Andrei Vasilev

Comparison of React components testing patterns

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

1 November 2021

Abstract

Author:	Andrei Vasilev
Title:	Comparison of React components testing patterns
Number of Pages:	38 pages
Date:	1 November 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solution
Supervisors:	Hannu Markkanen, Researching Lecturer

There is an enormous amount of various testing patterns, and tools for testing React applications. Each of them is intended for specific purposes and has its concept and philosophy beyond. Such abundance gives good flexibility for developers; however, it also shifts the responsibility for the right decision for themself. A mistakenly picked testing pattern can cause harm down the road.

The purpose of this study is to provide a comprehensive comparison of different approaches to testing React components, analyse the advantages and disadvantages of each and try to find the balanced testing recipe for medium and large size React applications. Moreover, based on the research and development experience, identify the deficiencies in the testing patterns of the study case React application and provide options for improving it.

The study results show the importance of choosing the correct testing tools and pattern for a specific React application and business needs and describe the wrong choice's consequences.

Keywords: Software testing, component testing, unit testing, React, quality improvement

Contents

List of Abbreviations

QA	Quality Assurance
AST	Automated Software Testing
E2E	End to End testing
AAA	Arrange, Act and Assert
RTL	React Testing Library
MVP	Minimum viable product

1 Introduction

Building software is a complex process that requires great effort and time to build a robust, high-quality product that would satisfy all business and user needs. To become competitive and satisfy the rapidly changing business requirements, big and medium-size IT companies introduce software development processes such as requirements analysis, software design, implementation, and testing.

One of the crucial parts of software development processes is software automation testing. The word automation means that software is being tested using automation tools rather than doing it manually.

Automation testing aims to recognise many potential bugs and errors before delivering a product to the customer and ensure system operability at different levels. In addition to that, automation testing gives developers immediate feedback about the correctness of their work and meeting business requirements.

Thereby all stakeholders may verify that the new or modified part of the project works as expected and does not adversely impact other program parts. Furthermore, continuous test writing increases the developers' productivity, reduces debugging time, eradicates fear of change, and improves code quality and project architecture.

As any other software, web applications need to be appropriately tested to guarantee good performance and usability. Nowadays, single-page applications (SPA) have become more and more popular. With such popularity, competition and complexity of applications are also growing. Modern web has nothing in common with websites that took place a decade ago.

There is a huge variety of open-source technologies on the market for building web applications of any complexity. However, more popular, and stable for

2021 are Angular, React and Vue. All these technologies have their advantages and disadvantages and have been created to solve specific problems.

The main goal of this paper is to research and understand the intricacies of existing approaches and tools for testing React components, get their advantages and disadvantages, and define their scope of usage. Examine which testing patterns provide maximum value with minimum development costs and maintenance efforts, demonstrate their relevance and influence on the end product. This is quite a controversial topic, because there is not the only proper way to accomplish it. Different libraries have different testing concepts underneath and they are pushing their own ideas and concepts, which they believe are the only ones.

To fully understand the problem and existing solutions, as an example will be considered and improved some component tests in the project where the author of this work is one of the main contributors and maintainers.

The project is a part of an enterprise-level advertisement SaaS platform written using the popular modern React library. It uses the various testing libraries and patterns, which makes it a good subject for analysing.

Chapter 2 contains an overview of the software automation testing process and gives the primary understanding of core testing principles and its importance and influence on the development and end product. Chapter 3 briefly introduces the React framework and describes its main concepts and problems it solves. Chapter 4 describes and compares various React components testing approaches, techniques, appropriate tools, and libraries. Chapter 6 assesses the current state of the study case test code and provides the steps to improve the tests patterns and, consequently, the overall application quality.

2 Software testing

Software testing is a risk management strategy that helps to verify that the software meets all functional software requirements, is defect-free, and has enough reliability to deliver it to the end-users. [1.] There are two ways of testing software applications:

- Manual testing
- Automation testing

In manual testing, a QA analyst performs software testing step by step to ensure that the end-product does not contain any critical bugs. However, testing of tremendous and complex systems can be a human-intensive or sometimes even impossible task.

To ensure the application works correctly, in some cases a thousand tests need to be executed, which is physically impossible for the human. For this purpose, a special software written by developers and which automatically runs the test cases and generates the test results has been created [2,3].

2.1 Testing levels

Many diverse testing levels help to check the behaviour and performance of the system. This study considers only the most used, such as unit, integration, and system testing. [5.]

Testing levels:

- Unit testing checks that the isolated unit's functionality matches the business specifications.
- Integration testing checks how the units work in integration with each other, and the data flow from one unit to another
- System testing (E2E) check the whole system efficiency from the user perspective of view

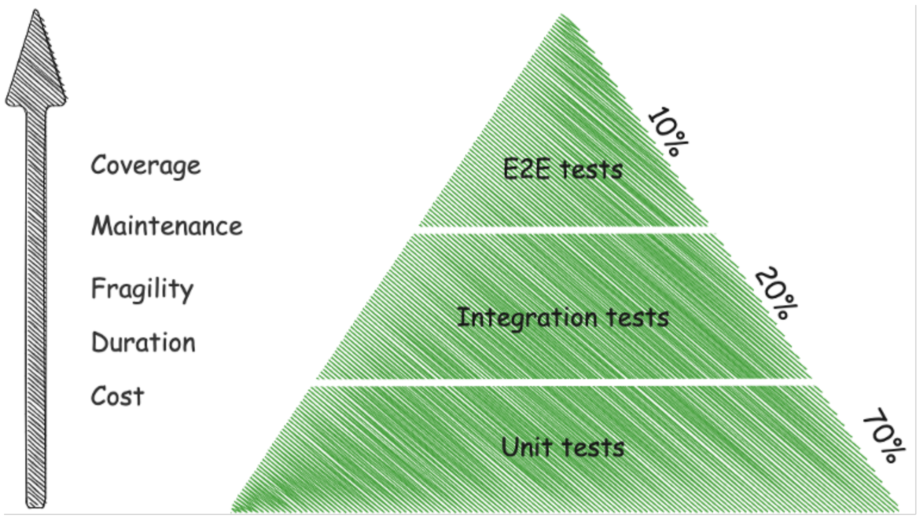

Figure 1. Create by Mike Cohn software testing pyramid.

