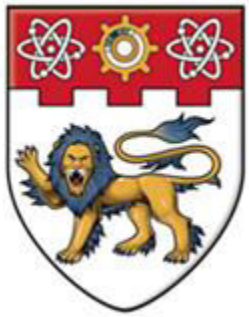

NANYANG TECHNOLOGICAL UNIVERSITY

CZ3003 - Software System Analysis & Design

Testing Strategy

Project Name: Game of Thrones
Group Name: Team TWO
Lab group: TDDP1
Date of Submission: 27/10/2021

Group Member	**Matriculation Number**
Chee Jia Yuan	U1921773A
Chong Jie Sheng	U1920968D
Ernest Ang Cheng Han	U1921310H
Joshua Toh Sheng Jie	U1921471F
Koh Tzi Yong	U1920076C
Lek Zhi Ying	U1922765K
Leong Hao Zhi	U1920469K
Li Zheng Jun, Jefferson	U1922429B
Lim Guo Quan	U1920769A
Loo Yi Ying Phoebe	U1921301A
Remus Neo Keng Long	U1922206F
Tan Wei Lun	U1822950L

Table of Contents

1 Integration Strategy

The Game of Thrones system is made up of 3 parts:

1. Unity engine game server
2. React frontend
3. SQLite and FastAPI backend

which each portion of the system developed independently. Hence, we integrated backend with frontend and game server through FastAPI framework that allowed HTTP requests from frontend and game server, in an asynchronous manner.

2 Overall Testing Strategy

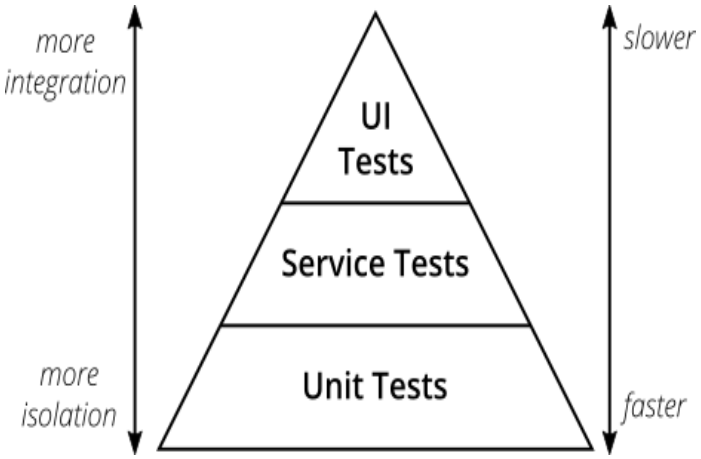

Fig 1.1: Functional Testing Strategy

Fig 1.2 Non-Functional Testing Types

Based on an article written by Martin Fowler, a renown software developer, author and intentional speaker from the United States, we have decided to adopt his muchly used and highly recommended practical testing strategy - the testing pyramid, as the overarching strategy when we are testing and performing quality assurance on both the functional and non-functional requirements of our application. Up the pyramid, parts of the software to be tested decrease in isolation while increase in integration with other components as well as increase in development time and costs.

We will start off at the bottom of the pyramid, performing unit tests on our interfaces, which are the smallest pieces of code that can be isolated from a system. After testing the individual components, classes and methods across our entire project and application, we will move on to service tests. Also referred to as component testing or integration testing, this layer aims to test modules, methods and classes dependent and integrated logically together as a group to carry out a function. After writing test suites to verify and validate components of our application working together to carry out the functional requirements stated previously in our software requirement specification document, we will perform UI tests which aim to test any component related to an interaction triggered on the frontend of any interface by users. It is over at the UI tests layer where we will be performing system testing or e2e (end-to-end) testing given that at this layer, many different components of the application, regardless of whether it is from the frontend or backend are, logically working together to perform a request made by the user. After completing the UI tests, we can move on to non-functional testing

which are categories of software testing not related to the actual functionalities of our application. These include usability testing or usability acceptance testing (UAT) and performance testing which ensures the performance of our interfaces under any load. For our project, we will be performing load testing on our backend and our game interface due to the huge amount of traffic it will most likely received if implemented in a production environment by Nanyang Technological University School of Computer Science and Engineering (NTU SCSE) where as for our administration interface, we will be performing security testing given the sensitive nature and access this platform grants to users, hence making it imperative for us to ensure that this interface is not susceptible to security flaws and cyberattack (e.g. SQL Injections, Cross Site Scripting).

3 Testing Techniques

3.1 White Box Testing Methodology

White box testing is a methodology whereby the software tester is aware and has knowledge of the internal behavior of the system. This includes the various branches, conditions, edge cases, code coverage and any other internal functionalities of the application. White box testing is widely regarded as a low level form of testing which assumes that every logical part and subsequent path that the application can take is known and can be hence tested, which is usually not the case especially when bugs are produced in a production environment in spite of heavy white box testing.

