does the product offer?
Q 1a.3	What degree of replaceability does the product offer?
Q 1a.4	What degree of coexistence does the product offer?
Sub-goal 1b: *Analyse the software product with the aim of evaluating it with respect to maintainability from the software engineer's point of view*	
Q 1b.1	What degree of analyzability does the product offer?
Q 1b.2	What degree of changeability does the product offer?
Q 1b.3	What degree of testability does the product offer?
Q 1b.4	What degree of technology concentration does the product offer?
Q 1b.5	What degree of stability does the product offer?
Sub-goal 1c: *Analyse the software product with the aim of evaluating it with respect to reliability from the software engineer's point of view*	
Q 1c.1	What degree of robustness does the product offer?
Q 1c.2	What degree of recoverability does the product offer?
Sub-goal 1d: *Analyse the software product with the aim of evaluating it with respect to functionality from the software engineer's point of view*	
Q 1d.1	What degree of functional adequacy does the product offer?
Q 1d.2	What degree of interoperability does the product offer?
Q 1d.3	What degree of functional accuracy does the product offer?
Sub-goal 1e: *Analyse the software product with the aim of evaluating it with respect to usability from the user's point of view*	
Q 1e.1	What degree of pleasantness does the product offer?
Q 1e.2	What degree of operability does the product offer?
Q 1e.3	What degree of understandability does the product offer?
Q 1e.4	What degree of learnability does the product offer?
Sub-goal 1f: *Analyse the software product with the aim of evaluating it with respect to efficiency from the software engineer's point of view*	
Q 1f.1	How the product is characterised in terms of time behaviour?
Q 1f.2	How the product is characterised in terms of resources utilisation?

Figure 4.3.: Questions in the EFFORT measurement framework pertaining to product quality. Reproduced from [57]

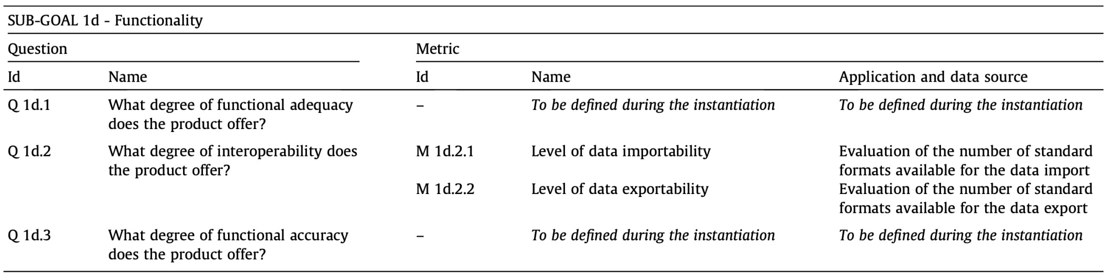
As can be seen in figure 4.4 and 4.5, some goals can't be completely defined in a generic manner. Rather they need to be completed during the instantiation process (see subsection 4.2).

Individual metrics can differ vastly. Whilst some may have a large continuous scale, others may have unique enumerated possible values. Therefore, each measurement is mapped to a discrete score ranging from 1 to 5, where 1 is interpreted as inadequate, 2 as poor, 3 as sufficient, 4 as good, and finally 5 as excellent. A higher value can also be interpreted either positively or negatively, depending on the related question. Therefore, a correct interpretation (positive or negative) needs to be defined. Lastly, since different questions have different importances within a goal, each question is assigned a relative relevance.



SUB-GOAL 1a – Portability				
Question		Metric		
Id	Name	Id	Name	Application and data source
Q 1a.1	What degree of adaptability does the product offer?	M 1a.1.1	Number of operating systems supported	Analysis of the documentation on the official web site
Q 1a.2	What degree of installability does the product offer?	M 1a.2.1	Time required for installation	Measurement of the required for installation of the product
		M 1a.2.2	Availability of the installation manual	Analysis of the documentation on the official web site
		M 1a.2.3	Automation level and use of installation scripts	Evaluation of the percentage of steps that can be automated during the installation
		M 1a.2.4	Dependence on third-party components	Evaluation during the product installation
		M 1a.2.5	Nominal length of the installation procedures	Analysis of the installation manual
		M 1a.2.6	Number of configuration files	Analysis of the documentation on the official web site
		M 1a.2.7	Availability of default options	Evaluation during the product installation
		M 1a.2.8	Internationalisation of the manual	Analysis of the documentation on the official web site
		M 1a.2.9	Number of unforeseen issues	Evaluation during the product installation
		M 1a.2.10	Degree of knowledge of the required operating environment	Evaluation during the product installation
		M 1a.2.11	Efficacy of the guide	Evaluation during the product installation
Q 1a.3	*What degree of replaceability does the product offer?*	–	*To be defined during the instantiation*	*To be defined during the instantiation*
Q 1a.4	*What degree of coexistence does the product offer?*	–	*To be defined during the instantiation*	*To be defined during the instantiation*

Figure 4.4.: Questions and metrics in the EFFORT measurement framework pertaining to sub-goal 1a. Reproduced from [57]



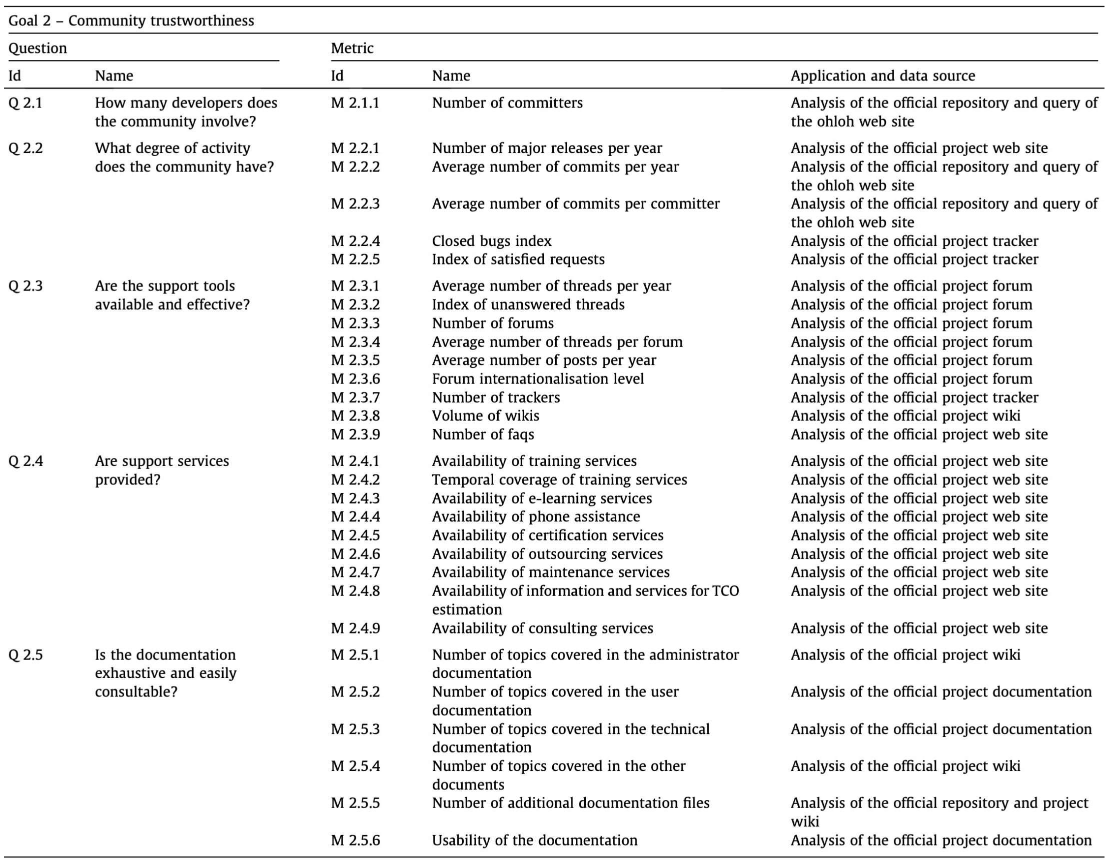
SUB-GOAL 1d - Functionality				
Question		Metric		
Id	Name	Id	Name	Application and data source
Q 1d.1	What degree of functional adequacy does the product offer?	–	*To be defined during the instantiation*	*To be defined during the instantiation*
Q 1d.2	What degree of interoperability does the product offer?	M 1d.2.1	Level of data importability	Evaluation of the number of standard formats available for the data import
		M 1d.2.2	Level of data exportability	Evaluation of the number of standard formats available for the data export
Q 1d.3	What degree of functional accuracy does the product offer?	–	*To be defined during the instantiation*	*To be defined during the instantiation*

Figure 4.5.: Questions and metrics in the EFFORT measurement framework pertaining Sub-goal 1d. Reproduced from [57]

Consequently, the quality of each goal can be calculated as follows:

$$q(g) = \frac{\Sigma_{q \in Q_q} r_q * m(q)}{\Sigma_{q \in Q_q} r_q}$$

where r_q is the relevance associated with the question q, Q_g is the set of questions related to the Goal g, and $m(q)$ is the aggregation function of the metrics of the question q. Essentially, the quality of a goal is defined by the weighted average of its questions' scores.