Figure1 shows the testing pyramid. It is the concept which Mike Cohn introduced in his book "Succeeding with Agile". It describes the different layers of software testing and the required number of tests for each of them. The low level contains many small and fast unit tests, middle some more coarse-grained tests and very few high-level tests that test an application from end to end. The provided concept is quite simplistic, and modern testing frameworks have much more testing layers. However, it gives a solid rule of test organising and granularity [6.]

2.2 Unit testing

Unit testing is the first level of testing, where individual components or software units are tested independently from other parts [7]. Unit testing isolates a piece such as a separate function, class, component of the codebase. It verifies its correctness, which can help find and fix low-level bugs in the early development stage. Usually, unit tests serve as project documentation or requirements that eventually help the developers understand the purpose of a particular unit and make changes quickly. [8.] Furthermore, unit testing ensures sustainable growth of the software project [9]. A software project without unit tests becomes unmaintainable in the long term and requires enormous effort and cost, leading to total project failure.

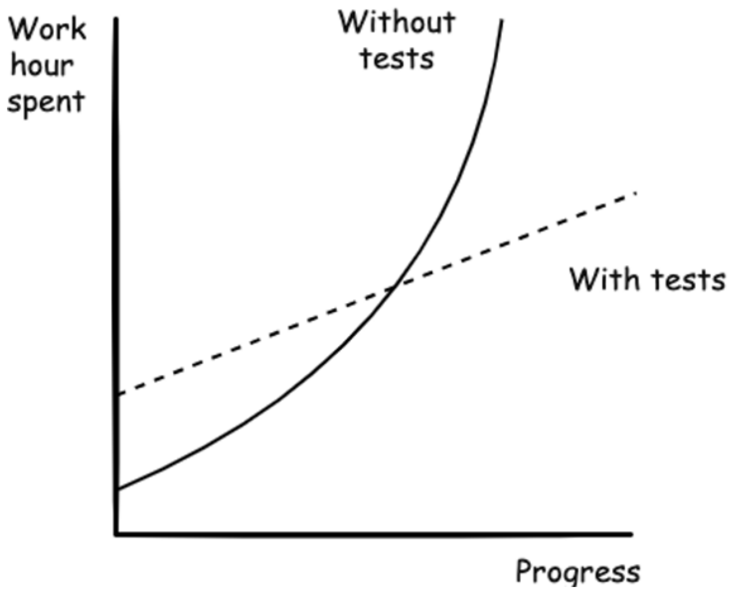

Figure 2. Relationship between amount of work hours and progress with and without tests with time.

Figure 2 demonstrates the development speed with and without unit tests in the long term. Extra time spent at the beginning of the project allows a company to maintain and develop it for years.

Usually, the ratio between production code and test code is between 1:1 and 1:3 (there are one to three lines of test code for each production line code). If production code is hard to unit test, it is the first symptom of badly designed code that requires some improvements. Usually, the reason for that is a tight coupling, which means units are coupled with each other, so it is not easy to test them in isolation. Nevertheless, the ability to unit test a module cannot indicate good code quality. The project can contain terrible code even with loose coupling. [9.]

2.2.1 Bad and good unit test

A unit test can be either good or bad. Good unit tests are valuable and contribute to overall software quality. Poor tests raise false alarms, do not catch bugs, are slow and difficult to maintain. Projects with a vast number of low-quality tests, which value is close to zero, can slow down code deterioration initially, but in the long term, stagnation is still inevitable [9].

It is crucial to consider both the test's value and its upkeep cost to provide good quality unit tests. - The cost part is determined by the amount of spending time on various activities:

- Updating the test after changing the underlying component
- Test execution time on each code change
- Reading tests to understand how tested component behaves

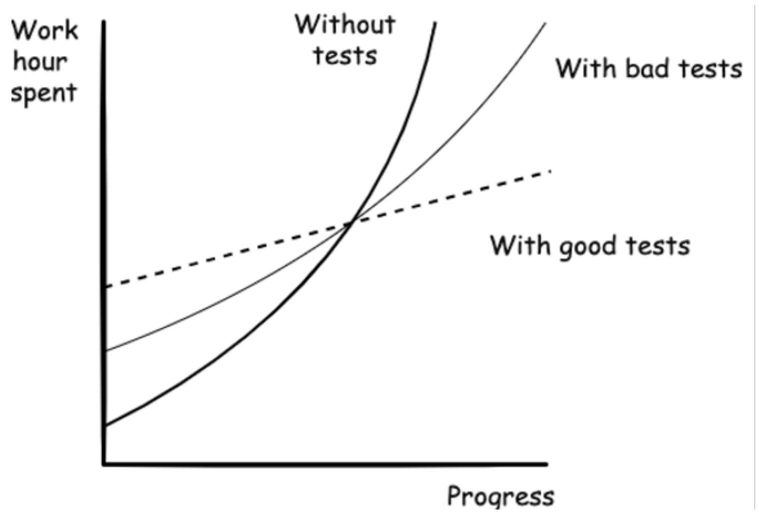

Figure 3. Relationship between amount of work hours and progress with good and bad tests with time.

