Software Testing 2024/5 Portfolio

Max Peerutin - S2264156

January 27, 2025

1 Overview

The Software that will be tested is taken from a tutorial on Youtube by ChrisCourses that I recreated and changed to suit this coursework which can be found here: `https://github.com/chriscourses/multiplayer-game`. The software is a browser based multiplayer game using express, socket.io, node.js and the languages used are Javascript and HTML. The game involves your character shooting projectiles to eliminate other players making your score increase. The link for my GitHub repository can be found here: `https://github.com/peerutinmax/multiplayer-game`

2 Learning Outcome 1

All functional and non-functional requirements are specified in a supporting document provided here[1]. In order to meet the requirements specified, the following testing strategies will be implemented: Functional testing which includes unit tests, integration tests and system tests. Multiplayer games provide the opportunity to implement all levels of testing thoroughly as the game requires the server to interact with the client continuously meaning that integration testing is an extremely fruitful strategy. There are also very many opportunities to implement unit tests as the game has distinct functions such as shooting projectiles, detecting collisions, enforcing boundaries and more. System test's are usually quite qualitative when it comes to games as it is at the top of the testing pyramid however it is simple to test as game's are all about the user experience and therefore non-functional requirements such as smooth game play, satisfying feedback and aesthetically pleasing are all tested simply by playing the game. By employing all levels of testing the system will be thoroughly evaluated and therefore there will be comprehensive coverage. Automated testing frameworks, including Jest for unit and integration testing and Puppeteer for end-to-end testing, will be leveraged to maintain consistency and accuracy across various test cases. With a structured and well-rounded testing approach, the system's robustness, reliability, and scalability will be ensured, resulting in a high-quality multiplayer gaming experience.

3 Learning Outcome 2

The test plan will be provided in a separate document that you can find here[2]. The test plan is designed to be thorough and adaptable. Its comprehensive scope ensures all critical aspects of the game are tested. Clear objectives and schedules facilitate progress tracking, however, the plan remains flexible to allow for potential faults or delays. Add more about the quality of the test plan.

The test instrumentation that was used came from the javascript testing framework, Jest, that I used. Jest has built-in coverage functionality which returns text, JSON and HTML reports which I expanded upon when I implemented the CI/CD pipeline using a tool called Codecov which represents the coverage based off each commit pushed to GitHub. Here is the link to the associated Codecov for this repository[3]. The instrumentation of the code was extremely useful when creating the tests it ensured that quality was kept throughout. Jest provides information of how exhaustive your tests are in terms of statements, branches, functions and lines and provides you with the exact lines that your tests

[1] https://github.com/peerutinmax/multiplayer-game/blob/main/REQUIREMENTS.pdf
[2] https://github.com/peerutinmax/multiplayer-game/blob/main/TestPlan.pdf
[3] https://app.codecov.io/github/peerutinmax/multiplayer-game

were missing. Codecov allow you to see graphically where in your code base the untested areas were. The initial source code obtained from ChrisCourse's was not modular and this was instantly pointed out by the coverage tests provided by Jest and therefore a significant time was spent refactoring the code in order to allow for it to be tested. Therefore, the instrumentation of the code proved extremely useful and benefitted greatly towards ultimately achieving our goals set out in the test plan.

4 Learning Outcome 3

There were comprehensive test suites implemented for this project using techniques to make sure that functional and non-functional requirements were met. A structural based approach was used to exhaust all the the branches for the unit tests as you can see attached below[4]. This was determined to be most suitable for the backend as all calculations and checks were performed by the server. The source code was originally very difficult to test as there was no structure, once the code was refactored and functions such as isProjectileOutOfBounds() and checkProjectileCollisions() were extremely easy to write unit tests for and this resulted in a high coverage for most of the backend, however it was not all easy. The backend also had some integration tests which proved difficult to test according to the structural approach. Another approach used was a Model Based Approached which tracked the creation, connection, gameplay and disconnection of a player. Therefore each step was tested along the way. The player was initialized with a simple class and then sent to the backend where it could get tested to see if all its parts were working. This tested the fact that the player could connect, move, shoot and disconnect. Statement coverage, branch coverage and function coverage were emphasized. Additionally, system testing was incorporated to validate the entire user journey, ensuring that the frontend and backend systems interacted seamlessly. Tools such as Puppeteer were used to simulate real user interactions, including logging in, moving within the game world, and interacting with other players. This provided valuable insights into potential performance bottlenecks and UI/UX issues.

5 Learning Outcome 4

There are unfortunately some gaps and omissions in the testing process and the major one is load and stress testing. Multiplayer game's usually have one major problem and that is hosting thousands of concurrent user's and therefore not being able to test this was unfortunate. The reason for this is time. As previously mentioned, Multiplayer Games involve a lot of moving and integrating modules and therefore testing all the functionality required while hosting the game locally proved costly in time. This was a gross oversight in the test plan. The target coverage was not hit but it was not far off, the target coverage was 70% and 67.94% was obtained. The unit tests had 100% coverage and the server had 78.13% coverage it was the front-end that caused the coverage to drop where only 46.67% coverage was achieved. The frontend had to receive information and commands from the server and send that information to the HTML form and these transitions proved extremely difficult to test. If there was more resources available (time and people) there would be network traffic and load stress simulations. Details of all the coverage can be found attached [5]. If the codebase itself was created with modularity as a major focus higher coverage would have been easier to achieve as a significant amount of the modules rely on other modules and therefore being able to test interlinked segments in a one-to-one fashion would prove useful. If there was more time more focus would be put into security tests to make sure that people connected securely and that every effort to thwart attacks would be made however this would greatly increase the complexity compared to these test suites and the testing was only referring to local hosting compared to an actually deployed multiplayer web game in which security testing would be a major priority.

6 Learning Outcome 5

There are three review criteria that jump out to be the most important in regards to this project. Thorough documentation, maintaining coding standards and modularity. The original source code

[4]https://github.com/peerutinmax/multiplayer-game/tree/main/tests/unit
[5]https://github.com/peerutinmax/multiplayer-game/blob/main/Coverage.png

provided proved incredible difficult to set up tests and standards because there was not a clear structure or consistent practice. Refactoring the code to place maximum emphasis on modularity and documentation allowed for the necessary tests to be implemented. Now that the codebase is consistent with naming conventions and modular, newly hired developers will be able to naturally understand the codebase. Documentation is also extremely important and that is why all functions are commented adequately to aid in showing inexperienced developers how specific functions. If the modular nature and thorough documentation was not enough all the tests that were implemented were included in the CI Pipeline Construction. Utilizing GitHub Actions, the pipeline was configured to trigger on each push and pull request to the repository. The pipeline included steps for setting up the development environment, installing dependencies, running unit, integration and system tests and generating the code coverage reports which were uploaded as previously mentioned to Codecov where the coverage is extremely well laid out for the user to understand where the faults in their code lies. There were some difficulties in implementing the CI pipeline and they came from the fact that each set of tests required a different framework. Jest, using the basic node environment was used for the unit test and these were simple to implement. The main integration test that dealt with the frontend which integrated javascript and HTML was difficult as jsdom was used to create a mock DOM to test on. Finally, the most difficult for implementation in the CI pipeline was Puppeteer which is an API to mimic behaviors of a web browser and this was implemented as a system test to prove that the game ran efficiently and effectively in a web browser. Overall, the CI Pipeline functioned as intended and even went beyond what was necessary to provide automatic code coverage reports. By using code reviews, establishing a robust CI pipeline, and implementing automated testing, the project achieved a high standard of code quality and reliability, essential for delivering a dependable multiplayer gaming experience.