Goal 2 – Community trustworthiness				
Question		Metric		
Id	Name	Id	Name	Application and data source
Q 2.1	How many developers does the community involve?	M 2.1.1	Number of committers	Analysis of the official repository and query of the ohloh web site
Q 2.2	What degree of activity does the community have?	M 2.2.1	Number of major releases per year	Analysis of the official project web site
		M 2.2.2	Average number of commits per year	Analysis of the official repository and query of the ohloh web site
		M 2.2.3	Average number of commits per committer	Analysis of the official repository and query of the ohloh web site
		M 2.2.4	Closed bugs index	Analysis of the official project tracker
		M 2.2.5	Index of satisfied requests	Analysis of the official project tracker
Q 2.3	Are the support tools available and effective?	M 2.3.1	Average number of threads per year	Analysis of the official project forum
		M 2.3.2	Index of unanswered threads	Analysis of the official project forum
		M 2.3.3	Number of forums	Analysis of the official project forum
		M 2.3.4	Average number of threads per forum	Analysis of the official project forum
		M 2.3.5	Average number of posts per year	Analysis of the official project forum
		M 2.3.6	Forum internationalisation level	Analysis of the official project forum
		M 2.3.7	Number of trackers	Analysis of the official project tracker
		M 2.3.8	Volume of wikis	Analysis of the official project wiki
		M 2.3.9	Number of faqs	Analysis of the official project web site
Q 2.4	Are support services provided?	M 2.4.1	Availability of training services	Analysis of the official project web site
		M 2.4.2	Temporal coverage of training services	Analysis of the official project web site
		M 2.4.3	Availability of e-learning services	Analysis of the official project web site
		M 2.4.4	Availability of phone assistance	Analysis of the official project web site
		M 2.4.5	Availability of certification services	Analysis of the official project web site
		M 2.4.6	Availability of outsourcing services	Analysis of the official project web site
		M 2.4.7	Availability of maintenance services	Analysis of the official project web site
		M 2.4.8	Availability of information and services for TCO estimation	Analysis of the official project web site
		M 2.4.9	Availability of consulting services	Analysis of the official project web site
Q 2.5	Is the documentation exhaustive and easily consultable?	M 2.5.1	Number of topics covered in the administrator documentation	Analysis of the official project wiki
		M 2.5.2	Number of topics covered in the user documentation	Analysis of the official project documentation
		M 2.5.3	Number of topics covered in the technical documentation	Analysis of the official project documentation
		M 2.5.4	Number of topics covered in the other documents	Analysis of the official project wiki
		M 2.5.5	Number of additional documentation files	Analysis of the official repository and project wiki
		M 2.5.6	Usability of the documentation	Analysis of the official project documentation

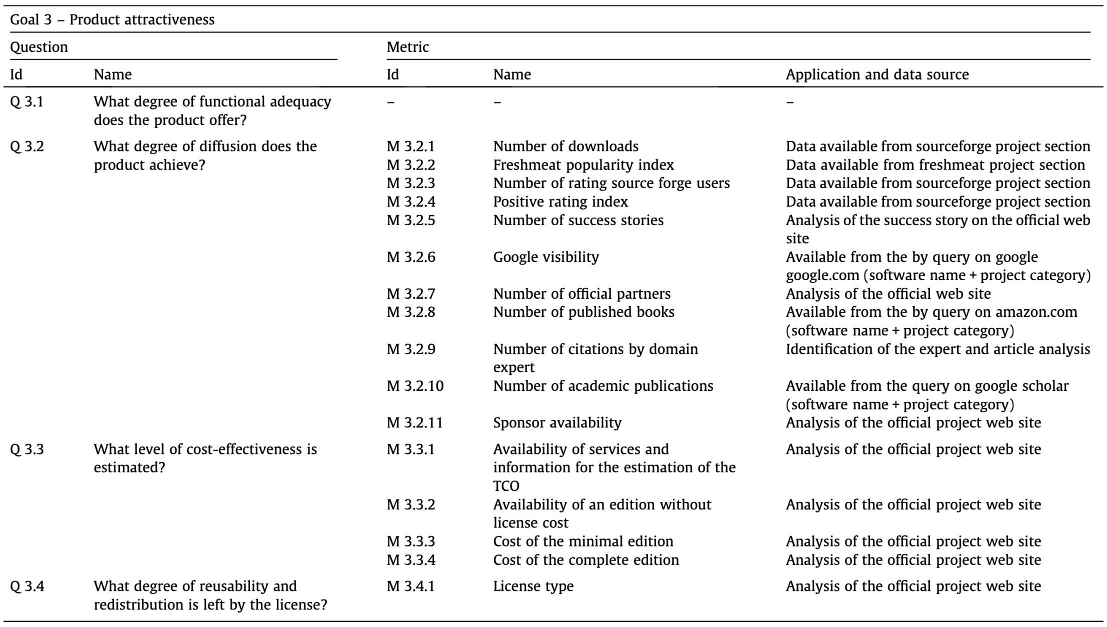
Figure 4.6.: Questions in the EFFORT measurement framework pertaining to community trustworthiness. Reproduced from [57]

The aggregation function of the metrics of a question q is defined as follow:

$$m(q) = \frac{\{\Sigma_{id \in M_q} i(id) * v(id) + [1 - i(id)] * v(id) mod 6]\}}{|M_q|}$$

where $v(id)$ is the value of the metric id and $i(id)$ is the interpretation of the metric with respect to the question q and has the value of 0 if the metric has a negative interpretation and 1 otherwise. M_q is the set of metrics related to a question q. Simply put, the aggregation function is the average score obtained by its metrics, whereas if the metric is negatively interpreted, its complement is taken.

In this work, $mod6$ is interpreted as the metric's complement within the given metrics range $[1 - 5]$ and not the modulo, since using the modulo would simply return the same value, as displayed in figure 4.8.



Goal 3 – Product attractiveness				
Question		Metric		
Id	Name	Id	Name	Application and data source
Q 3.1	What degree of functional adequacy does the product offer?	–	–	–
Q 3.2	What degree of diffusion does the product achieve?	M 3.2.1	Number of downloads	Data available from sourceforge project section
		M 3.2.2	Freshmeat popularity index	Data available from freshmeat project section
		M 3.2.3	Number of rating source forge users	Data available from sourceforge project section
		M 3.2.4	Positive rating index	Data available from sourceforge project section
		M 3.2.5	Number of success stories	Analysis of the success story on the official web site
		M 3.2.6	Google visibility	Available from the by query on google google.com (software name + project category)
		M 3.2.7	Number of official partners	Analysis of the official web site
		M 3.2.8	Number of published books	Available from the by query on amazon.com (software name + project category)
		M 3.2.9	Number of citations by domain expert	Identification of the expert and article analysis
		M 3.2.10	Number of academic publications	Available from the query on google scholar (software name + project category)
		M 3.2.11	Sponsor availability	Analysis of the official project web site
Q 3.3	What level of cost-effectiveness is estimated?	M 3.3.1	Availability of services and information for the estimation of the TCO	Analysis of the official project web site
		M 3.3.2	Availability of an edition without license cost	Analysis of the official project web site
		M 3.3.3	Cost of the minimal edition	Analysis of the official project web site
		M 3.3.4	Cost of the complete edition	Analysis of the official project web site
Q 3.4	What degree of reusability and redistribution is left by the license?	M 3.4.1	License type	Analysis of the official project web site

Figure 4.7.: Questions in the EFFORT measurement framework pertaining to product attractiveness. Reproduced from [57]

Interpretation	Modulo (`n%6`)	Metric complement (`6-n`)
1	1	5
2	2	4
3	3	3
4	4	2
5	5	1

Figure 4.8.: Interpretation of *mod*6 as modulo and complement

Up until now, the presented goals, questions and metrics and their relevance were based mainly on an Open Source Software point of view. As previously stated, the EFFORT framework can be customized to suit the context of its application. In the following chapter, the instantiation of EFFORT is presented.

4.2. Instantiation of EFFORT

The customization of EFFORT, i.e. its binding to a concrete use case, is achieved through three steps [57]. First, the application domain should be analyzed in order to gain thorough context specific knowledge and understand the specific requirements. This has been done in chapter 2 of this work. Next, the collected information of the previous step is used to verify the validity of the questions of the EFFORT baseline version. In the third step, additional questions and metrics can be added to better match the application domain [57].

Following, the second and third step of the customization of EFFORT are presented. Although the third step can be split between integration tasks, where metrics are added to the baseline versions and extension tasks, where goals are extended by adding questions, these two steps are presented at once by iterating over the goals which contributes to an improved comprehensibility as there is no need to switch back and forth between different topics. The resulting definition of questions and metrics including interpretation of the metrics, their mapping from a raw value to the given scale of $[1-5]$, and the questions' relevance are to be found in the appendix A.

Upon analyzing the application context along with the baseline EFFORT goals, questions and metrics, it could be established that the proposed questions are mostly sufficient. The only question that has been added is "How maintainable is the application code using the product?"(1b.6) in the goal Maintainability (1b). Moreover, only a few questions have been removed due to their inapplicability in the given domain. Within the first sub-Goal 'Portability' (1a), the question related to 'Coexistence' (1a.4) has been removed due to its irrelevance. Within the question (1a.1) regarding Adaptability, the metric 'Number of operating systems supported' has been replaced by 'Support for Function and Class Components'(1a.1.2) , and 'Adoption of React Suspense'(1a.1.3), which is an upcoming React feature [46]. Since the libraries can be easily installed with a package manager such as npm [26], multiple metrics of the question related to 'Installability' (1a.2) have been left out. Moreover, the 'Number of configuration files' (1a.2.6) has been replaced by the 'Minimum number of files added and changed for minimum functionality'. Additionally, the 'Availability of default options' (1a.2.7) has been extended to include their rationality. In order to measure 'Replaceability', the metric 'Existence of other libraries with the same functionality and a similar API' (1a.3.1) has been introduced.

Within the ‘Maintainability goal’ (1b.2), ‘Changeability’ was considered irrelevant, since libraries cannot be changed by definition. Furthermore, the question regarding ‘Technology Concentration’ is excluded due to its ambiguity. In order to measure the ‘Analyzability’, multiple metrics have been added among others ‘Availability and quality of developer tools’. Besides, ‘Stability’ is measured by the ‘Existence and degree of Breaking API changes’ and ‘Testability’ is assessed by the ‘Possibility to test business logic without React’. The question ‘How maintainable is the application code using the software product?’ (1b.6) is introduced and it consists of the metrics ‘Code duplication’, ‘Cyclomatic complexity’ [72] and ‘Cognitive complexity’[60]. Cognitive complexity is a measurement, that attempts to “reflect the relative difficulty of understanding the source code” [60]. While formulating these metrics, it was assumed, that a more suitable library would result in simpler and less source code in general.

The goal ‘Reliability’ that includes questions about ‘Robustness’ and ‘Recoverability’, is considered irrelevant in this use-case. The sub-goal ‘Functionality’ is based on the problems presented in section. Thereby, two metrics are attributed to the question ‘Functional adequacy’, namely ‘The possibility to share state without prop drilling’ and ‘The possibility to have derived state’. The metrics of ‘Interoperability’ have been adjusted as well. Instead of measuring the level of ‘Data importability’(1d.2.1) and ‘Data exportability’(1d.2.2), the ‘Availability of community plugins’ is adopted as a unique metric. In addition, the metrics ‘Lines of code’, ‘Statement count’, and ‘Function count’ are introduced in order to measure the ‘Functional accuracy’. It is worth mentioning, that these metrics refer to the source code of the implementation of the use case formulated in the section 2.3 while using the libraries, and not the source code of the libraries itself. Once again, less code is interpreted as a sign of higher suitability.