Figure 3 demonstrates the difference between projects with a good and bad test. The project with poorly written tests exhibits the properties of a project with good tests at the beginning, but it eventually falls into the stagnation phase. [9]

2.2.2 Code coverage

Code coverage or test coverage is a significant and frequently used metric in unit testing, which shows the ratio of the number of code lines executed during the test and the total number of lines in the source code [9].

$$Test\ coverage = \frac{Lines\ of\ code\ executed}{Total\ number\ of\ lines} \qquad (1)$$

Formula 1 shows how the test coverage calculation is performed, the number of executed lines of code divided by the total number of lines. For instance, if the number of lines of code in a project is 100 and the number of lines executed across all existing test cases is 10, then the test coverage is (10 / 100) * 100 = 10%.

The branch coverage gives more precise results because instead of using raw code lines, this metric concentrates on control structures such as if and switch statements. It provides an amount of traversed control structures by at least one test in the suit [9].

$$Branch\ coverage = \frac{Branch\ traversed}{Total\ number\ of\ branches} \qquad (2)$$

Formula 2 shows the calculation of test branch coverage. It is a relationship of traversed branches to the total amount of branches.

An amount of unit tests in the project does not tell anything about its quality. It is easy to reach high numbers of test coverage even with low-quality testing. [10.] even 100% test coverage cannot always guarantee that the test verifies all the possible outcomes of the executed code under a test. Nevertheless, a low percentage of test coverage can indicate that there are not enough unit tests in the project. [9.]

2.2.3 AAA pattern

The AAA pattern is one of the popular unit testing patterns. It helps to arrange and organise test code to make it more readable and understandable by separating each unit test into three parts: Arrange, Act and Assert. [4;16.]

```
// Arrange
const mailService = new MailService();

// Act
mailService.sendEmail(emailData);

// Assert
expect(sendEmail).hasBeenCalledTimes(1);
```

Listing 1. Example AAA testing pattern

Listing 1 demonstrates the usage of the AAA pattern. The first arrangement step is preparation for the test, where a mailService instance of the tested class. In the actual stage, the method to be checked is called. And in the final assertion stage, the assertion happens. The test verifies that the tested sendEmail method has been called once.

Arrange is the preparation section of the unit test, where all initialisations, mocks, inputs, environment setups, and other things that help arrange the test should be done. [4.]

Act is the section where the test acts to the tested unit and performs desired interaction, such as clicking the button, calling method, entering text to the input field. [4.]

Assert is the third part where the test asserts that the interaction, performed in the act section, gave the expected outcome, such as verifying that the correct method has been called or the error message displayed. [4.]

2.2.4 Classical and London approach

There are two different schools of the unit testing Classical and London. People who prefer the Classical school are known as classicists and the others as mockists. The root cause of the dispute between London and Classical schools is the unit's isolation manner and definition of the unit term itself.

London school advocates that all unit's dependencies should be replaced with test doubles, which provides an opportunity for separating behaviour of tested units from any external influence.

In the Classical approach, the code should not be tested in an isolated manner, but unit tests themselves should be run in isolation; therefore, executed in parallel unit tests should not affect each other via shared dependencies. A typical example of such dependency is a database updated by different tests simultaneously and deterministically affecting the different tests [9.]

2.2.5 Test doubles

The test doubles are a simplified version of the original unit that reduces the complexity and facilitates testing [9]. It helps isolate the tested code from the surroundings and get from its expected behaviour [11]. There are much more benefits of using doubles in the unit test:

- Isolate the code under test
- Speed up test execution
- Make execution deterministic
- Simulate special conditions
- Gain access to hidden information

There are multiple types of test doubles:

- Dummy objects usually are used to fill a parameter list. For example, it can be passed as a fake config to the function or class constructor.
- Fake object has some simple working implementation that does not suit the production usage but perfectly works under the test suite. For example, a fake database can be an in-memory implementation of the actual heavy database.
- Stub is an object used to fake a method that has pre-programmed behaviour. It can be used instead of an existing real method in order to avoid unwanted side-effects. (e.g make fake http requests and get defined in advance data).
- Spy is an enhanced stub that also collects some meta-information based on how it was called. In a simple case, it tracks the numbers of calls or provided arguments.
- Mock - is an object used to fake a method that has pre-programmed behavior as well as pre-programmed expectations. if given expectations are not met the requirements the mock will cause the test to fail. (I.e., if a mock of the function has been called with an unexpected argument, it can throw an exception.)

2.3 Integration testing

Integration testing is software testing where individual units are integrated logically and tested as a group [6]. Another definition of integration test is a test that verifies that a unit or units work correctly with shared dependencies, such as a database or microservice developed by other teams' code [9]. Integration testing is necessary because even if all separate units were tested well with unit tests, there would be no guarantee that they will work together as a system or a part of a system. Therefore, this level of testing aims to expose defects between software units when they are integrated.

2.4 System testing

System or end-to-end testing (E2E) is a software testing level that involves testing an application workflow from start to end and integrating with external interfaces. System testing aims to test the whole application for dependencies, data integrity, and communication with other systems such as third-party services, interfaces, and databases. [9.] For example, a simple user login end-to-end testing workflow implies going through the login page and checking the username and password validation, password strength, and error messages. Usually, system testing simulates the system usage of the end-user.

