Software Design Patterns Project

React

Submitted By: Darragh Elbel T00211193

Computing with Software Development

Date Submitted: 07/12/23

Contents

[Background 3](#_Toc152874497)

[Existing Structure 4](#_Toc152874498)

[Decorator Pattern – React Fetch 4](#_Toc152874499)

[Rationale, Downsides, and Alternatives 6](#_Toc152874500)

[Complexity 6](#_Toc152874501)

[Composite Pattern – React Children 7](#_Toc152874502)

[Code Analysis 7](#_Toc152874503)

[Composite Pattern Rationale and Alternatives 10](#_Toc152874504)

[Flexibility and Scalability 10](#_Toc152874505)

[Factory Pattern - Context 10](#_Toc152874506)

[Code Analysis 11](#_Toc152874507)

[Factory Pattern Rationale, and Alternatives 14](#_Toc152874508)

[Encapsulation 14](#_Toc152874509)

[Flexibility and Extensibility 14](#_Toc152874510)

[Client Simplification 14](#_Toc152874511)

[Proxy Pattern – React lazy 15](#_Toc152874512)

[Proxy Pattern Rationale, Downsides and Alternatives 17](#_Toc152874513)

[Resources Management 17](#_Toc152874514)

[Simplicity for Developers 17](#_Toc152874515)

[Almost the Memento Pattern - React Memo 17](#_Toc152874516)

[Testing, Validation, and Code Quality 19](#_Toc152874517)

[Jest – Overview and React Implementation 19](#_Toc152874518)

[Code Quality 20](#_Toc152874519)

[Project Contribution 20](#_Toc152874520)

[Project Contribution – In Practice 21](#_Toc152874521)

[Conclusions 26](#_Toc152874522)

[Reflections 26](#_Toc152874523)

[References 28](#_Toc152874524)

## Background

For this project I have chosen to examine the React framework. React is an open-source JavaScript framework developed by Facebook, now Meta, in 2013 primarily used for building user interfaces in web applications. React components are reusable and self-contained pieces of code that dictate how a particular section of the UI should appear and behave. One of React's key features is its virtual DOM (Document Object Model), which optimizes rendering by re-rendering only elements of the DOM whose state has been altered, rather than the entire page. This results in faster and more efficient updates, enhancing the user experience and allowing for highly complex applications to be built that would not be feasible using only vanilla JavaScript, CSS, and HTML. React's component-based architecture, combined with its ability to easily integrate with other libraries and frameworks, has made it a popular choice among developers for creating both single-page applications and complex web apps. Not only is React popular among developers for their own projects, but it is has also seen massive uptake in industry, being used be companies including Instagram, Netflix, Tesla, Airbnb, and more (Fetisov & Talochka, 2023).

While this report will focus on the source code behind the React framework, I will be demonstrating how various modules of the framework are called from within the client by developers, either directly or indirectly. React developers interact with the framework through a layer of abstraction, isolating them from the intricacies and details of complex operations happening behind the scenes. To keep the scope of this exploration manageable, I will primarily be focusing on the modules contained with the ‘react’ package itself available here <https://github.com/facebook/react/tree/main/packages/react/src>. This specific package contains the functionality required to define React components, covering use cases including state management, element creation, HTTP fetch requests, and caching. This report will explore the composite, factory, proxy, and decorator patterns via the functionality exposed through the modules React Children, Context, Fetch, and Lazy respectively. Additionally, this report will take a brief look at React Memo, which bears resemblance to the Memento pattern but after investigation I concluded was not a direct implementation of it. The website <https://sdp.elbel.dev> can be used as a quick reference for any code mentioned within this report, containing live implementations of the components which were used to demonstrate design patterns, and containing links to React’s source code where applicable.

The React framework is written in JavaScript, any source code snippets may also include Flow annotations. Flow is a static type checker for JavaScript, developed by Meta. It allows developers to annotate their code with types, enabling the identification of type inconsistencies before runtime. In the React codebase, Flow is used extensively to annotate function parameters, return types, and object properties, helping to maintain type consistency across the codebase. React’s maintainers are still in the process of porting the entire codebase to use Flow annotations, which is why some snippets may appear without these annotations (Meta, 2023).