Regarding the sub-goal ‘Usability’, the metric ‘Availability and quality of developer tools’ is used to measure the degree of ‘Operability’ whereas the ‘Learnability’ is measured by looking at the degree of ‘Usability of the documentation’. ‘Usability of the documentation’ takes into account whether installation instructions, a project setup, a hello world example, and advanced guides are provided.

In order to measure the ‘Efficiency’, the two questions of ‘Time behavior’ and ‘Resources utilization’ are extended. The metrics of ‘Time behavior’ measure the loading speed of a webpage and include the ‘First Contentful Paint’ [15], the ‘Speed Index’ [41], the ‘Largest Contentful Paint’ [23], the ‘Time to Interactive’ [48], the ‘Total Blocking Time’ [50], and the ‘Cumulative Layout Shift’ [8]. For measuring the ‘Resource utilization’, the problem of unnecessary re-renderings (P4) presented above is used.

In order to evaluate the ‘Community trustworthiness’, multiple metrics are changed. First, the metric ‘Number of major releases per year’ is removed, since it can be differently interpreted. Nevertheless, a few metrics have been adjusted and others are removed namely ‘Number of forums’, because the major forum used is Stack Overflow.

Other metrics proposed by the EFFORT baseline version have been left out due to their inapplicability to libraries, among others ‘Availability of outsourcing services’, ‘Availability of maintenance services’, ‘Availability of information and services for TCO estimation’, and ‘Temporal coverage of training services’. Also, since the differentiation between administrator, user, technical, and other documentations is not applicable in the context of libraries, metrics related to the mentioned documentations are removed.

When it comes to ‘Product attractiveness’, two factors play a role. First, the libraries are free of cost and second, the libraries’ source code is available at GitHub. Additionally, the number of downloads has been adjusted to weekly downloads to match the stats available from “npm.com” [26].

5. Discussion

In this chapter, the results of applying EFFORT are presented. Mainly the resulting score of each approach is analyzed and more general conclusions are drawn upon the suitability of each approach. Afterwards, the hypotheses formulated in chapter section 1.1 are verified and a reflection on the research process is presented.

5.1. Findings

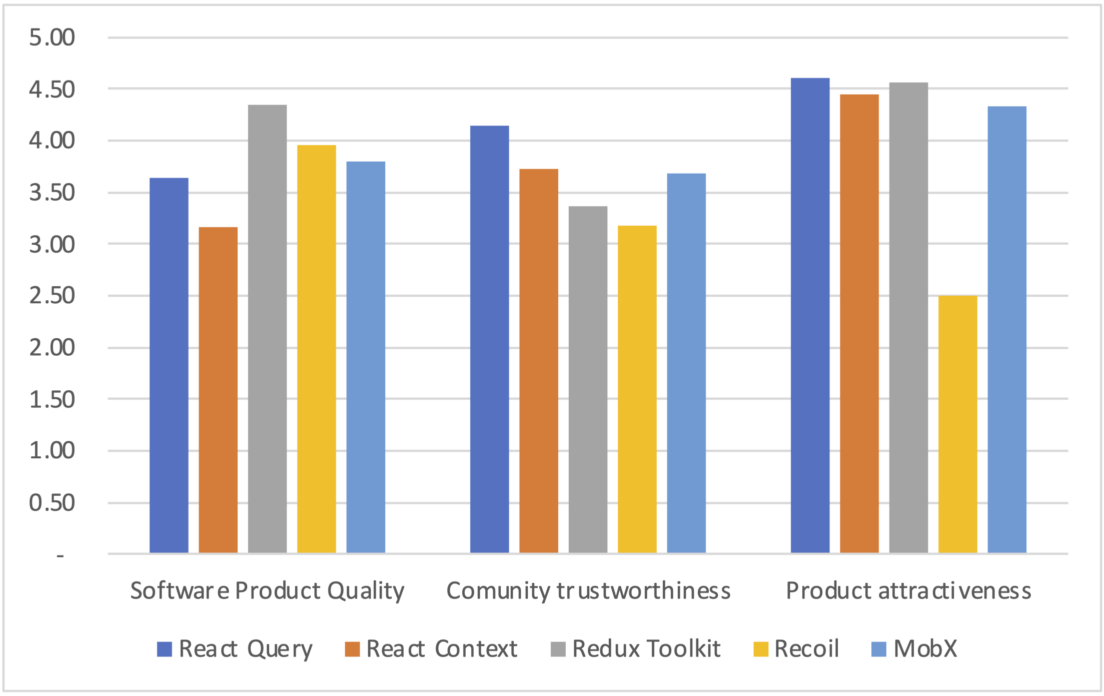


Figure 5.1.: Scores achieved by the SMLs by applying EFFORT

The results of the application of EFFORT on React Context, React Query, Redux Toolkit, Recoil and MobX is shown in figure 5.1. A general overview on the results shows that the approaches display varying levels of 'Software Product Quality', whereby React Context achieved the lowest score. The 'Community trustworthiness' has also varying results. Finally the 'Product attractiveness' is high in all the approaches with the exception of Recoil which achieved a remarkably lower score. The scores within each goal along with the influencing factors are explained further in the following. The detailed scores achieved by each metric can be found in the appendix B.

5.1.1. Software Product Quality

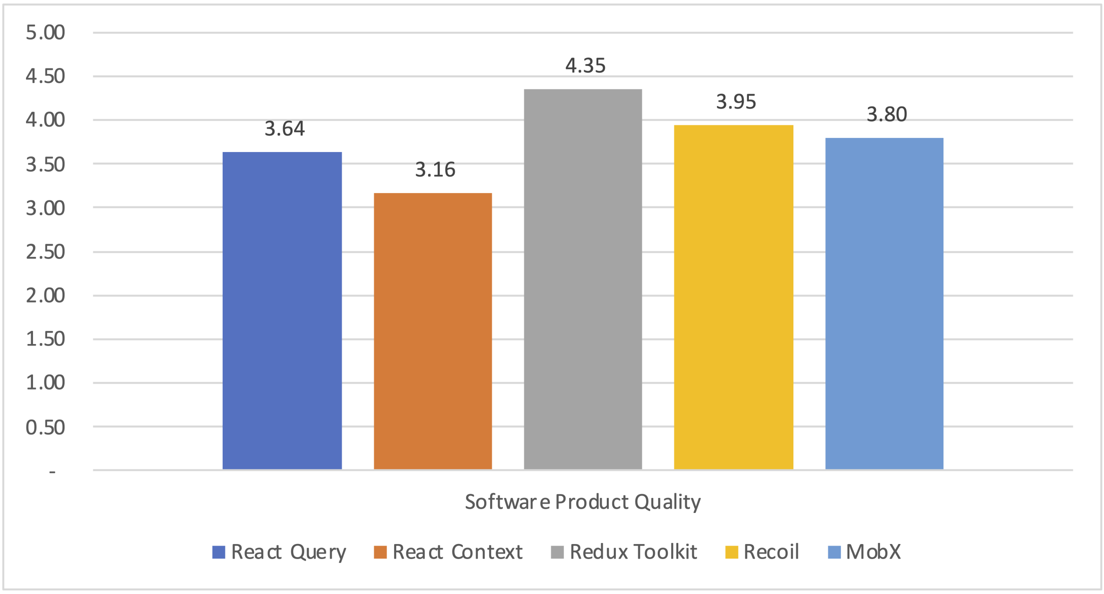


Figure 5.2.: Comparison of the software product quality of the five evaluated SMLs

Starting with the software product quality, the figure 5.2 presents the results of the analyzed libraries where React Context exhibits the lowest score. This is unsurprising, since React Context is a mechanism whose only purpose is to solve the problem of prop drilling [3]. On the other hand, the first place is taken by Redux Toolkit. The high score is due to the rich feature set of the library as well as its high degree of 'Testability'. Since Redux Toolkit is an opinionated library that bases on Redux, it simplifies the definition of action creators and async actions while retaining a high level of testability. Additionally, its sub-package RTK Query covers the majority of React Query's functionality. The second place belongs to Recoil and is followed closely by MobX in the third position. Both of these libraries benefit from efficient re-renderings.

Moreover, testing in both libraries can be done easily. Although it must be mentioned that the use of the Object-Oriented Programming (OOP) by MobX simplifies the writing of code and the testing. In contrast, Recoil solves common OOP problems with its atomic state structure approach, most notably when having a complex dependencies in the application state. React Query comes second to last. Although it fulfills the requirements of managing server state, handling client state exceeds the purpose of this library. Implementing client side state requires using a global variable and invalidating the cache after each change. This process is error prone and complicates the creation of tests.