3 ReactJS library

React is a declarative and component-based library for building user interfaces across different platforms such as web, mobile, desktop. That means everything in React application is build using components. React render system, in turn, manages these components and keep the application up to date according to the current state. Components are a building block of React applications. It should be easy to think about and integrate with other React components. Each component follows the predictable lifecycle and can have its inner state. Component with own internal state called stateful and without it stateless component. [12.] React application is a composition of hundreds, or thousands of components nested inside each other. It has a robust and diverse community and an enormous number of third-party libraries, which can help to solve almost every problem.

3.1 Pros of using React

React is one of the most popular and loved JavaScript frameworks nowadays, and there are a few reasons for that. The most apparent benefits of using React are:

- Performance - React is high-speed technology because of the virtual DOM and internal comparison algorithms, which first apply DOM changes to the virtual DOM and update only affected elements in the real DOM. This mechanism guarantees a minimum update time to the real DOM and provides higher performance and a smooth user experience. [13.]
- Learning curve - React is easy to learn and easy to use. Every developer with a JavaScript background can start to write React application after reading through the official documentation.
- Cross-platform - React can be used for developing cross-platform applications such as mobile, native and desktop [13]. Thereby reducing development time and saving money on hiring platform-specific developers.
- Excellent developer tools - React provides the extension for most popular browsers, allowing developers to inspect component

hierarchies in the virtual DOM, check component data flow, and easily debug the logical and performance issues [13].

3.2 Cons of using React

There is no silver bullet for all problems and despite React is a great tool, it also has some drawbacks. React is positioning itself as a UI library, which means it is not designed to do many things out of the box in comparison, with more comprehensive frameworks such as Angular.

React does not provide opinionated solutions for various development aspects such as HTTP, routing, data modelling.

One of the main downsides of choosing to react is the freedom that it gives to developers, as opposite to Angular, with already predefined architecture and development patterns, React developers should find their own custom solution for different areas of the application. With this approach, the end product result increasingly depends on developer experience and qualification because the responsibility of design and architecture of application entirely lies with them. [12.]

3.3 Virtual DOM

The reason for the high-speed performance of React is a virtual DOM. Virtual DOM is the collection of data structures that represent the browser DOM. Direct browser DOM manipulations are an expensive operation that significantly reduces the application performance. To prevent it, React keeps the virtual DOM in memory and performs internal diffing to determine what element has been changed and makes an intelligent update of the real DOM. This process is called reconciliation. [12;14.]

It is easy to imagine virtual DOM as an intermediate layer between the application and the browser DOM. It encapsulates the complexity of diffing and management from the developers to a specific layer of abstraction.

3.4 Components

Components are the building blocks of everything in React. They are well encapsulated, reusable and composable. These traits give the ability to form the new complex composite component through a composition of smaller ones. Component composition is one of the most potent aspects of React. Each component can be easily reused for the rest of the application. [12.]

3.5 JSX

JSX is XML / HTML like syntax extension to ECMAScript which allows HTML like text co-exist with JavaScript / React code. JSX facilitates React component creation, which significantly reduces React learning curve. The syntax is intended to be used by pre-processors (i.e., transpilers like Babel) to convert HTML-like text into standard JavaScript objects that a JavaScript engine will parse. [15;14.]

```
<div className="sidebar" />
```

Listing 2. Snippet of code using JSX syntax.

```
React.createElement('div', {className: 'sidebar'})
```

Listing 3. JSX code from Listing 2 compiled to pure JavaScript.

JSX code shown in Listing 2 compiles into React element demonstrated Listing 3, which is pure JavaScript.

3.6 Props and state

To pass the data into a component from outside, React uses a mechanism called props. Props are the primary way to pass the immutable data into the component. They can be provided to the component from the parent component or via the "defaultProps" static method in the component itself. [12.]

```
const Parent = () => <Child name='child'/>

const Child = ({name}) => <div>{name}</div>
```

Listing 4. Usage of React props

In the listing 4 Parent component passes the name prop with string value ‘child’ to Child component. Child component, in turn, receives the name prop and can perform any manipulations with it.

In addition to the ability to receive props, each component can contain its state. The state is a mutable data structure that preserves the component local data over the component's lifetime. The internal state allows making the component more complex and interactive. To sync component with its state, React re-renders the component after each state change. [12;14.]

```
const ReactComponent = () => {
	const [state, setState] = React.useState(0);
	const handleButtonClick = () => void setState(prev => prev + 1);
	return (
		<div>
			<p>Current state is {state}</p>
			<button onClick={handleButtonClick}>Increment</button>
		</div>
	)
}
```

Listing 5. Shows the usage of React component state.

Listing 5 demonstrates the React component with internal state, which has been defined using a special function (hook) useState(0). useState hook receives the initial value of the state and returns the state (state) and setter function (useState). To update the state setState must be called with a new state in this case it is the previous state + 1.

4 Testing React components

4.1 Test Runners

A test runner is a tool for creating, running, and executing unit tests in an appropriate environment and providing a comprehensive report with test details such as execution time, amount of completed and failed tests, and test coverage. The most popular JavaScript test runner (test framework) is Jest. Jest is the default test runner in CRA (tool for bootstrapping React application). Jest provides a straightforward test structure. The basic test structure is shown in Listing 6. Each test suit contains one or more describe blocks that group several related tests. Function test is an actual test block, where usually happens expected outcomes assertions. [16.]

```
describe("Data transformation function", () => {
     test("should transform input data from one shape to another", () => {
          // actual test
     });
});
```

Listing 6. Demonstrates the basic structure of the test using Jest test runner.