As part of this project, I also was able to identify a code smell, develop a fix for it, and submit a pull request to the official React repository. While at the time of writing this pull request has not yet been accepted or denied, it was a beneficial educational exercise to participate in.

## Existing Structure

### Decorator Pattern – React Fetch

The Fetch API is included within the JavaScript language and facilitates the sending and manipulating of HTTP requests. React employs the Decorator pattern to enhance the functionality of the standard JavaScript fetch API. The Decorator pattern is a structural design pattern that can be used to extend or alter the behaviour of objects at runtime, adhering to the open/closed principle by allowing for extension of the fetch API, but not modification, which is encouraged by clean coding standards. In the context of React Fetch, the Decorator pattern is employed to enhance the standard fetch function with additional capabilities tailored to the needs of React applications, this is achieved without altering the original fetch function’s core functionality, ensuring compatibility and avoiding conflicts with existing code.

A diagram of a software application

Description automatically generated

Figure 1 Fetch Decorator

The CachedDataFetcher is a simple component with the sole responsibility of sending a GET request to the PokéAPI, an open-source API available for testing purposes. The UML diagram above demonstrates how the CachedDataFetcher uses the Fetch API, but this request is intercepted and modified by React Fetch itself. This operation is obfuscated from the client, as can be seen in the code snippet below from the client perspective it appears as if the fetch API is being called as normal.



Figure 2 Fetch From Client



Figure 3 Decorated Fetch Request

The code in the figure above demonstrates the implementation of the Decorator pattern to enhance the fetch function. The first line of the snippet acts as a gatekeeper checking feature flags to ensure that the fetch function will only be decorated if the flags enableCache and enableFetchInstrumentation are true. Following this two constants are declared, originalFetch and cachedFetch. The originalFetch variable copies an original version of the fetch function, allowing it to be used as a fallback if necessary, however the application of the Decorator pattern is done through the declaration of the cachedFetch variable. If no dispatcher is found this is an indication that the function is outside of a cached scope, and originalFetch is called directly, however if it is present a unique cache key is generated using the generateCacheKey function. To check if a request has already been cached, a cache.get(url) check is performed against the requested URL, if no entries are found the originalFetch function is executed. If a cached request response is found the corresponding value, which in this case is a promise of the response is assigned to the match variable. This response is cloned using response.clone() as responses are streams that can only be read once.

#### Rationale, Downsides, and Alternatives

When exploring alternatives to the Decorator pattern in the context of React, several design patterns emerge as potential substitutes, each with their unique trade-offs. One alternative that could be considered is the Strategy pattern, a behavioural pattern that allows the definition and swapping of a family of algorithms at runtime. In the case of React Fetch, the Strategy pattern could enable the dynamic selection of different fetch behaviours based on runtime conditions or feature flags, it is worth noting that these flags are already part of the original code seen in the line ‘if(enableCache && enableFetchInstrumentation)’. While this approach would offer the flexibility currently present through the Decorator pattern, it would add an unnecessary layer of complexity to the application.

Another potential alternative to the Decorator pattern is the Chain of Responsibility, which could be used to pass the result of a fetch request through a handler that would then handle caching functionality. Despite being a perfectly plausible way to extend the fetch API’s functionality, it would not offer the direct on-the-fly extensibility that requests decorated by the Decorator pattern would.

Due to the lack of what I believe to be a suitable alternative the use of the Decorator pattern to extend the fetch API is justified. Not only does it enhance the API without modification, but also accommodates future scaling if necessary. A system is already in place to add decorators based on feature flags, there is no reason new behaviours could not be added as decorators in the future.

##### Complexity

The primary downside of the Decorator pattern is the added complexity that comes with it. It introduces additional layers and wrappers to functionality that is already a core feature of the language which can make the code harder to understand and maintain, especially for those not familiar with the pattern. This issue is especially prominent in the case of augmenting the ubiquitous fetch API. This is a very common API to find used in non-React applications and augmenting its functionality without explicit knowledge of the developer is concerning due to the possibility of introducing unintended side-effects.