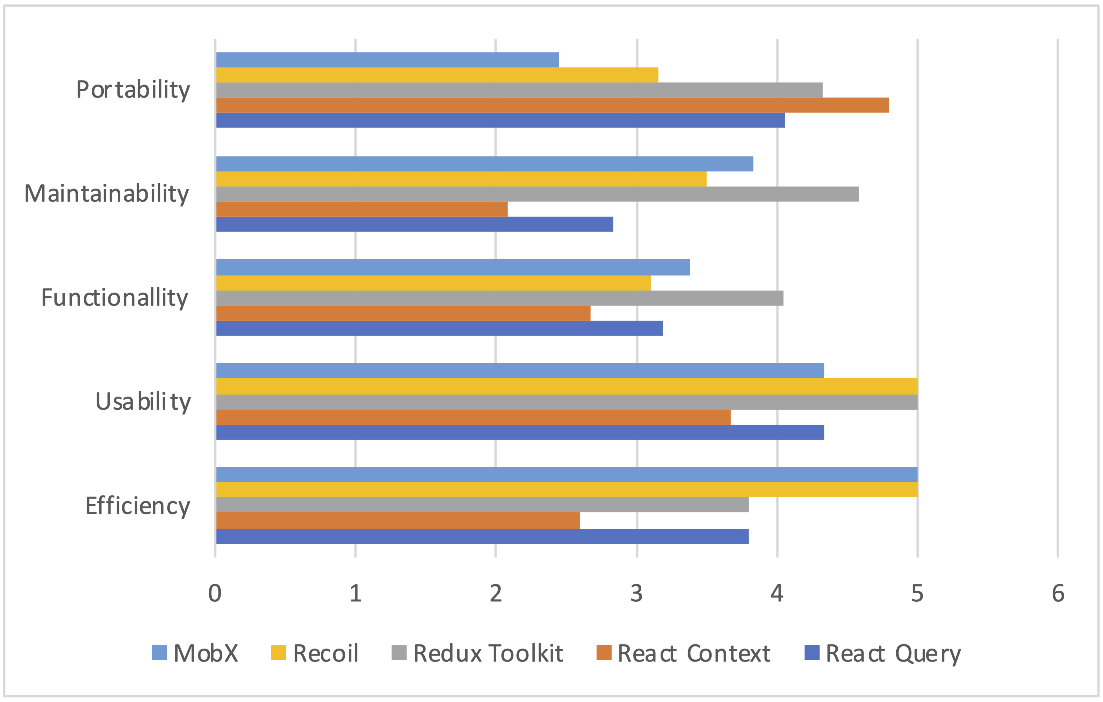


Figure 5.3.: Detailed results pertaining to the software product quality of the five evaluated SMLs

By analyzing the results of each subgoal of ‘Software Product Quality’ displayed in Figure 5.3, a more elaborate understanding of the differences between the libraries can be reached.

To begin with, the libraries show different degrees of ‘Portability’. React Context’s high degree of ‘Portability’ comes from its support for both function and class components and its support for React Suspense by definition, since it is part of the React library.

Next come Redux Toolkit and React Query. While React Query can be easily integrated into an existing application with minimal additional code and file changes, Redux Toolkit supports both class and function components (which is lacking in React Query and Recoil) and the support for React Suspense is planned [45]. MobX occupies the last place mainly due to its irreplaceability and lack of default options. The replaceability was measured by the number of libraries available that offer an equivalent functionality and a similar API. While similar libraries exist for the rest, none could be found for MobX. Additionally, MobX lacks a set of meaningful defaults, offering the developer more flexibility and control, yet imposing the need to write common code such as making API calls manually.

When it comes to maintainability, Redux Toolkit takes the lead. This is directly caused by the high degree of testability and the sophisticated developer tools provided [37]. The testability of the code can be owed to the clearly defined architecture that allows to run tests on different layers. More importantly the use of pure functions in the reducer and the selectors make it possible to write unit tests while retaining the possibility of writing integration tests that include the UI layer [54]. The developer tools provided for Redux Toolkit are highly practical. Their key features include the possibility to browse the history of the dispatched actions along with their effects on the state, revert to a given state and the possibility to export and import the complete state history [37]. Next come React Query and Recoil, that both include developer tools, yet with less functionality. React Query surpasses Recoil in Stability. This is comprehensible, since at the time of writing, Recoil is still in an experimental state [13].

Although there have been few breaking API changed introduced in React Query, they included not only a migration guide, but a script that applies the needed changes to the code using the library [24]. On the last place comes React Query mainly because of the difficulty of testing it. While calculating the 'Maintainability, other metrics were taken into account such as 'Code duplication', 'Cyclomatic complexity' and 'Cognitive complexity', which are directly linked to the source code. The libraries performed similarly in all of these measures and no conclusions could be made in this regard.

Regarding 'Functionality', all the libraries fulfill the requirements of sharing the state without prop drilling and having a derived state. Redux Toolkit receives the highest score because of its high degree of 'Interoperability' stemming from the large amount of plugins available, an area which is lacked by others. It is followed by MobX, which displayed a high degree of ‘Functional Adequacy’, due to the low count of lines, statement and functions needed to implement the same functionality.

Redux Toolkit and Recoil show the highest level of usability. Thanks to their documentations that include advanced guides and are translated to multiple languages, the libraries excel in ‘Learnability’. Their ease of ‘Operability’ comes from having high quality developer tools.

Lastly, Efficiency was strongly influenced by ‘Effectiveness of the re-rendering of the components’, since all the libraries achieved similar high score related the loading page speed. That is to say that the choice of these libraries does not seem to have any effect on the loading speed of the application whatsoever. Within the ‘Efficiency of re-rendering’ , Recoil and MobX have shown efficient re-renderings, Redux Toolkit and React Query suboptimal re-renderings and React Context redundant re-renderings (see section 2.2 for a definition of the terms).

5.1.2. Community Trustworthiness

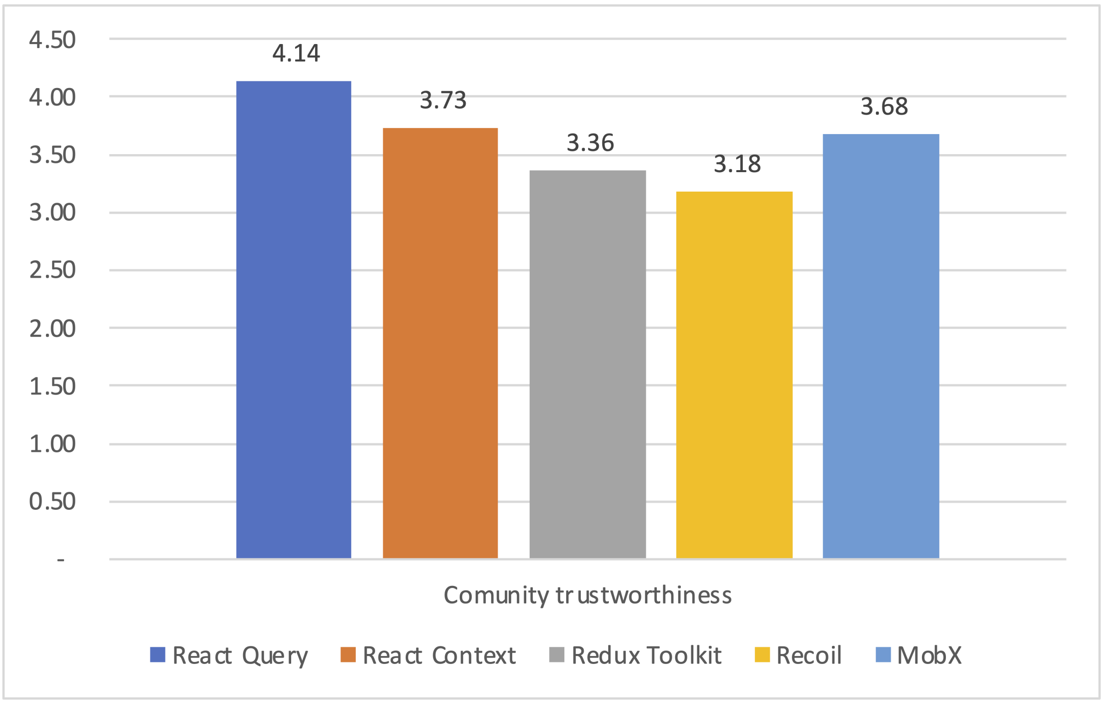


Figure 5.4.: Comparison of the community trustworthiness of the five evaluated SMLs

The score achieved by the libraries regarding to the second goal, 'Community Trustworthiness' are shown on figure 5.4, where a gradual distribution can be seen. React Query has shown the highest degree of trustworthiness thanks to the availability of support and consulting services for sponsors as well as e-learning materials. This is complemented by the high number of its contributors and an extensive documentation. On the second and third place come React Context and MobX with neighboring scores of 3.75 and 3.68 respectively. Redux Toolkit and Recoil, which occupy the penultimate and last places have a relatively low number of involved developers and low degree of activity of their communities. This can be explained in the case of MobX by the high index of unanswered threads on Stack Overflow and in Recoil by the low number of threads per year on Stack Overflow.

5.1.3. Product Attractiveness

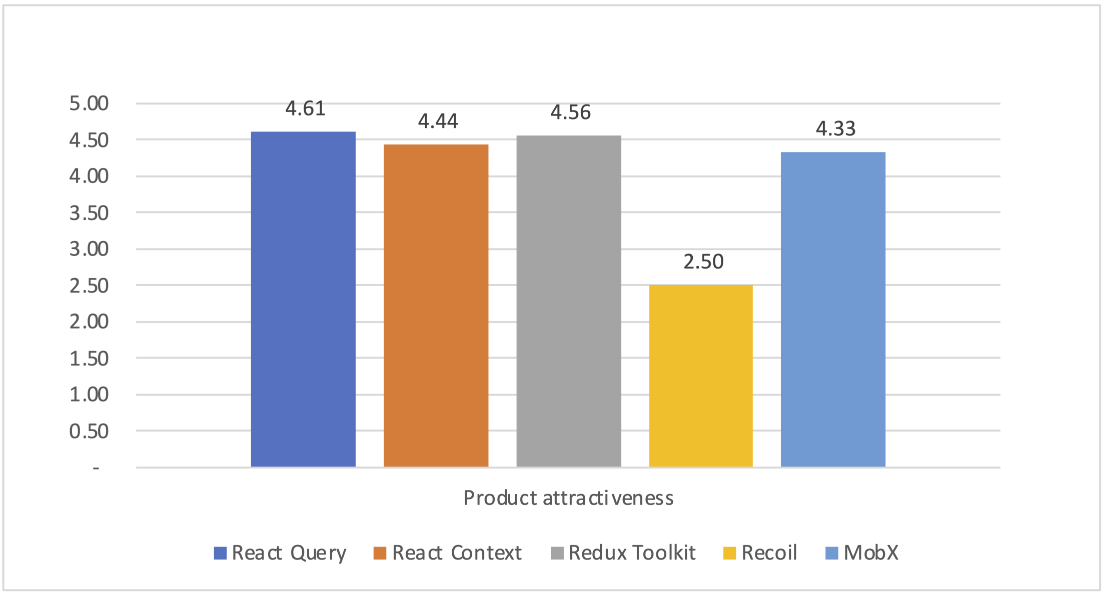


Figure 5.5.: Comparison of the product attractiveness of the five evaluated SMLs

The results of the Goal 'Product attractiveness' depicted in the figure 5.5 reveal that with the exception of Recoil, the libraries perform at a similar level. Looking into the questions and metrics that lead to the score, it can be established that Recoil's low number of weekly downloads is the main driver. Additionally, the low amount of threads in Stack Overflow combined with a relatively high number of unanswered questions contributed in the low score. Interestingly, the project has more stars and forks on GitHub than Redux Toolkit. This may be explained by the fact that Recoil is still in an experimental state in the time of this study and that it is maintained by the same team behind React [32].