4.2 Basic React component testing

The simplest way to test React component is to render the component itself using “render” method from “react-dom” library, the same method is used for the production application, and then check that render output contains required data. [14.]

```
test('should render component with name prop', () => {
     container = document.createElement("div");
     document.body.appendChild(container);
     act(() => { render(<Component name="Hello world!" />, container)});
     expect(container.textContent).toBe("Hello world!"); }
)
```

Listing 7. Shows the simplest way of testing React components via rendering it using “react-dom” library.

Shown in Listing 7 global function "test" from Jest framework is used for running the test. It contains the test name and test body, where the main logic is defined.

The code presented in Listing 2 is a working example of basic component testing. However, it contains much boilerplate code, slowing down the development. The third-party React testing libraries encapsulate recurring logic and provide a simple interface for a more convenient testing experience.

Efficient test writing requires additional testing tools such as a test runner, which executes unit tests and gathers the required information to provide a resulting report and testing util libraries such as Enzyme or React testing library. These libraries encapsulate the test's arrange stage and provide an easy way to test React component output.

4.3 Shallow mount vs mount

Shallow mount or shallow rendering (shallow rendering and shallow mount are interchangeable conceptions) renders the component in complete isolation and one level deep, which means it allows assertions about what its render method returns without worrying about the behaviour of child components. Shallow rendering does not require a DOM because the output of it is a plain JavaScript object. [9.]

```
const Children = ({title}) => <div>{title}</div>
const Parent = (props) => <Children title={props.title}/>

/** Shallow rendering output */

{ "nodeType": "component",
      "props": {
            "title": "child"
      },
      "instance": null,
      "rendered": {
            "nodeType": "component",
            "props": {
                  "title": "child"
            },
            "instance": null,
            "rendered": {
                  "nodeType": "host",
                  "type": "div",
                  "props": {
                        "children": "child"
                  },
                  "instance": null,
                  "rendered": [
                        "child"
                  ]
            }
      }
}
```

Listing 8. Output of Shallow rendering using the “react-test-renderer” library.

Listing 8 illustrates the shallow mount output of React component with one child. Output is a pure JavaScript object.

Mount or full rendering renders an application tree in a browser-like environment, using the “jsdom” library, a headless browser implementation in pure JavaScript. As shown in Figure 9 it renders the whole component tree with all nested children and executes all component lifecycle methods. The output of the mount is pure HTML. The deep rendering is suitable for testing interactions between components and DOM.

```
const Children = ({title}) => <div>{title}</div>
const Parent = (props) => <Children title={props.title}/>

/** Mount output */

<body>
     <div id="container">
          <div>child</div>
     </div>
</body>
```

Listing 9. Output of mounted component.

4.4 React testing libraries

Before considering third party libraries, it is worth noting the testing tool provided by React team itself, “react-test-renderer”. This small util package renders React components to pure JavaScript objects, using shallow mount, irrespective of DOM or other environments. This approach allows to unit test the component in isolation from the external systems (DOM) or its children implementation details. [14.]

Another increasingly popular and officially recommended by React team library is “React Testing Library”. It is a lightweight testing library built on top of “react/dom” and “react-dom/test-utils” packages and provides only a full tree rendering to the DOM. The main philosophy of this library is to allow testing components from the user perspective via interacting with DOM. [14;17.] The simple test case algorithm is:

- Find the button by its text
- Click the button
- Check the DOM changes according to the button handler

4.5 Components testing principles

According to testing pyramids to build a robust and well-tested application, it should contain unit, integration, and system (E2E) tests in a different proportion. System testing does not contradict its definition. It walks through the whole application from the user perspective and verifies the efficiency and performance of the tested system. Usually, React system tests runs in the browser environment where special testing tools clicks to application elements to simulate user interaction.

Interpretation of the unit and integration tests for React components can, for the most part, depend on the definition of the unit.

4.5.1 Shallow mount and mount comparison

Tests written with a shallow mount allows verifying component behaviour in isolation from its children. Thereby, tests become faster and more succinctly than with the mount approach.

Another important benefit of using shallow mount is the ability to specify the contract of the tested component. A component's contract defines the expected behaviour of the component and what assumptions are reasonable to have about its usage. React component contract should include:

- Information about obtaining props
- Information about rendering output and passing down props

A well-tested component's contract allows safely refactoring components without harm to related components.

Despite all the advantages, the shallow mount also has some disadvantages. One of those disadvantages is an inability to test components from the user perspective. Since shallow mount returns just a plain JavaScript object, there is no way to verify some interactions between the user and the actual DOM.

Shallow mount pros:

- Specify the contract of the component
- Test component behaviour in isolation from children
- Fast execution

- Easy to write

Shallow mount cons:

- Cannot test component from the user perspective. Since shallow mount returns just a plain JavaScript object, there is no way to verify some interactions between the user and the actual DOM.
- Challenging to keep children component stubs up to date. In case if a component was updated without updating the corresponding test. Stub does not match the component contract.
- Some lifecycle methods and hooks do not work with the shallow mount.

Written with mount tests are closer to the users' behaviour. They allow verifying a component's behaviour from the user perspective. Test all UI interactions and DOM events because it executes all component life cycles. To verify the system's efficiency fewer tests are needed because the mounted component also renders all its children.

