Testing

Introduction

Test-driven development (TDD) is an overall approach that writes tests before implementing functionalities [1]. This approach has been considered as a primary methodology to sure the quality of the software. There are four major testing phases in this project, unit testing, integration testing, release testing, and acceptance testing. Unit testing is responsible for individual pieces, ensuring the basic functionality of each component. Integration testing checks whether combinations of components work as expected. Release testing tests most of the possible interactions to ensure the stability of the whole system. Acceptance testing checks requirements and specifications by the customer, telling whether the software is accepted. Continuous integration is applied to cover the whole development phase for spotting errors early and improve efficiency.

Unit Testing

As TDD instructs, developers in the team wrote unit tests for most basic software components before coding any actual functionalities. Therefore, unit tests work as the base of the whole project. By doing unit testing, the team has a clearer view of what features a component is expected to achieve. This can be proved by an example of a group in the team. At the early stage, TDD was not taken seriously by some of the team members. One group of two in the team did not follow the instruction of TDD and wrote code directly without writing any unit test. The component displays appropriately at the beginning, but coding is painstaking as no clear plan was made—the group of two modified their design multiple times. After TDD was stressed to be vital, the group added unit tests for the component but found a title in it was wrong. Compared to human eyes and testing manually, automated unit testing helps design the code and prevents potential mistakes by checking components each time they are modified.

Specifically, unit testing in the project works for checking whether a fundamental component contains expected texts, buttons and testing whether functions inside a component run appropriately. Since we use React [2] as the JavaScript library, Jest [3] is the project's primary unit testing tool. React-testing-library [4] is a testing utility that encourages good testing practices and simplifies testing processes such as rendering components and creating snapshots. It is possible to test a combination of several components as well. As fundamental components are already tested, mocking is utilised in testing combinations to avoid repeated tests. Tested basic components and third-party components will be mocked to avoid unnecessary rendering and unexpected errors.

All of the unit tests were firstly planned by documenting test plans in detail. Any failed case and modification were also recorded in a test log for future bug track convenience.

Detailed test plans and logs can be viewed in appendix X.

Examples of unit testing points are as follows:

1. Should contain specific text.

2. Should contain buttons.

3. A function should have been called after a button click event.

4. A subcomponent should have been called while rendering.

5. Functions should work as expected.

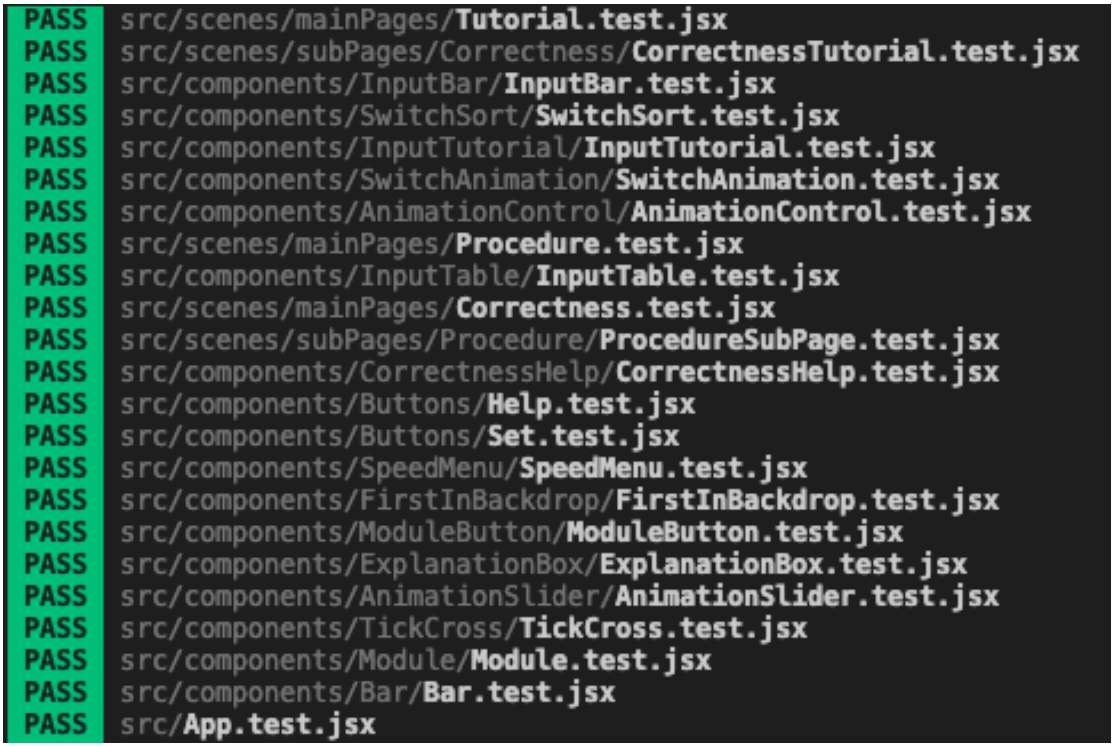

Figure. Unit testing & Integration testing

Integration testing

Integration testing tests subsystems [1]. In this project, scenes and huge combinations of multiple components are considered subsystems. Their interfaces were tested by jest snapshot, and their interactions were tested manually. Snapshot testing is a helpful tool to ensure a subsystem has not been modified. If any of the elements were changed by accident, the snapshot test would fail by comparing it to the old one. Integration testing was often conducted at the end of a sprint and may expose some bugs related to interaction. This is relatively helpful to check whether a subsystem works as the specification expected.

Examples of Integration testing are as follows:

1. Snapshot created and match with old one.

2. Test interactions manually in a subsystem to check them meet the requirement.

Release testing

Release testing is expected to be conducted by an individual quality assurance team that has not been involved in the system development [1]. However, due to the team's small size, all the team members have done something related to the system. In this case, two members who focus more on user interface would take the responsibility of release testing. They tested the software as a whole system manually to check whether the system achieves all the specifications and works without abnormal. Specifically, they took actions to simulate the user stories we defined. Non-functional specifications were tested as well. Once it has been done, the software is ready for acceptance testing and external use.

Three strategies taken are as follows:

1. Performance driven.

2. Specification driven.

3.Scenario driven.

Acceptance testing

THIS PART IS NOT YET DONE

Continuous Integration

Continuous integration (CI) suggests that all code changes will be processed in the mainline of version control to build and test the software automatically [1]. This approach supports TDD well since each submission will be built and tested on the server, which enforces testing and ensures all the tests pass before coding new features. Since all the tests will be run, it could prevent previous work from being broken from new changes, and bugs could be identified quickly. However, those benefits only work when the team strictly follows the instruction of CI. One issue occurred because the team did not pay much attention to CI after the server had been set up. Build and test failed for many submissions while no one resolved them. This resulted in old bugs not being fixed until the team found a series of errors displayed on the CI server. The difficulty level of fixing bugs increased as well. After discussing utilizing CI, the team agreed to fix presenting problems first, before any new changes. The pass icons on the server also kept the team motivated and increased velocity.

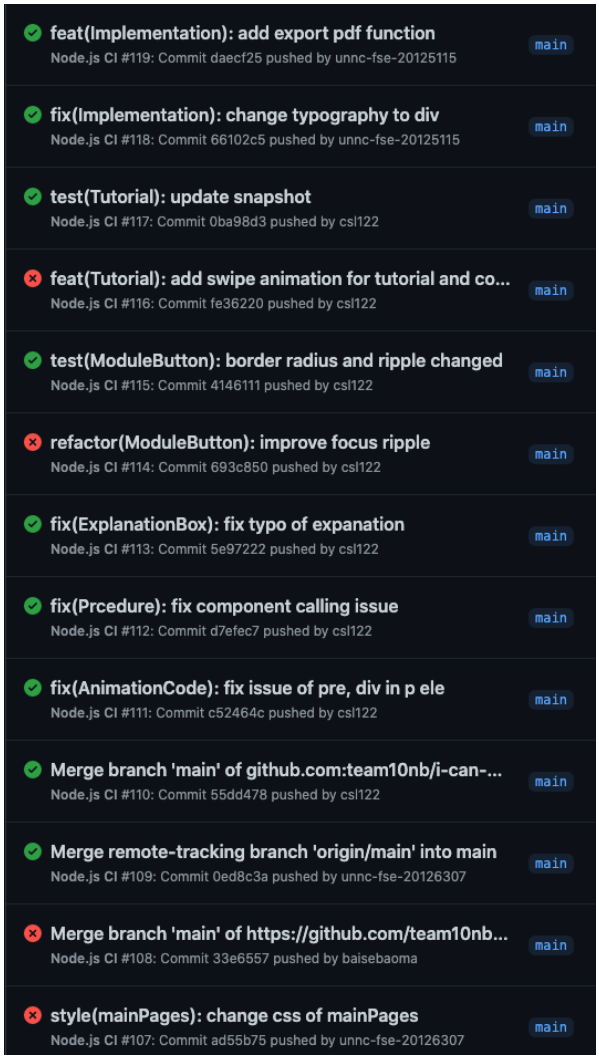

Figure. Continuous integration

References (this would be formalised later)

1. Software engineering 10th
2. https://reactjs.org/
3. https://jestjs.io/
4. https://testing-library.com/docs/react-testing-library/intro/