5.2. Suitability of the Approaches

After the in depth comparison of the state management libraries has been conducted, the suitability of each library is to be discussed. Thereby it is important to note that the choice of a technology will most likely involve other business related factors.

Drawing from the previous results, it has become clear that React Context is perfectly suitable to handle global state. It is therefore a good fit for states that are used globally and entail little logic like the application theme, the currently authenticated user or the selected locale. Since such a global state usually requires a change in the whole UI, the re-rendering optimization provided by other approaches are irrelevant.

React Query has shown that although it can be stretched to manage client state, it is best suitable for applications whose state lies in a backend server and needs to be synchronized with the client. Most notably when server state is shared in multiple parts of the UI, the built in cache can be utilized to reduce the API calls. It is also best suitable for improving the user experience by the ease of implementing optimistic updates. This makes React Query a good solution for building a new user interface for an existing backend server.

When client side state needs to be shared as well, Redux Toolkit can be an appropriate choice. Redux Toolkit provides all the tools needed for creating a software product of high quality, mainly a built in query client, a concise API, and a high level of testability.

MobX is a minimalistic yet effective library that can be easily learned, thanks to its use of the OOP paradigm. Since it supports class components, it can be a good suit for a slow introduction in a legacy React application that does not use function components. Moreover, thanks to its simple integration with react, it can be a good solution to solve one of the presented problems with little changes in the application.

Lastly, Recoil's strength in defining atoms and explicitly building a dependency tree to represent an application state hints to its suitability for complex applications where performance is a requirement, such as a graphics editor.

5.3. Hypotheses Verification

After defining the suitability of the application of the libraries, the hypotheses presented in chapter 1.1 can be verified on the basis of the performed analysis.

This study confirms the first hypothesis: "React Context is best suited for data which is rarely changed, such as the authenticated user or the current UI theme." As previously stated, React Context solves the problem of prop drilling and therefore is suitable for the above mentioned use case.

The second hypothesis: "State Management Libraries are suitable for complex User Interfaces (UIs) that don't communicate much with a backend server." can only be partially confirmed when the term State Management Libraries is interpreted as MobX, Recoil and Redux Toolkit. As explained in the previous chapter, these libraries do support complex client states, but there is no restriction when it comes to communicating with a backend server since API calls can be made within all of them. Additionally Redux Toolkit also offers the functionality to communicate with a backend server through its built-in query client [38].

The third hypothesis: "Query approaches are better suited for UIs, that need to interact with a backend server." is also confirmed. As previously presented, React Query fulfills the requirements of communicating with a backend server, by managing API calls, cache, loading states and optimistic updates.

5.4. Reflection

Before concluding this thesis, it is important to reflect on the research process. This includes remarks and takeaways that emerged during the practical application of the research design. These are related to the usage of metrics and the application of the framework EFFORT.

5.4.1. Usage of Metrics

The use of metrics in the scope of this thesis was implicitly assumed, yet their practical usage comes with various challenges. The first challenge lies in the interpretation of the metrics. For example 'The average threads per year in Stack Overflow' can be interpreted differently. A low value could mean a low adoption of the product or that it has reached a stable state and its usage is clear. A better metric could analyze the evolution of the tread count in Stack Overflow to verify the stability and maturity of the product. A similar effect can be seen in the metric 'The number of major releases per year'. Other metrics were arguably less practical. These include "the Average number of commits per committer", whose goal is to determine the level of commitment the maintainers have. The result could be easily diluted: A lower number could be justified by the core maintainers actively promoting maintaining the project. An alternative metric that would fulfill the same goal could be 'Count of authors who have 50 commits or more'. Yet, new metrics need to be tested in order to avoid similar problems and edge cases.

Another challenge emerges from the fact that knowing about the usage of metrics had a clear impact on the produced code. For instance, being aware of the metric "code duplication", has led to a perfect measurement of 0 in the applications of all the libraries. A more practical indicator could be for example 'The tendency of developers to write repetitive code when using the library', which could be measured by analyzing open source projects that use the library. This leads to a broader topic concerning the effect of metrics on the developers. Or as Goodhart's law (named after British economist Charles Goodhart) puts it: "When a measure becomes a target, it ceases to be a good measure" [71]. The usage of quantitative metrics has also shown little usefulness. Namely the application of the metric "Cognitive complexity" in the context of the thesis experiments has proven to be incomplete for the reason that it does not consider the knowledge of third party libraries required to understand the code. Ideally this metric would include the degree to which the library is incorporated into the source code. Other qualitative metrics has yielded very similar results, such as the 'Count of Lines of Code', 'Count of functions' and 'Count of statements'. Moreover, multiple metrics are defined specifically for OOP languages. This may explain the limited choice of static code analysis tools available for the used programming language (TypeScript) in comparison to other programming languages, such as Java. It is to mention, that although the used language does support the OOP paradigms such as class definitions, interfaces and inheritance, it is seldom used in that way. Generally, there seems to be a gap between metrics proposed in scientific papers, which have a sound and mathematical basis and the one that can be measured in practice.

All in all, the results point out that qualitative metrics play a bigger role when choosing a software product. This introduces a level of subjectivity into the assessment process, which is understandable since the choice of a software product is influenced by other factors and needs to fulfill greater requirements than the technical aspects. Luckily, the flexibility of EFFORT provided by the instantiation step allows for great customization to match user specific use cases.

5.4.2. EFFORT

The possibility to edit and add goals, questions, and metrics within EFFORT is an advantage over many of the other approaches. The practicability of the framework comes from the definition of metric mappings that brings the values to the same range of 1-5, from the possibility to define a positive or negative metric, the relevance of the questions, and finally from having a formula to calculate the performance within goal. Yet there have been a few ambiguities during the application. For one, the mapping values defined as (1 = inadequate, 2 = poor, 3 = sufficient, 4 = good, and 5 = excellent) can be misleading, since each metric can have either a positive or negative interpretation. A scale from lowest to highest would be more clear for the user.

Moreover, the framework doesn't explicitly allow the removal of the baseline version questions, goals or metrics that are not applicable to the application domain. This was deduced from the example application on ERP systems where the sub-goal "Security" was left out [57].

Other ambivalences were experienced, such as the term "Technology concentration" used under the maintainability goal, as well as the absence of the source of the metrics and how to measure them, and the differentiation between question markers in the FlOSS context and in the specific context [57]. Additionally no instructions are given on creating the metric mappings. In the carried out experiments, the quantitative metric mappings were created after the raw metric results were collected. Although the metrics may be general, this results in a use case specific mapping. Plus, the mapping of the metrics as a whole depends greatly on the selected software products to be compared.

All things considered, the adoption of the framework EFFORT for the comparison has yielded satisfying results. Its high level of customizability has allowed the comparison of libraries, although it was previously only used to compare standalone applications.

6. Outro

This work has given an in depth analysis of state management libraries and provided general conclusions about the suitability of each approach. By using the EFFORT framework, a found comparison based on software quality characteristics could be made. Since only a selection of libraries was analyzed, future work could include other libraries by using the proposed instantiation of EFFORT, especially to compare state management libraries within the same category. Related topics, which were out of the scope of this study, could be the subject of further research projects. Among others, further research could address ambiguities related to the interpretation of metrics by extending the existing qualitative and quantitative metrics for open source software and providing standard metric values in order to facilitate the creation of metric mappings.

Appendices

A. EFFORT Instantiation

A.1. Questions Definition

Table A.1 displays the questions defined during the instantiation of EFFORT along with their relevance within their associated goals. $*$ means the question has been added, $^{\square}$ means the question has not been changed, and $^{\times}$ means the question has been removed.

Table A.1.: Questions defined during the instantiation of EFFORT.

Id	Question	Relevance
1a.1	What degree of Adaptability does the product offer?$^{\square}$	2
1a.2	What degree of Installability does the product offer?$^{\square}$	1
1a.3	What degree of Replaceability does the product offer?$^{\square}$	2
1a.4	What degree of Coexistence does the product offer?$^{\times}$	
1b.1	What degree of Analyzability does the product offer?$^{\square}$	2
1b.2	What degree of Changeability does the product offer?$^{\times}$	
1b.3	What degree of Testability does the product offer?$^{\square}$	2
1b.4	What degree of Technology concentration does the product offer?$^{\times}$	
1b.5	What degree of Stability does the product offer?$^{\square}$	2
1b.6	How maintainable is the application code using the product?*	2
1c.1	What degree of Rebustness does the product offer?$^{\times}$	
1c.2	What degree of Recoverability does the product offer?$^{\times}$	
1d.1	What degree of Functional adequacy does the product offer?$^{\square}$	3
1d.2	What degree of Interoperability does the product offer?$^{\square}$	2
1d.3	What degree of Functional accuracy does the product offer?$^{\square}$	2
1e.1	What degree of Pleasantness does the product offer?$^{\times}$	

Table A.1.: Questions defined during the instantiation of EFFORT.

Id	Question	Relevance
1e.2	What degree of Operability does the product offer?$^{\square}$	1
1e.3	What degree of Understandability does the product offer?$^{\times}$	
1e.4	What degree of Learnability does the product offer?$^{\square}$	2
1f.1	What degree of Time behavior does the product offer?$^{\square}$	2
1f.2	What degree of Resources utilization does the product offer?$^{\square}$	3
2.1	How many developers does the community involve?$^{\square}$	3
2.2	What degree of activity does the community have?$^{\square}$	2
2.3	Are the support tools available and effective?$^{\square}$	2
2.4	Are support services provided?$^{\square}$	1
2.5	Is the documentation exhaustive and easily consultable?$^{\square}$	3
3.1	What degree of functional adequacy does the product offer?$^{\square}$	3
3.2	What degree of diffusion does the product achieve?$^{\square}$	2
3.3	What level of cost-effectiveness is estimated?$^{\times}$	
3.4	What degree of reusability and redistribution is left by the license?$^{\square}$	1

A.2. Metrics Definition

Table A.2 displays the metrics defined during the instantiation of EFFORT, where $*$ means the metric has been added, $^{\Delta}$ means the metric has been changed, $^{\square}$ means the metric has not been changed, and $^{\times}$ means the metric has been removed. Additionally, negatively interpreted metrics are marked with $(-)$ and positive ones with $(+)$.