This is advantage and disadvantage at the same time If the tested component has a large three of subcomponents, the mount renders the whole tree from the tested component to the bottom and it can become a problem to define which component caused the test failure.

Mount pros

- Verify the system's efficiency with fewer tests
- Execute all component lifecycles
- Test the application components in the way the user would use it

Mount cons:

- Tests are more fragile.
- It is challenging to simulate failing test paths. Sometimes it can be not trivial to simulate a failing test path because it can require additional work in the arranging stage.
- Tests' arrange stage can contain not related to the tested components data and grow up quickly.

- Slow execution because it renders everything and calls all component's lifecycles.

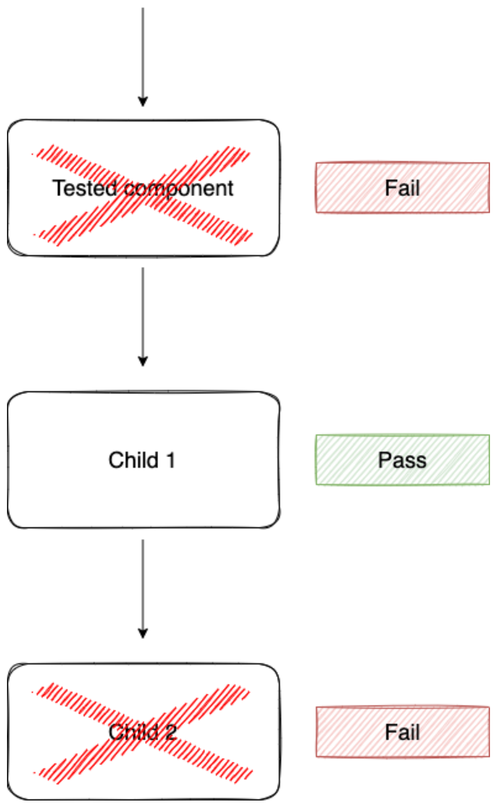

Figure 4. A broken child component causes the parent component to fail.

In figure 4 is shown the case when failure in one of the child components brings the failure of the tested component itself. Such a test is fragile, and it is difficult to debug in a test failure.

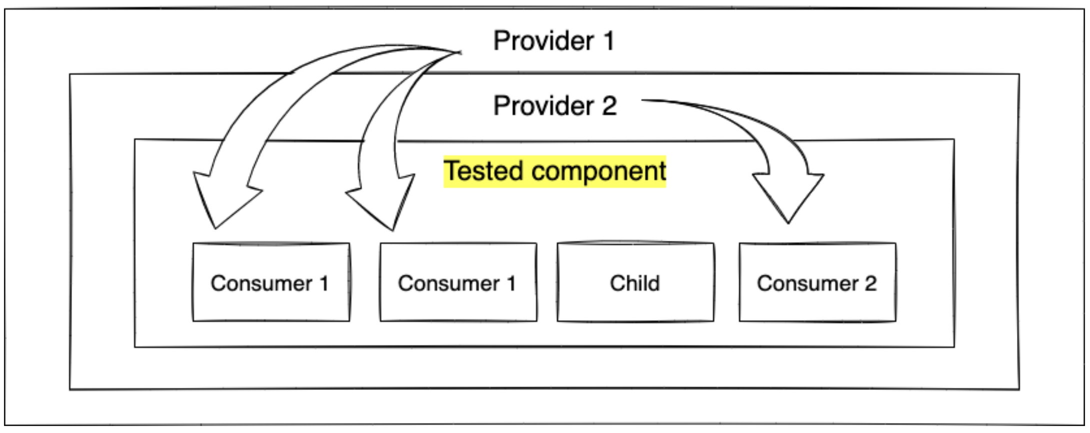

Figure 5.Tested component depends on the dependencies of the child components.

```
const Consumer1 = () => {
  const contextValue = useContext(Context1);
  return <div>{contextValue}</div>
};

const Consumer2 = () => {
  const contextValue = useContext(Context2);
  return <div>{contextValue}</div>
};

const TestedComponent = (props) => {
  return (
    <>
      <Consumer1 />
      <Consumer2 />
    </>
  );
};

const setup = () => {
  return render(
    <Context1.Provider value={contextValue}>
      <Context2.Provider value={contextValue2}>
        <TestedComponent {...props} />
      </Context2.Provider>
    </Context1.Provider>
  );
};

it('renders Tested component', () => {
  const element = setup();
  // Perform act and assertions
});
```

Listing 10. Test setup with unrelated to the target test preparations.

As shown in figure 5 and listing 10 setup function contains non-related to tested component preparations. The TestedComponent is wrapped into multiple Context.Provider components, which have nothing to do with TestedComponent, but are needed for its child components' operability.

There is no clear border between the component unit and integration testing. The definition of these terms is blurry and entirely depend on the developer's views and preferences. Classicists promote the notion that the unit is one specific functional behaviour, and even if it consists of multiple components, it should be tested as one unit. The mockists, in turn, considers one component as one tested unit, which should be isolated from its collaborators, such as the component's children and side effects.

To summarize, the mount and shallow mount are both great tools, but they are used for different purposes. The shallow mount matches all definitions of unit

(See Unit testing on page 9) testing and is perfectly suitable for component unit testing. It verifies the component's behaviour in isolation and specifies its contract. Using shallow mount tests is fast because they do not render the whole underlying component tree and do not translate React elements into DOM representation, which is time-consuming and is the implementation detail.

Component full rendering is more suitable for testing a user's interaction with DOM and components testing in integration, giving more confidence in the general operability of the system.