### Composite Pattern – React Children

The Composite pattern is a structural design pattern that facilitates the composition of objects into tree structures and furthermore provides capabilities to manipulate these structures programmatically. This pattern’s implementation in React plays a pivotal role in both state management and the component rendering process, being well-equipped to process the tree structures that make up React component hierarchies. A tree structure is a structure in which a base root node can have branching nodes, components in the case of React, as children. These children can then in turn have their own children branching from them, making them composite components, or have no children making them simple components, known as leaf nodes in the context of a tree. In React, components can maintain their own state and react to changes in this state or to changes in props passed from their parents, triggering re-renders as required. The design of the Composite pattern ensures that each component, whether it be a simple element, a leaf node with no children, or a composite element, fits into a cohesive tree structure. When a parent component’s state changes, the re-render triggered is not only applied to the parent itself but also cascades down through its child components, requiring updates throughout potentially many branches of the tree. The implementation of the Composite pattern makes this propagation seamless due to its uniform treatment of simple and composite components. The use of this pattern also aids the predictability of a React application’s behaviour as the structure of the component tree directly correlates with the data flow and rendering logic. The virtual DOM leverages this pattern to minimize real DOM manipulations, comparing the state of the tree before and after application state changes which leads to enhanced performance and rendering efficiency.

#### Code Analysis

The ReactChildren.js module provides utilities for handling components and their children. For example, it includes functions like mapChildren, forEachChildren, and toArray, which are used to iterate over, transform, and manage the children of a React component. These functions abstract the handling of children, whether they are single elements or arrays of elements, thus aligning with the Composite Pattern's principle of treating simple and composite components uniformly. The code snippet in Figure 1 contains the mapChildren function, which iterates over a list of React nodes calling the mapIntoArray function on each, which recursively processes each child, and their nested children if applicable.



Figure 4 mapChildren Function

This approach is evident in the mapIntoArray function, which recursively processes children, treating single elements and arrays of elements uniformly. It maps each child, whether it's a single React element or an array of elements, into a new array, applying a callback function to each one. This recursive processing of children, regardless of their individual or composite nature, is a hallmark of the Composite pattern as mentioned previously. The following code snippet was extracted from the aforementioned mapIntoArray function and initially performs a check on the element to ascertain whether it is a simple or composite element, it does this by verifying that the element's child key is not null, if the statement returns true, then it is confirmed to be a composite component. Next an escapedChildKey is generated based on the childKey to uniquely identify each element in the array, this is necessary for React’s reconciliation process to update the DOM. If the component did not pass the composite check it is handled as a simple component and the mappedChild is cloned and has a new key constructed before the component is pushed to the array.



Figure 5 mapIntoArray Snippet

The UML diagram in Figure 6 is a simplified demonstration of React’s implementation of the composite pattern. ReactElement represents a basic React component. ReactComponent extends ReactElement which represents a component with its own state and render method. Finally, ReactChildren is a utility class which implements the Composite pattern to provide functions like map, forEach, toArray, and count to manipulate the children of components.

A screenshot of a computer

Description automatically generated

Figure 6 React Composite Pattern Implementation

#### Composite Pattern Rationale and Alternatives

There are potentially other design patterns capable of handling React’s component hierarchy. The Decorator Pattern, which allows behaviour to be added to individual objects without affecting the behaviour of other objects from the same class could be a potential fit. However, its focus biases towards adding responsibilities to objects rather than managing a hierarchy as a whole. The Chain of Responsibility could also have certain comparisons drawn to the Composite pattern, its ability to pass requests along a chain of handlers could be used as a way to propagate events or state changes through the components making up the nodes of a tree, yet similarly to the Decorator, it lacks the structural processing ability that is intrinsic to the Composite pattern. While the patterns mentioned are not suitable replacements for the Composite, researching them highlighted their potential as complements of the Composite pattern.

###### Flexibility and Scalability

The Composite pattern significantly contributes to the flexibility and scalability of React applications, especially as they grow and evolve. It allows for the easy integration of new component types, fitting seamlessly into the existing component hierarchy without disrupting the overall architecture. This flexibility is crucial in dynamic environments where requirements change frequently, and new features or UI elements are regularly introduced. Additionally, the pattern excels in managing larger component trees, a common scenario in growing React applications. As applications expand, the complexity of their component hierarchies naturally increases. The Composite pattern facilitates the handling of this complexity by enabling a consistent and systematic approach to managing both simple and complex components within the tree. This scalability is vital for maintaining the performance, maintainability, and readability of the code as the application grows, ensuring that it remains robust and manageable even as new layers of functionality and UI complexity are added.

