Testing

Introduction

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

Unit Testing

As what TDD instructs, developers in the team wrote unit tests for most components of the software 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 a contrast in a group of the team. At early stage, TDD was not taken seriously by some of the team members. One group of two developers in the team did not follow the instruction of TDD and wrote code directly without writing any unit test. The component displays properly at the beginning, but coding is painstaking as no clear plan was made. The group of two modified their design for multiple times. After TDD was stressed to be vital, the group added unit test for the component but found a title in it was wrong. Comparing to human eyes and testing manually, automated unit testing not only helps design the code, but also prevents potential mistakes and checks components not being broken by future changes.

Specifically, unit testing in the project works for checking whether a basic component contains expected texts, buttons and testing whether functions inside a component runs properly. Since we use the JavaScript library React, Jest is the major unit testing tool of this project. React-testing-library is a testing utility which encourages good testing practices and simplifies testing process such as rendering components and creating snapshot. It is possible to test a combination of several components as well. As basic ones are already tested, mocking is widely utilised in testing combinations. Tested basic components and third party components will be mocked to avoid unnecessary rendering and unexpected errors.

All of the unit tests were first planned by documenting test plans in detail. Any fail case and modifications were also recorded in a test log for the convenience of future bug track. Detailed test plan and log 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.

Integration testing

Integration testing tests subsystems [1]. In this project, scenes and huge combinations of multiple components are considered as subsystems. Their interfaces were tested by jest snapshot and their interactions were tested manually. Snapshot testing is a useful tool to ensure a subsystem has not been modified. If any of the elements were changed by accident, snapshot test will fail by comparing 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

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 makes sure all the tests pass before coding new features. Since all the tests will be run, it could prevent previous work being broken from new changes, and bugs could be identified quickly as well. However, those benefits only work when the team strictly follows the instruction of CI. One issue occurred is that 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 it. This resulted in old bugs was not fixed until one day the team found a series of errors displayed on CI server. The difficulty level of fixing bugs increased as well. After a discussion on 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.

References

1. Software engineering 10th