5 Analysing and improving test patterns of case study React application

5.1 Background

A case study is a part (micro frontend) of an enterprise-level Finnish SaaS platform that automates digital ads production and ad buying at scale. The technical stack is React, Redux, Apollo client (most popular library for working with GraphQL technology), and many lesser-known third-party libraries. The project in total consists of hundreds of components of different complexity and size. To verify that the application works well in different circumstances, it contains hundreds of component tests and few system tests that go through the application and simulate user interactions.

For component testing is mostly used, recommended by React, “React testing library”, which propagates component testing from the end-users’ point of view or via direct interaction with DOM.

After a while of using this approach, the testing execution time, test's sustainability, and developers’ satisfaction have significantly decreased. Test writing for non-trivial components became a daunting task, and sometimes developers just skipped them because it could take even more time than the writing component itself.

During the session where the whole product development team discussed the situation, it became clear that the developers' satisfaction with the test writing process was 4 points from 10. Such a low score meant that some changes were needed in the testing approach.

5.2 Drawbacks of using React Testing Library on scale

The main issue of "React testing library" is that it renders the whole component tree from a root to the bottom, which means it tests the integration between components instead of unit testing. According to the software testing pyramid, ideally, a project should contain approximately 20% of integration tests because they are slower, more brittle and expensive than unit tests. The case study contained more than 80% of integration tests and only 20% of unit tests.

The author of the given work considers this fact as the main culprit. Instead of testing each component (unit) in isolation, hundreds of unwieldy and heavy integration tests were implemented. With the project's growth, the writing of integration tests is getting more complicated because dependencies are also increasing.

To understand the problem, consider the one particular case, a PostForm component consisting of a hundred smaller components such as inputs, labels, text areas and others. The PostForm component is used in two different page components CreatePostPage and EditPostPage. First is the main page, where a new post can be created by filling in all form inputs. The second is the page where the user can edit already existing posts.

```
const CreatePostPage = (props) => {
      ...
      const handleCreatePost = {
      // call 'POST' '/post'
      }
      return(
            <PostForm onSubmit={handleCreatePost} />
      )
}
```

Listing 11. CreatePostPage component.

Listing 11 shows the simplified version of CreatePostPage component which renders PostForm form component as a child.

```
const EditPostModal = (props) => {
      ...
      const initialValues = {
      // call 'GET' '/post/:id'
      }

      return(
            <>
            <PostForm onSubmit={handleCreatePost}
             initialValues={initialValues} />
            </>
      )
}
```

Listing 12. EditPostModal component.

Listing 12 shows the simplified version of EditPostModal page component which renders PostForm form component as a child.

```
const PostForm = ({onSubmit, initialValue = {}}) =>
{
...
return (
      <Form>
            <TitleFormSection title={initialValue.title} />
            <PostContentFormSection content={initialValue.title} />
            ...
            <SubmitPostButton onClick={onSubmit} />
      </Form>
      )
}
```

Listing 13. PostForm component.

In Listing 13 is shown the PostForm component, which consists of smaller subcomponents, which in composition represents a message form.

Testing this example using React testing library (mount approach) mounts all these components, which means it will render and test the PostForm component three times, which is entirely redundant. It is easy to imagine how slow can be tests if PostForm would consist of thousands of components and be used in ten different places.

```
it('test CreatePost component using React Testing Library', () => {
  const element = render(<CreatePost />);
  …
  expect(element.findByPlaceholderText('Add new message')).not.toBeNull();
  expect(element.findByText('Submit message')).not.toBeNull();
});
```

Listing 14. CreatePost component testing using React Testing Library

Listing 14 shows the simplified version of the CreatePost component test using React Testing Library. As shown in the listing to find and check needed elements the developer needs to know about input field placeholder text and submit button label, although they have nothing to do with tested component. If someone will change the placeholder or labels texts, CreatePost test will also fail.

The next colossal flaw is the increasing arrange stage. If the PostForm component uses React context (a way to pass data through the component tree without having to pass props down manually at every level [6].) and Context.Provider component is higher on the tree, then in the test arrange stage PostForm should be wrapped into all provider components that its children use.

Enzyme testing library gives another approach to component testing. It provides the “shallow” function, which performs component shallow rendering. Shallow rendering does not render the child components of a component being tested. The developer does not need to know how the child components have been implemented, only knowledge of these contracts (props they receive) is necessary.

```
it('renders PostForm component', () => {
  const initialValue = {};
  const wrapper = shallow(
    <PostForm onSubmit={onSubmitMock} initialValue={initialValue} />
  );
  expect(wrapper.find(TitleFormSection)).toHaveLength(1);
  expect(wrapper.find(TitleFormSection).props().title).toEqual(
    initialValue.title
  );

  expect(wrapper.find(PostContentFormSection)).toHaveLength(1);
  expect(wrapper.find(TitleFormSection).props().content).toEqual(
    initialValue.content
  );
  expect(wrapper.find(SubmitPostButton)).toHaveLength(1);

expect(wrapper.find(TitleFormSection).props().onClick).toEqual(onSubmitMock);
});
```

Listing 15. Unit test for PostForm component using Enzyme library and shallow mount.

Listing 15 demonstrates the test case, written using the shallow function from the Enzyme library. It verifies that the tested component in this case PostForm renders three other components and checks that correct props are passed down to its children. Meanwhile, the developer should not be aware of the implementation details of lower elements in the tree hierarchy; this makes the test writing process faster and simpler.