### Factory Pattern - Context

The Factory Pattern ‘is a creational pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created (Refactoring Guru, 2023)’. In the file ReactContext.js the createContext function serves as a factory method. It is responsible for creating and configuring the context object which includes the nested Provider and Consumer objects. The implementation of the Factory Pattern in this context ensures a decoupling of the object’s creation and its utilization, adhering to the principles of encapsulation and abstraction. This design choice enhances the maintainability of the code, as modifications to the context object’s structure or initialization logic can be confined to the createContext function, mitigating potential impacts on the client code.

Context is used as an alternative to the process of prop drilling within a React component hierarchy. Prop drilling is the process by which data is passed down through the levels of a tree, moving from component to component before it finally reaches its destination. The practice of prop drilling can lead to verbose code, potentially passing numerous unnecessary arguments to functional components making code more difficult to understand and maintain (GeeksForGeeks, 2023).

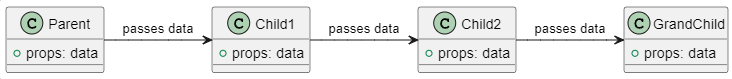


Figure 7 Prop Drilling

Figure 7 illustrates the processes of prop drilling. The data originates in the Parent component, before being passed to Child 1, which then passes it to Child 2, and finally, the data is passed to GrandChild where it will be utilized. It is clear that this process of passing data from the source through multiple nested children is inefficient and could lead to unnecessary memory allocation, context solves this problem.

#### Code Analysis

The UML diagram in the following figure outlines the structure of a small demonstration application written in React that makes use of context. This application contains three user-defined components, ContextParent, ContextChild1, and ContextChild2, ContextParent is the parent of ContextChild1, which in turn is the parent of ContextChild2. As we can see diagram, ContextChild2 is able to declare a variable contextValue and assign it the value of ContextParent’s MyParentContext object without passing data through the intermediate ContextChild1 component. To access this context creation functionality, ContextParent is making a call to ReactHooks, which then calls the ReactContext class for implementation details.

A diagram of a component

Description automatically generated

Figure 8 Context Application UML

The provider object plays a important role in the React Context API, acting as a distributor of values within the component tree. It allows values to be transmitted to nested components, ensuring that updates to the context's value are propagated efficiently. The consumer object, on the other hand, serves as a subscriber to the provide, enabling components to access the context's stored values. It facilitates a reactive connection to the provider, ensuring that any changes in the context's value are reflected in the components that consume the context.



Figure 9 Context Factory

The createContext function in the code snippet above serves as a straightforward example of the Factory pattern in action. It simplifies the creation of ReactContext objects. By encapsulating the details of context object creation, the function allows developers to generate these objects without needing in-depth knowledge of their underlying structure. For instance the assignment of default values to the context object are handled internally within the function, mirroring the Factory Pattern’s approach of centralizing and abstracting the creation logic. This abstracting is made clear when implementing a context provider and context consumer on the client of a React application. In the ContextParent component MyParentContext is intilized and has a message passed to its provider as was seen in the previous UML diagram. To read this message out all that was required of ContextChild was to import the context object and pass it into the useContext hook.



Figure 10 Client Side Context Creation



Figure 11 Client Side Context Consumption

#### Factory Pattern Rationale, and Alternatives

When examining the use of the Factory pattern in react, alternative design patterns emerge as potential candidates for managing object creation, yet they possess distinct limitations compared to the Factory. The Singleton pattern could potentially be considered for creating single context objects that could be shared across the application. However, the rigid structure of the Singleton would restrict the utility of context and violate the principle of separation of concerns. Another alternative, the Builder pattern, offers a fine-grain approach to object creation, allowing for the construction of more complex objects. While this granularity is beneficial for creating intricate objects, it would introduce unnecessary complexity for simpler operations like context object generation. While these patterns provide value in certain scenarios, their limitations in flexibility, complexity management, and task specificity underscore why I believe the Factory pattern was a good choice for React’s context object creation needs.

##### Encapsulation

The Factory Pattern encapsulates the complexity of object creation and configuration, hiding the internal details from the client code. This encapsulation simplifies the client interface, making the API easier to use and understand. It also ensures that any changes in the object’s creation process or internal structure can be made without affecting the client code, adhering to the Open/Closed Principle which states that a class should be open for extension but closed for modification (Cecil, 2003).