Table A.2.: Metrics defined during the instantiation of EFFORT.

Id	Metric	Metric mapping
1a.1.1	Number of operating systems supported$^{\times}$	
1a.1.2	Support for Function and Class Components*(+)	No=1 Yes=5

Table A.2.: Metrics defined during the instantiation of EFFORT.

Id	Metric	Metric mapping
1a.1.3	Adoption of React Suspense*(+)	Implementation available=5 Planned=3 No=1
1a.2.1	Time required for installation$^{\times}$	
1a.2.2	Availability of the installation manual$^{\square}$(+)	No=1 Yes=5
1a.2.3	Automation level and use of installation scripts$^{\times}$	
1a.2.4	Dependence on third-party components$^{\square}$(−)	0=1 1-5=2 5-10=3 10-15=4 15-20=5
1a.2.5	Nominal length of the installation procedures$^{\times}$	
1a.2.6	Minimum number of files added and changed for minimum functionality$^{\triangle}$(−)	One file=1 Two files = 3 Three and higher=5
1a.2.7	Availability and rationality of default options$^{\triangle}$(+)	No default options=1 Default options provided=3 Useful/rational default options provided=5
1a.2.8	Internationalisation of the manual$^{\square}$(+)	Multiple languages supported=5 English supported=3 Only non-english language supported=1
1a.2.9	Number of unforeseen issues$^{\square}$(−)	Zeo issues=1 Two/Three issues=3 More issues=5
1a.2.10	Degree of knowledge of the required operating environment$^{\square}$(−)	Little previous knowledge required=1 Medium knowledge required=3 Understanding of a paradigm required=5
1a.2.11	Efficacy of the guide$^{\square}$(+)	Installation guide non existing=1 Installation guide available=5
1a.3.1	Existence of other libraries with the same functionality and a similar API*(+)	No equivalent library available=1 $<$ 3 Similar libraries available=3 3+ similar libraries available=5

Table A.2.: Metrics defined during the instantiation of EFFORT.

Id	Metric	Metric mapping
1b.1.1	Availability and quality of developer tools*(+)	No=1 Yes=3 Yes, with advanced debugging features=5
1b.3.1	Possibility to test business logic without React*(+)	No=1 Yes=5
1b.5.1	Breaking API changes*(+)	Breaking API changes in major versions=1 Breaking API changes in major versions with migrations guide=3 Breaking API changes in major versions with migration guide and codemod=4 No breaking API changes introduced major update=5
1b.6.1	Code duplication*(−)	0=1 1-10=2 10-20=3 10-30=4 30+=5
1b.6.2	Cyclomatic complexity*(−)	0-92=1(Base 82) 92-102=2 102-112=3 112-122=4 122+=5
1b.6.3	Cognitive complexity*(−)	1-10=1 11-20=2 21-50=4 Over 50=5
1d.1.1	Possibility to share state without prop drilling*(+)	No=1 Yes=5
1d.1.2	Possibility to have derived state*(+)	No=1 Yes, through React=3 Yes=5
1d.2.1	Level of data importability$^{\times}$	
1d.2.2	Level of data exportability$^{\times}$	
1d.2.3	Availability of community plugins*(+)	No=1 Yes=5
1d.3.1	Lines of code*(−)	0-685=1 685-710=2 710-735=3 735-760=4 760+=5
1d.3.2	Statement count*(−)	0-114=1 114-124=2 124-144=3 134-144=4 Over 144=5
1d.3.3	Functions count*(−)	0-59=1 59-69=2 69-79=3 79-89=4 Over 89=5
	// non applicable to domain	
1e.2.1	Availability and quality of developer tools*(+) (same as 1b.1.1)	No=1 Yes=3 Yes, with advanced debugging features=5

Table A.2.: Metrics defined during the instantiation of EFFORT.

Id	Metric	Metric mapping
	// non applicable to domain	
-	Same metrics as in Question 2.5	-
1f.1.1	First Contentful Paint*(−)	0-1.8=1 1.8-3=3 Over 3=5
1f.1.2	Speed Index*(−)	0–3.4=1 3.4–5.8=3 Over 5.8=5
1f.1.3	Largest Contentful Paint*(−)	0–2.5=1 2.5–4=3 Over 4=5
1f.1.4	Time to Interactive*(−)	0–3.8=1 3.8–7.3=3 Over 7.3=5
1f.1.5	Total Blocking Time*(−)	0–200=1 200–600=3 Over 600=5
1f.1.6	Cumulative Layout Shift*(−)	0–0.1=1 0.1–0.25=3 Over 0.25=5
1f.2.1	Efficient re-renders*(+)	Redundant re-renders=1 Suboptimal re-renders=3 Optimal re-renders=5
2.1.1	Number of committers$^{\square}$(+)	0-100=1 100-200=2 200-300=3 300-400=4 Over 400=5
2.2.1	Number of major releases per year$^{\times}$	
2.2.2	Average number of commits per year$^{\square}$(+)	0=1 1-10=2 10-100=3 100-1000=4 Over 1000=5
2.2.3	Average number of commits per committer$^{\square}$(+)	0-2.5=1 2.5-5=2 5-7.5=3 7.5-10=4 Over 10=5
2.2.4	Closed issues on GitHub$^{\Delta}$(+)	0-250=1 250-500=2 500-750=3 750-1000=4 Over 1000=5
2.2.5	Index of merged pull requests on GitHub$^{\Delta}$(+)	0-250=1 250-500=2 500-750=3 750-1000=4 Over 1000=5
2.3.1	Average number of threads per year on Stack Overflow$^{\Delta}$(+)	0-200=1 200-400=2 400-600=3 600-800=4 Over 800=5
2.3.2	Index of unanswered threads on Stack Overflow$^{\Delta}$(−)	0-5=1 5-10=2 10-15=3 15-20=4 Over 20=5
2.3.3	Number of forums$^{\times}$	
2.3.4	Average number of threads per forum$^{\times}$	
2.3.5	Average number of posts per year$^{\times}$	
2.3.6	Forum internationalisation level$^{\times}$	
2.3.7	Number of trackers$^{\times}$	

Table A.2.: Metrics defined during the instantiation of EFFORT.

Id	Metric	Metric mapping
2.3.8	Usability of the documentation (same as 2.5.6)$^{\Delta}$(+)	Documentation provides no instructions=1 Documentation includes installation instructions=2 Documentation provides project setup=3 Documentation provides hello world=4 Documentation provides advanced examples and guides=5
2.3.9	Number of faqs in the documentation$^{\Delta}$(+)	No FAQ available=5 $<$ 10 Questions=3 11+ Questions=5
2.4.1	Availability of training services$^{\square}$(+)	No=1 Yes=5
2.4.2	Temporal coverage of training services$^{\times}$	
2.4.3	Availability of e-learning services$^{\times}$	
2.4.4	Availability of phone assistance$^{\times}$	
2.4.5	Availability of certification services$^{\square}$(+)	No=1 Yes=5
2.4.6	Availability of outsourcing services$^{\times}$	
2.4.7	Availability of maintenance services$^{\times}$	
2.4.8	Availability of information and services for TCO estimation$^{\times}$	
2.4.9	Availability of consulting services$^{\square}$(+)	No=1 Yes=5
2.5.1	Number of topics covered in the administrator documentation$^{\times}$	
2.5.2	Number of topics covered in the user documentation$^{\times}$	
2.5.3	Number of topics covered in the technical documentation$^{\times}$	
2.5.4	Number of topics covered in the other documents$^{\times}$	
2.5.5	Number of additional documentation files$^{\times}$	

Table A.2.: Metrics defined during the instantiation of EFFORT.

Id	Metric	Metric mapping
2.5.6	Usability of the documentation$^{\square}$(+)	Documentation provides no instructions=1 Documentation includes installation instructions=2 Documentation provides project setup=3 Documentation provides hello world=4 Documentation provides advanced examples and guides=5
3.2.1	Number of weekly downloads$^{\triangle}$(+)	0-250K=1 250K-500K=2 500K-750K=3 750K-1M=4 Over 1M=5
3.2.2	Number of stars on GitHub$^{\triangle}$(+)	0-50=1 50-500=2 500-5K=3 5K-50k=4 Over 50K=5
3.2.3	Number of forks on GitHub$^{\triangle}$(+)	0-250=1 250-500=2 500-750=3 750-1000=4 Over 1000=5
3.2.4	Positive rating index$^{\times}$	
3.2.5	Number of success stories$^{\times}$	
3.2.6	Google visibility$^{\times}$	
3.2.7	Number of official partners/sponsors$^{\triangle}$(+)	None=1 Many=5
3.2.8	Number of published books$^{\square}$(+)	None=1 Many=5
3.2.9	Number of citations by domain expert$^{\times}$	
3.2.10	State of academic publications$^{\triangle}$(+)	No academic publications found: 1 Academic publications done by the users of the software product: 3 Academic publications done by the authors of the software product: 5
3.2.11	Sponsor availability$^{\square}$(+)	No=1 Yes=5
3.3.1	Availability of services and information for the estimation of the TCO$^{\times}$	
3.3.2	Availability of an edition without license cost$^{\times}$	
3.3.3	Cost of the minimal edition$^{\times}$	

Table A.2.: Metrics defined during the instantiation of EFFORT.