Using shallow mount rendering it's easy to test each component in isolation the listing y demonstrates how can be tested each of the above components. Because shallow mount does not render child components there is a confidence that tests aren't indirectly asserting on behaviour of child components.

```
it('renders CreatePost component', () => {
  const wrapper = shallow(<CreatePost createPost={createPostMock} />);

  expect(wrapper.find(PostForm)).toHaveLength(1);
  expect(wrapper.find(PostForm).props().onSubmit).toEqual(createPostMock);
});
```

Listing 16. Unit test for CreatePost component using shallow mount.

```
it('renders EditPostModal component', () => {
  const props = { ..., createPost: createPostMock, initialValues: {} };
  const wrapper = shallow(<CreatePost {...props} />);

  expect(wrapper.find(PostForm)).toHaveLength(1);
  expect(wrapper.find(PostForm).props()).toEqual({
    onSubmit: props.createPost,
    initialValues: props.initialValues,
    ...
  });
});
```

Listing 17. Unit test for EditPostModal component using shallow mount.

In listings 16 and 17 are test suites for CreatePost and EditPostModal components. At this moment, <PostForm /> has been already tested, so there is no needs to do it again. In the above listings, tests verify that the tested components render PostForm component with correct props.

Summarize mount and shallow mount comparison, It is possible to conclude that tests written using shallow mount concept are more succinct, robust and honest. The execution time is much faster because there is no need to render whole component three from the tested component to the bottom. Furthermore, such tests are easy to write and read because developers do not have to understand the whole project hierarchy and dependencies of all child components.

6 Conclusion

Comparing different approaches to testing React applications demonstrates that mainstream and overhyped solutions are not always better than old time-tested ones. Sometimes incorrect usage of testing tools can even lead to development stagnation when application maintenance and adding a new feature become impossible.

This work has been compared the two most popular React component testing approaches, mount and shallow mount. The RTL (mount rendering) can be used for testing MVP and small applications where the fast development time is essential, and the main functionality of the software should be tested with minimum effort.

Another usage of RTL is testing integration between two or more components or interactions with DOM from the user perspective. Using RTL as the only component testing library in medium and large projects can do more harm than good.

A shallow rendering, provided by Enzyme and React Test Renderer libraries, is suitable for unit testing in total isolation from its collaborators. Written with shallow mount tests should be approximately 80% of all tests. Unit tests should be easy to write and understand. They are fast and completely independent from each other. As mentioned in the previous chapter, shallow rendering works well to verify components' contracts and describe their behaviour.

In this thesis work, there was no mention of system testing because of the apparent boundary between system testing and unit/integration testing. It has a precise definition, and usually, it is difficult to do it in the wrong way. Besides, there are excellent tools and frameworks for implementing system testing, such as Cypress, Puppeteer, Selenium.

This thesis focused on exploring the difference of React testing approaches from the developer’s point of view with a practical example.

Furthermore, the benefits of shallow rendering used for unit testing were discussed (or illustrated). It is not feasible to rewrite everything that exists in a project test by using a different approach. Still, it is possible to develop new rules and habits inside the team to follow them in the future. The concrete action point can be a discussion with a development team, or a small workshop based on this work with a detailed explanation of all cons and pros of each approach with its subsequent application in practice.

References

1 Lewis W; Dobbs D; Veerapillai G. 2009. Software testing and continuous quality improvement. Boca Raton: CRC Press.

2 Mitchell JL; Black R. 2015. Advanced software testing. Santa Barbara: Rocky Nook.

3 Software testing - definition, types, methods, approaches. 2021. https://www.softwaretestingmaterial.com/software-testing. Accessed 5 August 2021.

4 Spillner; Slinz T. 2021. Software testing foundations. S.l.: ROCKY NOOK.

5 Shen, J. J. 2019. Software Testing: Techniques, Principles, and Practices. Independently published.

6 Fowler M. The practical test pyramid. Online. martinfowler.com. https://martinfowler.com/articles/practical-test-pyramid.html. Accessed 7 August 2021.

7 Rajkumar. 2019. Unit testing guide. Online. https://www.softwaretestingmaterial.com/unit-testing/. Accessed 7 August 2021.

8 Unit testing tutorial: What is types, tools & test example. Online. https://www.guru99.com/unit-testing-guide.html. Accessed 7 August 2021.

9 Khorikov V. 2020. Unit testing: Principles, practices and patterns. Shelter Island, NY: Manning Publications.

10 Fowler M. 2012. Test coverage. Online. martinfowler.com. https://martinfowler.com/bliki/TestCoverage.html. Accessed 21 August 2021.

11 Koskela L. 2013. Effective unit testing: A guide for java developers. Shelter Island, NY: Manning Publications Co.

12 Thomas MT. 2018. React in action. Shelter Island, NY: Manning Publications.

13 Willoughby J. 2021. The top 5 benefits of react that make life better. Online. Telerik. Online. https://www.telerik.com/blogs/5-benefits-of-reactjs-to-brighten-a-cloudy-day. Accessed 7 August 2021.

14 React docs. Online. https://reactjs.org/docs/getting-started.html. Accessed 14 August 2021.

15 Software testing - definition, types, methods, approaches. 2021. Online. https://www.softwaretestingmaterial.com/software-testing. Accessed 8 August 2021.

16 Jest official web page. Online. Jest Blog RSS. https://jestjs.io/. Accessed 9 August 2021.

17 React testing Library. Online. Testing Library Blog RSS. https://testing-library.com/docs/react-testing-library/intro/. Accessed 17 August 2021.