##### Flexibility and Extensibility

The Factory Pattern provides a flexible and extensible framework for object creation. If new types of context objects or configurations are needed in the future, new factory methods or classes can be introduced without altering existing client code. This aligns with the Liskov Substitution Principle, ensuring that new subclasses (or factory methods) can be substituted for the base class (or original factory method) without affecting the correctness of the program.

##### Client Simplification

The simplification of client code is a direct consequence of encapsulation and the abstraction provided by the Factory Pattern. By presenting a simplified interface for object creation, the pattern reduces the complexity that the client code must handle. Martin Fowler emphasizes the importance of such simplifications in API design in his book “Patterns of Enterprise Application Architecture”, highlighting how design patterns can be employed to create more intuitive and accessible interfaces (Fowler, 2002). In the case of the React Context API, the createContext function abstracts the intricacies of context object creation, providing a straightforward and easy-to-use interface for developers.

### Proxy Pattern – React lazy

The Proxy Pattern is a structural design pattern that provides an object acting as a surrogate or placeholder for another object to control access to it. This pattern is used to implement React’s lazy loading functionality. It enables efficient loading and rendering of React components, particularly beneficial for optimizing performance in large-scale applications. The React Lazy function serves as a proxy creator. It is responsible for creating a proxy component that acts as a stand-in for the actual component that is only loaded when required. This delayed loading, known as lazy loading, is essential for improving the application's load time by splitting the code at designated points and loading parts of it only when necessary. The proxy component, once invoked, initiates the loading of the actual component. During this process, the proxy component manages various states: it starts in an uninitialized state, may go into a loading or pending state, and eventually resolves to a loaded state or rejects with an error. These state transitions are managed internally within the React Lazy module. The UML diagram in Figure 12 demonstrates the relationship between our App component, which we can consider the root component, and the Loading and LazyLoadedComponent components.

A screenshot of a computer

Description automatically generated

Figure 12 Lazy Loading UML

The code to instantiate this lazy loaded object is quite simple, the component’s import statement is wrapped inside a Reat.lazy function call, after this the stateful Boolean variable loadComponent is declared, once React detects the value of loadComponent being set to true the suspense component will then render our LazyLoadedComponent instead of the fallback Loading component. The snippet below demonstrates this functionality.



Figure 13 Lazy loading from the root

The proxy component created by React Lazy acts as a stand-in for the actual component. It manages the loading process, transitioning through various states (uninitialized, pending, resolved, rejected) as it loads the component asynchronously. Furthermore, the proxy pattern in React Lazy is seamlessly integrated with other React features like Suspense. Suspense allows developers to specify fallback content such as loading spinners are placeholders to display while waiting for the lazy-loaded component. This integration exemplifies how the Proxy Pattern not only improves performance but also contributes to a better overall user experience by handling loading states elegantly.

The code snippet in Figure 14 creates a payload with a status uninitialized. The lazyType object acts as the proxy for the actual component. It contains the payload and an init method which is used for initializing the actual component when it’s required.



Figure lazy function

#### Proxy Pattern Rationale, Downsides and Alternatives

No other design pattern maps to this functionality as seamlessly as the proxy pattern does and for this reason it appears to be the best possible choice for the implementation of lazy loading through a design pattern.

##### Resources Management

The use of the proxy pattern allows for more efficient resource management and application responsivity. It ensures that heavy components, such as those that are either computationally expensive or large in size (e.g. those containing images) are not loaded until they are required. Taking this approach reduces the initial load time of the application.

##### Simplicity for Developers

From a developer's perspective, the Proxy Pattern abstracts away the complexities of lazy loading. Developers can work with components in a standard way, regardless of whether they are loaded lazily or not. This abstraction makes it easier to develop and maintain applications without worrying about the underlying loading mechanisms.

### Almost the Memento Pattern - React Memo

Prior to investigating React’s source code I was already familiar with the concept of memoization within React. This is the process of wrapping an object within a memo to ensure that it will not automatically be re-rendered if its parent object re-renders, instead re-renders will only occur when the object's own state changes.