Id	Metric	Metric mapping
3.3.4	Cost of the complete edition$^{\times}$	
3.4.1	License type$^{\square}$(+)	Commercial usage allowed=5 Commercial not usage allowed=1

B. Results

B.1. Raw Metrics Results

Table B.1.: Raw metrics results of the analysed State Management Libraries

Metric Id	React Query	React Context	Redux Toolkit	Recoil	MobX
1a.1.2	No	Yes	Yes	No	Yes
1a.1.3	Beta implementation available	Implementation available by definition	Planned	Implementation available	No
1a.2.2	Yes	Yes	Yes	Yes	Yes
1a.2.4	0	0	4	0	0
1a.2.6	2 (App.js for adding QueryProvider + usage in component)	2 (App.js for adding ContextProvider + usage in component)	4 (defining Store,adding Store-Provider in App.js, defining slice, usage in component))	2 (App.js for adding RecoilRoot + usage in component)	2 (store definition, usage in component)
1a.2.7	Useful/rational default options provided	No default options	Useful/rational default options provided	No default options	No default options
1a.2.8	English supported	Multiple languages supported	English supported	Multiple languages supported	Multiple languages supported

Table B.1.: Raw metrics results of the analysed State Management Libraries

Metric Id	React Query	React Context	Redux Toolkit	Recoil	MobX
1a.2.9	0	0	0	0	0
1a.2.10	Medium knowledge required	Medium knowledge required	Understanding of a paradigm required	Understanding of a paradigm required	Little previous knowledge required
1a.2.11	Installation guide available	Installation guide available	Installation guide available	Installation guide available	Installation guide available
1a.3.1	3+:SWR, urlq, apollo, RTK-Query	3+ (basically all state management libraries)	3+:e.g. USM,Zustand, Flux, Reflux, Dva	<3: jotai	No equivalent library available
1b.1.1	Yes	Yes	Yes, with advanced debugging features	Yes, with advanced debugging features	Yes (includes advanced features but includes bugs and is not updated to support latest version)
1b.3.1	No	No	Yes	Yes	Yes

Table B.1.: Raw metrics results of the analysed State Management Libraries

Metric Id	React Query	React Context	Redux Toolkit	Recoil	MobX
1b.5.1	Breaking API changes in major versions with migration guide and codemod	Breaking API changes in major versions (new Context API in v16)	No breaking API changes introduced major update	Breaking API changes in major versions	Breaking API changes in major versions with migration guide and codemod
1b.6.1	0	0	0	0	0
1b.6.2	110	103	111	114	104
1b.6.3	38	38	39	40	38
1d.1.1	Yes	Yes	Yes	Yes	Yes
1d.1.2	Yes	Yes, through React	Yes	Yes	Yes
1d.2.3	No	No	Yes	No	No
1d.3.1	712	748	796	734	731
1d.3.2	136	149	148	151	131
1d.3.3	73	66	72	76	66
1e.2.1	Yes	Yes	Yes, with advanced debugging features	Yes, with advanced debugging features	Yes (includes advanced features but includes bugs and is not updated to support latest version)
1f.1.1	0.3	0.3	0.2	0.3	0.3
1f.1.2	0.3	0.3	0.3	0.5	0.3

Table B.1.: Raw metrics results of the analysed State Management Libraries

Metric Id	React Query	React Context	Redux Toolkit	Recoil	MobX
1f.1.3	0.6	0.6	0.5	0.6	0.6
1f.1.4	0.3	0.3	0.2	0.3	0.3
1f.1.5	0	0	0	0	0
1f.1.6	0.032	0.02	0.032	0.049	0.032
1f.2.1	Suboptimal re-renders	Redundant re-renders	Suboptimal re-renders	Optimal re-renders	Optimal re-renders
2.1.1	479	340	273	227	340
2.2.1					
2.2.2	547	1304.75	373.25	739.5	405.28
2.2.3	4	12	7	9	9
2.2.4	986	5537	1271	719	1814
2.2.5	1167	5756	833	893	1163
2.3.1	311	843.75	234	119.5	502.85
2.3.2	14.36 %	24.44 %	19.82 %	22.17 %	20.70 %
2.3.8	Advanced examples and guides provided	Hello world provided	Advanced examples and guides provided	advanced examples and guides provided	advanced examples and guides provided
2.3.9	No FAQ available	No FAQ available	11+ Questions	No FAQ available	No FAQ available
2.4.1	Yes	No	No	No	No
2.4.5	No	No	No	No	No
2.4.9	Yes, for sponsors	No	No	No	No
2.5.6	Advanced examples and guides provided	Hello world provided	Advanced examples and guides provided	Advanced examples and guides provided	advanced examples and guides provided
3.2.1	1,422,846	15,448,616	1,784,313	281,558	994,202
3.2.2	29,306.00	193,386.00	8,314.00	17,454.00	25,580.00
3.2.3	1,737.00	39,959.00	770.00	958.00	1,706.00

Table B.1.: Raw metrics results of the analysed State Management Libraries

Metric Id	React Query	React Context	Redux Toolkit	Recoil	MobX
3.2.7	177	0	0	0	19
3.2.8	0	Many	Many	0	Many
3.2.10	Academic publications done by the users of the software product	Academic publications done by the users of the software product	Academic publications done by the users of the software product	No academic publications found	Academic publications done by the users of the software product
3.2.11	Yes	No	Yes	No	Yes
3.4.1	MIT	MIT	MIT	MIT	MIT

B.2. Mapped Metrics Results

Table B.2.: Mapped metrics results of the analysed State Management Libraries

Metric Id	React Query	React Context	Redux Toolkit	Recoil	MobX
1a.1.2	1	5	5	1	5
1a.1.3	5	5	3	5	1
1a.2.2	5	5	5	5	5
1a.2.4	1	1	2	1	1
1a.2.6	3	3	5	3	3
1a.2.7	5	1	5	1	1
1a.2.8	3	5	3	5	5
1a.2.9	1	1	1	1	1
1a.2.10	3	3	5	5	1
1a.2.11	5	5	5	5	5
1a.3.1	5	5	5	3	1

Table B.2.: Mapped metrics results of the analysed State Management Libraries

Metric Id	React Query	React Context	Redux Toolkit	Recoil	MobX
1b.1.1	3	3	5	5	3
1b.3.1	1	1	5	5	5
1b.5.1	4	1	5	1	4
1b.6.1	1	1	1	1	1
1b.6.2	3	3	3	4	3
1b.6.3	4	4	4	4	4
1d.1.1	5	5	5	5	5
1d.1.2	5	3	5	5	5
1d.2.3	1	1	5	1	1
1d.3.1	3	4	5	3	3
1d.3.2	4	5	5	5	3
1d.3.3	3	2	3	3	2
1e.2.1	3	3	5	5	3
1f.1.1	1	1	1	1	1
1f.1.2	1	1	1	1	1
1f.1.3	1	1	1	1	1
1f.1.4	1	1	1	1	1
1f.1.5	1	1	1	1	1
1f.1.6	1	1	1	1	1
1f.2.1	3	1	3	5	5
2.1.1	5	4	3	3	4
2.2.2	4	5	4	4	4
2.2.3	2	5	3	4	4
2.2.4	4	5	5	4	4
2.2.5	5	5	4	4	5
2.3.1	2	5	2	1	3
2.3.2	3	5	4	5	5
2.3.8	5	4	5	5	5
2.3.9	1	1	5	1	1
2.4.1	5	1	1	1	1
2.4.5	1	1	1	1	1

Table B.2.: Mapped metrics results of the analysed State Management Libraries

Metric Id	React Query	React Context	Redux Toolkit	Recoil	MobX
2.4.9	5	1	1	1	1
2.5.6	5	4	5	5	5
3.2.1	5	5	5	2	4
3.2.2	4	5	4	4	4
3.2.3	5	5	4	4	5
3.2.7	5	1	1	1	5
3.2.8	1	5	5	1	5
3.2.10	3	3	3	1	3
3.2.11	5	1	5	1	5
3.4.1	5	5	5	5	5

B.3. Aggregated Questions Results

Table B.3.: Aggregated question results of the analysed State Management Libraries

Id	React Query	React Context	Redux Toolkit	Recoil	MobX
1a.1	3	5	4	3	3
1a.2	4.25	4	3.625	3.75	4.25
1a.3	5	5	5	3	1
1b.1	3	3	5	5	3
1b.3	1	1	5	5	5
1b.5	4	1	5	1	4
1b.6	3.33	3.33	3.33	3	3.33
1d.1	5	4	5	5	5
1d.2	1	1	5	1	1
1d.3	2.67	2.33	1.67	2.33	3.33
1e.2	3	3	5	5	3
1e.4	5	4	5	5	5

Table B.3.: Aggregated question results of the analysed State Management Libraries

Id	React Query	React Context	Redux Toolkit	Recoil	MobX
1f.1	5	5	5	5	5
1f.2	3	1	3	5	5
2.1	5	4	3	3	4
2.2	3.75	5	4	4	4.25
2.3	2.5	3	2	1	2
2.4	3	1	1	1	1
2.5	5	4	5	5	5
3.1	5	5	5	2	4
3.2	3.83	3.33	3.67	2	4.5
3.4	5	5	5	5	5

B.4. Aggregated Goals Results

Table B.4.: Aggregated results of the sub-goals of Software Product Quality

Id	Goal	React Query	React Context	Redux Toolkit	Recoil	MobX
1f	Efficiency	3.8	2.6	3.8	5	5
1e	Usability	4.33	3.67	5	5	4.33
1d	Functionality	3.19	2.67	4.05	3.10	3.38
1b	Maintainability	2.83	2.08	4.58	3.5	3.83
1a	Portability	4.05	4.8	4.325	3.15	2.45

Table B.5.: Aggregated results of the evaluation of the analysed State Management Libraries.

Id	Goal	React Query	React Context	Redux Toolkit	Recoil	MobX
1	Software Product Quality	3.64	3.16	4.35	3.95	3.80
2	Community trustworthiness	4.14	3.73	3.36	3.18	3.68
3	Product attractiveness	4.61	4.44	4.56	2.50	4.33

Bibliography

[1] Atoms | recoil. https://recoiljs.org/docs/basic-tutorial/atoms/. (Accessed on 08/29/2022).

[2] configurestore | redux toolkit. https://redux-toolkit.js.org/api/configureStore. (Accessed on 08/29/2022).

[3] Context – react. https://reactjs.org/docs/context.html. (Accessed on 08/29/2022).

[4] Core concepts | recoil. https://recoiljs.org/docs/introduction/core-concepts. (Accessed on 08/29/2022).

[5] Cqrs. https://martinfowler.com/bliki/CQRS.html. (Accessed on 08/29/2022).

[6] createasyncthunk | redux toolkit. https://redux-toolkit.js.org/api/createAsyncThunk. (Accessed on 08/29/2022).

[7] createslice | redux toolkit. https://redux-toolkit.js.org/api/createSlice. (Accessed on 08/29/2022).