The Memento Pattern captures the state of an object to allow its saving and restoration later. While the functionality and purpose of React Memo does not map 1-to-1 to the technical definition of the Memento pattern, they share the fundamental concept of preserving state. To demonstrate the processes of component memoization I have created two components, MemoParent and MemoChild which is illustrated by the UML diagram in the figure below. Whenever a user performs an action that calls the incrementParentValue function to increment parent Value a re-render of the MemoParent function will be triggered. However, MemoChild will not re-render because its props, in this case child Value have not been changed, and MemoChild has been memorized.

A screenshot of a computer

Description automatically generated

Figure 15 Component Memoization

The figure above illustrates the React memo function used to optimize rendering by preventing unnecessary re-renders of a component unless its props change. Below is a textbook example of the Memento design pattern, where the Originator is an object that can produce snapshots of its state, the Memento acts as a snapshot of the Originator's state, and the Caretaker is responsible for saving and restoring these states. Upon comparison, we notice some parallels but also significant differences. In the React context, MemoChild could be seen as akin to the Originator, managing its own state and rendering logic. However, MemoChild does not directly save its state; instead, the memo function is responsible for determining if MemoChild should re-render based on new props, similar to how a Caretaker manages Mementos. The key distinction lies in the scope of their operations: the Caretaker can manage multiple historical states (snapshots), while the memo function in React is concerned with the comparison between the current and the next set of props, effectively working with only the present and the immediate previous state.A diagram of a computer

Description automatically generated

Figure 16 Memento Diagram

After exploring the memento pattern, its purpose, and definition, I can confidently say that despite the similarity in naming and the shared responsibility of saving an object's state, React Memo does not embody the memento pattern. While React Memo effectively optimizes performance by caching a component's rendered output based on its props, it does not facilitate the externalization and subsequent restoration of an object's state for later use. The memento pattern's focus is on preserving the complete state of an object outside of itself, allowing for full reversion to past states, akin to an undo feature in software applications. React Memo's approach is more about controlling the rendering process and avoiding computational overhead than about state history management. Therefore, while both concepts deal with state in some form, their applications, mechanisms, and objectives are distinct.

## Testing, Validation, and Code Quality

React’s testing strategy relies primarily on two key approaches, rendering component trees in a simplified test environment and asserting on their output, and running complete applications in a realistic browser environment. The focus is more on the former, where components are tested in isolation to ensure their correct functionality. There are considerations to be made when deciding on a test approach, some tools offer quick feedback but may not model browser behaviour 1-to-1, conversely tools that use real browser environments can be slower and less reliable when incorporated into continuous integration pipelines. Jest is the testing framework recommend for testing react components, alongside this React also suggests using the React Testing library (React, 2023).

### Jest – Overview and React Implementation

Jest is a JavaScript testing framework which like React is maintained by Meta. One of the key features of Jest is its powerful and flexible mocking capabilities, which make it easy to test asynchronous code and components in isolation. This is particularly useful in a React environment where component isolation is important. Jest automatically mocks CommonJS modules, which allows you to write tests in a modular and maintainable way. Another notable feature of Jest is its Snapshot Testing. This allows you to capture snapshots of React trees or other serializable values to simplify testing and to analyse changes in outputs over time. These snapshots are human-readable and act as a powerful tool for identifying changes in your application's UI. Jest also comes with a built-in test runner and assertion library, eliminating the need for additional setup or integration with other test runners (Jest, 2023).

In the context of the React codebase tests are organized on a per-package basis. These tests are housed within the \_\_tests\_\_ folder of any given package. A simple example of a Jest test can be seen in Figure 17, its responsibility being to verify the React version in use aligns with version declared in the package.json dependencies. I have omitted the copyright notice from the comment heading the file for the sake of clarity, however I have left the ‘@jest-environment node’ declaration, this is used to indicate that the test is intended to be run with a NodeJS environment, not the browser. Following this the test itself will be executed based on the gate function passing when specific flags are set. The test logic is simple, importing the React package and the react/package.json file and running an expect assertation between their version numbers.



Figure 17 Jest Test Example

### Code Quality

React’s codebase is very large at 385+ thousand lines of code so in order to investigate it’s code quality I forked the repository and then used Sonarcloud to scan that fork. The resulting summary output of this scan can be seen in Figure 18.

A screenshot of a computer

Description automatically generated

Figure 18 Sonarcloud Summary

Across the codebase Sonarcloud identified 558 bugs, 13 thousand code smells, 8 security vulnerabilities, and 242 security hotspots, garnering a reliability rating of E, A maintainability rating, D security rating, and an E for security review rating. On further investigation into the prevalence of code smells I discovered that over 5,500 of these were detected within Jest test directories, a large number of them being for the error “Unknown property ‘prop’ found”. This was flagged as a code smell as in a live environment it is important to avoid using unknown or invalid properties to prevent unexpected behaviour at runtime, however in a test environment where a small subset of the overall system is being simulated this is not an issue, and hardcoding props is the best approach to ensure maintainability and readability.

### Project Contribution

The React project is open to contributions from community members. Contributing to the React project involves adhering to its Code of Conduct based on the Contributor Covenant and participating in open development on GitHub, where both core team members and external contributors send pull requests. React follows semantic versioning, and contributors are expected to submit changes directly to the main branch, using feature flags for experimental features. Bug tracking and reporting is conducted via GitHub Issues, with a separate procedure for security bugs. Contributors can connect through various channels like IRC, discussion forums, or Discord. Significant changes, especially those affecting the public API, should be proposed through an issue first. New contributors can start with "good first issues." Pull requests are reviewed by the core team, and contributors are expected to adhere to specific prerequisites and follow a development workflow that includes testing, code formatting, and style guidelines. Substantial changes go through a Request for Comments (RFC) process. All contributions are licensed under React's MIT license (Meta, 2023).

This approach of blended community and core team contributions to the project has proven to be productive for React. At the time of writing, the React GitHub repository has 13,699 closed pull requests, and a further 1,100 open issues in active discussion and development by the community.

### Project Contribution – In Practice

As part of my investigation in the React framework, I felt it would be beneficial to gain hands-on experience identifying a code smell, fixing it, performing prerequisite checks and testing, and finally submitting a pull request to have the pull request merged to the codebase. The code smell flagged by Sonarcloud that I chose to remedy was an unaddressed TODO comment present in the ReactChildren.js file, by using git blame I discovered that this specific TODO comment has been present in the codebase for the last 6 years.

A close up of a text

Description automatically generated

Figure 19 TODO Comment

As can be seen in the figure the comment states that there should be a test to verify that a single child and an array with one item have the same key pattern. Before immediately going to write a test I first checked the ReactChildren-test.js to verify that a test for this case was not already present and that the TODO comment had not simply been forgotten. I was able to identify related tests such as ‘should treat single arrayless child as being in an array’ which verifies an expected key pattern, but does not directly compare to the key pattern of a newly created child within an array, and ‘should treat single child in array as expected’. These tests can be seen in the figures below, the first of which does expect a specific key, but this is hardcoded and does not compare the dynamically generated keys of a single child and of an array with one item.



Figure 20 Related Test 1



Figure 21 Related Test 2

My proposed test that would allow the TODO comment to be removed can be seen in the following figure. My suggested test first instantiates two constants, singleChild and arrayWithOneItem. The value of the singleChild variable is a simple variable, with no user assigned key, the value of arrayWithOneItem is an array, containing a single div element. After this I am applying a React.Children.map function on both of declared variables to extract their keys, before finally performing an expect assertion to verify their equality.



Figure 22 New Test

Before submitting a pull request to include this test a number of requirements first had to be met. All contributors to any of Meta’s open source projects must first sign the Contributor License Agreement, this agreement once signed grants permission for individuals to submit pull requests. If the CLA is not signed prior to submitting a request, the request will automatically fail. An example of a failed CI/CD pipeline due to lack of CLA signature can be seen below. Alongside the CLA signature check we can see a number of other CI/CD checks a pull request must go through, in total there are 36 checks. Due to this extensive pipeline Meta recommends running the full React test suite before submitting changes to ensure no changes made have unintended consequences elsewhere in the application.

A screenshot of a computer error message

Description automatically generated

Figure Failed CI/CD

After I finished implementing the new test I ran the React Children test suite, prior to adding my test there were 33 tests, after adding my test there were 34 tests, which all passed as can be seen below.