[8] Cumulative layout shift (cls). https://web.dev/cls/. (Accessed on 08/30/2022).

[9] Destructuring assignment - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment. (Accessed on 08/29/2022).

[10] <div>: The content division element - html: Hypertext markup language | mdn. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div. (Accessed on 08/29/2022).

[11] Document object model (dom) - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model. (Accessed on 08/29/2022).

[12] Event sourcing. https://martinfowler.com/eaaDev/EventSourcing.html. (Accessed on 08/29/2022).

[13] facebookexperimental/recoil: Recoil is an experimental state management library for react apps. it provides several capabilities that are difficult to achieve with react alone, while being compatible with the newest features of react. https://github.com/facebookexperimental/Recoil. (Accessed on 08/30/2022).

[14] Fetch api - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API#fetch_interfaces. (Accessed on 08/29/2022).

[15] First contentful paint. https://web.dev/first-contentful-paint/. (Accessed on 08/30/2022).

[16] Flux: An application architecture for react – react blog. https://reactjs.org/blog/2014/05/06/flux.html. (Accessed on 08/29/2022).

[17] In-depth overview | flux. https://facebook.github.io/flux/docs/in-depth-overview/. (Accessed on 08/29/2022).

[18] Introducing hooks – react. https://reactjs.org/docs/hooks-intro.html. (Accessed on 08/29/2022).

[19] Iso 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. (Accessed on 08/30/2022).

[20] Iso 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?start=3. (Accessed on 08/30/2022).

[21] Iso 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?start=6. (Accessed on 08/30/2022).

[22] Jsx in depth – react. https://reactjs.org/docs/jsx-in-depth.html. (Accessed on 08/29/2022).

[23] Largest contentful paint (lcp). https://web.dev/lcp/. (Accessed on 08/30/2022).

[24] Migrating to react query 4 | tanstack query docs. https://tanstack.com/query/v4/docs/guides/migrating-to-react-query-4. (Accessed on 08/30/2022).

[25] Motivation | redux. https://redux.js.org/understanding/thinking-in-redux/motivation. (Accessed on 08/29/2022).

[26] npm. https://www.npmjs.com/. (Accessed on 08/30/2022).

[27] Optimistic updates | tanstack query docs. https://tanstack.com/query/v4/docs/guides/optimistic-updates. (Accessed on 08/29/2022).

[28] Overview | tanstack query docs. https://tanstack.com/query/v4/docs/overview. (Accessed on 08/29/2022).

[29] Presentational and container components | by dan abramov | medium. https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0. (Accessed on 08/29/2022).

[30] React – a javascript library for building user interfaces. https://reactjs.org/. (Accessed on 08/29/2022).

[31] Readme · mobx. https://mobx.js.org/README.html. (Accessed on 08/29/2022).

[32] Recoil. https://recoiljs.org/. (Accessed on 08/29/2022).

[33] Reconciliation – react. https://reactjs.org/docs/reconciliation.html. (Accessed on 08/29/2022).

[34] Redux - a predictable state container for javascript apps. | redux. https://redux.js.org/. (Accessed on 08/29/2022).

[35] Redux toolkit | redux toolkit. https://redux-toolkit.js.org/. (Accessed on 08/29/2022).

[36] Redux toolkit: Overview | redux. https://redux.js.org/redux-toolkit/overview. (Accessed on 08/29/2022).

[37] reduxjs/redux-devtools: Devtools for redux with hot reloading, action replay, and customizable ui. https://github.com/reduxjs/redux-devtools. (Accessed on 08/30/2022).

[38] Rtk query overview | redux toolkit. https://redux-toolkit.js.org/rtk-query/overview. (Accessed on 08/30/2022).

[39] Selectors | recoil. https://recoiljs.org/docs/basic-tutorial/selectors. (Accessed on 08/29/2022).

[40] <span>: The content span element - html: Hypertext markup language | mdn. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/span. (Accessed on 08/29/2022).

[41] Speed index. https://web.dev/speed-index/. (Accessed on 08/30/2022).

[42] Stack overflow developer survey 2021. https://insights.stackoverflow.com/survey/2021#most-popular-technologies-webframe. (Accessed on 04/22/2022).

[43] State and lifecycle – react. https://reactjs.org/docs/state-and-lifecycle.html. (Accessed on 08/29/2022).

[44] Style guide | redux. https://redux.js.org/style-guide/. (Accessed on 08/29/2022).

[45] Support for suspense in rtk query · issue #1574 · reduxjs/redux-toolkit. https://github.com/reduxjs/redux-toolkit/issues/1574. (Accessed on 08/30/2022).

[46] Suspense for data fetching (experimental) – react. https://17.reactjs.org/docs/concurrent-mode-suspense.html. (Accessed on 08/31/2022).

[47] Tanstack query | react query, solid query, svelte query, vue query. https://tanstack.com/query/v4/. (Accessed on 08/29/2022).

[48] Time to interactive. https://web.dev/interactive/. (Accessed on 08/30/2022).

[49] Tip 10-4: Use the quality tree as checklist! | arc42 documentation. https://docs.arc42.org/tips/10-4/. (Accessed on 08/30/2022).

[50] Total blocking time (tbt). https://web.dev/tbt/. (Accessed on 08/30/2022).

[51] Transparent functional reactive programming | mobx quick start guide. https://subscription.packtpub.com/book/web-development/9781789344837/9/ch09lvl1sec51/transparent-functional-reactive-programming. (Accessed on 08/29/2022).

[52] Usage guide | redux toolkit. https://redux-toolkit.js.org/usage/usage-guide#asynchronous-logic-and-data-fetching. (Accessed on 08/29/2022).

[53] Using the state hook – react. https://reactjs.org/docs/hooks-state.html#gatsby-focus-wrapper. (Accessed on 08/29/2022).

[54] Writing tests | redux. https://redux.js.org/usage/writing-tests. (Accessed on 08/30/2022).

[55] A. Adewumi, S. Misra, and N. Omoregbe. Evaluating open source software quality models against iso 25010. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 872–877. IEEE, 2015.

[56] L. Aversano and M. Tortorella. Applying effort for evaluating crm open source systems. In *PROFES*, 2011.

[57] L. Aversano and M. Tortorella. Quality evaluation of floss projects: Application to erp systems. *Information and Software Technology*, 55(7):1260–1276, 2013.

[58] V. R. Basili. Software modeling and measurement: the goal/question/metric paradigm. Technical report, 1992.

[59] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *in ICSE '76: Proceedings of the 2nd International Conference on Software engineering*, pages 592–605, 1976.

[60] G. A. Campbell and S. SonarSource. Cognitive complexity. *SonarSource: Geneva, Switzerland*, 2020.

[61] D. Coleman, B. Lowther, and P. Oman. The application of software maintainability models in industrial software systems. *Journal of Systems and Software*, 29(1):3–16, 1995.

[62] I. E. Commission. *Software Engineering-Product Quality*, volume 9126. ISO/IEC, 2001.

[63] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner. Software quality models: Purposes, usage scenarios and requirements. In *2009 ICSE workshop on software quality*, pages 9–14. IEEE, 2009.

[64] R. G. Dromey. A model for software product quality. *IEEE Transactions on software engineering*, 21(2):146–162, 1995.

[65] F. Duijnhouwer and C. Widdows. Open source maturity model. cap gemini expert letter (august 2003), 2008.

[66] R. B. Grady. *Practical software metrics for project management and process improvement.* Prentice-Hall, Inc., 1992.

[67] K. Haaland and A.-K. Groven. Free/libre open source quality models-a comparison between two approaches. 2010.

[68] J. Hunt. Gang of four design patterns. In *Scala design patterns*, pages 135–136. Springer, 2013.

[69] M. R. Lyu et al. *Handbook of software reliability engineering*, volume 222. IEEE computer society press Los Alamitos, 1996.

[70] I. Maier, T. Rompf, and M. Odersky. Deprecating the observer pattern. Technical report, 2010.

[71] C. Mattson, R. L. Bushardt, and A. R. Artino Jr. When a measure becomes a target, it ceases to be a good measure, 2021.

[72] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[73] J. McCall, P. Richards, and G. Walters. Factors in software quality, volumes i, ii, and iii, us rome air development center reports ntis ad/a-049 014. Technical report, NTIS AD/A-049 015 and NTIS AD/A-049 016, US Department of Commerce, 1977.

[74] J. P. Miguel, D. Mauricio, and G. Rodríguez. A review of software quality models for the evaluation of software products. *arXiv preprint arXiv:1412.2977*, 2014.

[75] E. Petrinja, R. Nambakam, and A. Sillitti. Introducing the opensource maturity model. In *2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 37–41. IEEE, 2009.

[76] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos. The sqo-oss quality model: measurement based open source software evaluation. In *IFIP international conference on open source systems*, pages 237–248. Springer, 2008.

[77] R. Semeteys, O. Pilot, L. Baudrillard, G. Le Bouder, and W. Pinkhardt. Method for qualification and selection of open source software (qsos) version 1.6. Technical report, Technical report, Atos Origin, 2006.

[78] M. Soto and M. Ciolkowski. The qualoss open source assessment model measuring the performance of open source communities. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 498–501. IEEE, 2009.

[79] D. Spinellis, G. Gousios, V. Karakoidas, P. Louridas, P. J. Adams, I. Samoladas, and I. Stamelos. Evaluating the quality of open source software. *Electronic Notes in Theoretical Computer Science*, 233:5–28, 2009.

[80] I. O. F. STANDARDIZATION. Iso/iec 25010: Systems and software engineering-systems and software quality requirements and evaluation (square), 2011.

[81] W. J. Sung, J. H. Kim, and S. Y. Rhew. A quality model for open source software selection. In *Sixth International Conference on Advanced Language Processing and Web Information Technology (ALPIT 2007)*, pages 515–519. IEEE, 2007.

[82] É. Tanter. Beyond static and dynamic scope. *ACM Sigplan Notices*, 44(12):3–14, 2009.

[83] A. Wasserman, M. Pal, and C. Chan. The business readiness rating model: an evaluation framework for open source. In *Proceedings of the EFOSS Workshop, Como, Italy*, 2006.

[84] P. Wegner. Interoperability. *ACM Computing Surveys (CSUR)*, 28(1):285–287, 1996.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 31. August 2022 Youssef Benlemlih