A black screen with white text

Description automatically generated

Figure 24 ReactChildren Test Suite

Once a pull request is submitted there is no definitive guide on how long that request will take to be viewed and approved/denied by a React core team member, especially for a request as trivial as adding a test to clear an old TODO comment. The following figure is a screenshot of the pull request creation page on GitHub where you can see the git diff of your changes against the main branch of React. I submitted this pull request and it successfully passed all checks, and had no conflicts with the base branch.

A screenshot of a computer

Description automatically generated

Figure 25 Pull Request Creation Screen

Finally once the request was submitted and checks passed, the pull request can be seen on the official React repository in GitHub.

A screenshot of a computer

Description automatically generated

Figure 26 Final Pull Request in Official Repository

## Conclusions

The React framework is a vast set of modules and systems all working together with the goal of providing a cohesive tool to allow developers to create fast and reactive web applications. Design patterns were identified within four separate modules of the React package, and by examining them and their functions within the system, each proven to have merit in their application, and weren’t just design patterns for the sake of design patterns.

Reacts testing strategy is comprehensive, but when viewed through the lens of code quality the framework is beginning to show its age. While many of the code smells identified by Sonarcloud were shown to be erroneous, there are still smells such as old TODO comments and the use of var which is no longer recommended since the release of let variable type in 2015 which offers scoped declarations and avoids variable hoisting.

## Reflections

I found this project to be challenging, but also hugely beneficial, it is rare that as students we have the opportunity to commit such a large amount of time to immersing ourselves in a specific piece of software and exploring its intricacies in a formal fashion. The React framework is unique in that the entire codebase of 350,000+ lines serves a single purpose of providing the tools for developers to create reactive web applications. While I was exposed to large systems while on my work placement in Salaso, none of them came close to the level of uniformity and standardization that I saw when working with React.

Working on identifying design patterns in a professionally developed system proved to be difficult for me. Up to the point of starting the project, my only real exposure to design patterns had been in an academic setting in isolated examples, and many of these examples that were covered in class and in other resources found online and in books such as Head First Design Patterns used Object-Oriented languages and applications to demonstrate the practical applications of patterns. Initially, I struggled to identify any patterns in React, as I was dealing with the learning curve of familiarising myself with a new type system (Flow) while also attempting to identify patterns in a non-OO setting.

Finally, working on identifying a code smell that I could fix was tough, as mentioned in the testing section tyring to find a valid code smell that wasn’t triggered by the issues being intentionally implemented poorly as part of tests for example was difficult. The TODO comment I chose to work on clearing is a small change but gave me great insight into the process that open-source projects such as React go through when accepting contributions. It was surprisingly intimidating, possibly due a fear of my work being incorrect or misguided, but overall an important thing to experience and has encouraged me to look into contributing to more open source projects in the future, or perhaps even publishing some of my own personal projects publicly.

# References

Cecil, R. C., 2003. The Open-Closed Principle. In: *Agile Software Development Principles, Patterns, and Practices.* s.l.:Pearson Education Limited, pp. 99-109.

Fetisov, E. & Talochka, A., 2023. *Top 30 Companies using React JS Development.* [Online]   
Available at: https://jaydevs.com/top-companies-using-react-js/  
[Accessed 1 October 2023].

Fowler, M., 2002. Domain Logic Patterns. In: *Patterns of Enterprise Application Architecture.* s.l.:Addison-Wesley, pp. 133-162.

GeeksForGeeks, 2023. *What is prop drilling and how to avoid it?.* [Online]   
Available at: https://www.geeksforgeeks.org/what-is-prop-drilling-and-how-to-avoid-it/  
[Accessed 22 October 2023].

Meta, 2023. [Online]   
Available at: https://github.com/facebook/react/blob/main/packages/react/src/ReactContext.js  
[Accessed 31 October 2023].

Meta, 2023. *Flow: A Static Type Checker for JavaScript.* [Online]   
Available at: https://flow.org/  
[Accessed 31 October 2023].

Meta, 2023. *How to Contribute.* [Online]   
Available at: https://legacy.reactjs.org/docs/how-to-contribute.html  
[Accessed 19 November 2023].

React, 2023. *Testing Overview.* [Online]   
Available at: https://legacy.reactjs.org/docs/testing.html  
[Accessed 18 November 2023].

Refactoring Guru, 2023. *Factory Method.* [Online]   
Available at: https://refactoring.guru/design-patterns/factory-method  
[Accessed 31 October 2023].