3.2 Black Box Testing Methodology

Black box testing is a methodology whereby the software tester does not have any idea on the internal behavior of the system. Given the nature of black box testing, test cases written are a high level overview of the entire application should behave under different circumstances and situations (i.e. inputs). It also usually involves the perspectives of external stakeholders as well as end users.

3.3 Development techniques

In order to support testing, we have also utilised the various development techniques which will ease and speed up the process of software testing and quality assurance. Below are the 2 techniques we have chosen to implement across all developers regardless of the interface that they are working on:

3.3.1 Behaviour Driven Development (BDD)

According to Agile Alliance, BDD is a development techniques which involves project stakeholders, from the Project Managers, to the Software Testers, to the developers, coming together and discussing the expected behavior and specification of a feature before creating documentation of this in a user/business-focused language, easily understandable and hence implementable by any team member. With BDD, most documentation is formatted and created using the Gherkin's GIVEN-WHEN-THEN language. Using BDD, all team members will be better equipped with the knowledge regarding the expected functionalities and testing knowledge of our Game Of Thrones application, thereby speeding up both the development and testing phase of our software development lifecycle.

3.3.2 Test Driven Development (TDD)

TDD involves first writing unit test cases to validate the behavior and specification of a feature before adding on new features in the form of code into the repository. If the test cases written from TDD fail, then developers will correct the part of the code and hence application causing the failure, before re-running the unit test cases successfully. Once

all of the test suite passes, we can check the code for redundancy and perform any refactoring if necessary, ensuring that our code base is clean and maintainable. Given that we start the development of new features by first writing its unit test cases, it reduces the time spent on otherwise analysing the already implemented feature before being able to write the test suites when developing without TDD.

3.4 Testing Frameworks/Tools

3.4.1 Overview

	Functional Testing	Non-Functional Testing
Unity Game Interface	Unity Testing Framework	Unity Performance Testing Plugin Unity Profiler
React Admin Interface	Jest and Enzyme	OWASP Zed Attack Proxy
Python FastAPI API	PyTest	Cicada Distributed

These are the various tools and frameworks we will be using to execute and write our functional and non-functional test suites, all specific and optimized to the programming language and/or framework we have chosen to build our application

2.4.2 Additional Tools

Git pre-commit hooks to automatically run unit tests on each commit, preventing commits that fail any of the unit tests

3.5 Continuous Integration/Continuous Delivery (CI/CD)

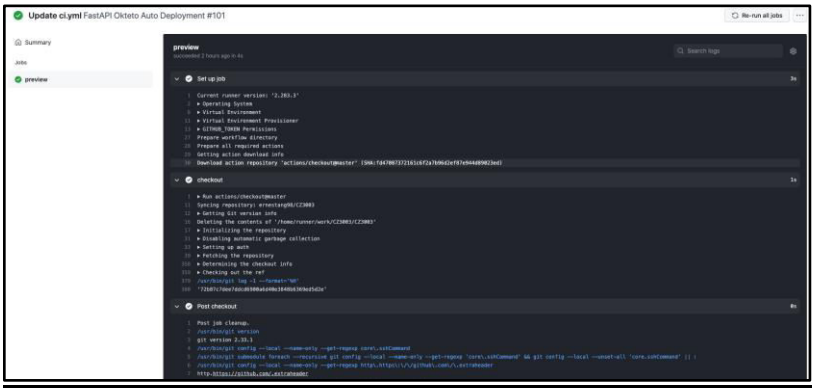

Fig 1.3 Github Actions for CI/CD

Fig 1.4 Pipeline-as-Code

Typically, after softwares are deployed into production on servers or hosting platforms such as Amazon Web Services or Heroku, developers that wish to integrate new features or perhaps commit new changes to fix any bugs surfaced by users would typically run the above mentioned test cases before integrating their code in order to have assurance that their changes will not affect any of the previous functionalities or cause crashes when these updates are pushed into production. CI pipelines can be constructed to aid developers to automatically run written test cases everytime commits are made. Our group used Github Actions to set up our CI pipeline and the building steps of our pipeline are configured using YAML files. In Fig 1.4, our CI pipeline is implemented as shown and every time there is a commit push to our repository, our above written test cases are automatically run and only if every single test case has been cleared and passed by the recent changes or commits, will it be merged and subsequently deployed.

4 Frontend Team

4.1 Testing Strategy

Testing is essential to guarantee a stable application, and we will be focusing on Unit Testing. We perform unit testing because in the React framework, it is built up from individual React components or pure functions. In the case of our frontend code, we used React functional components.

There are several reasons why we used unit testing for our frontend, one is what I already mentioned: React components can be considered units. Secondly, unit testing is prioritised above functional testing because to perform functional testing, we would be required to test multiple components at a time, which would lead to slow progress during the coding process (since we performed pre-commit hooks for automated testing). Thirdly, we want to be able to quickly find the issue within our code, and that is easily done with unit testing since we can pinpoint the component.

4.2 Testing Technique

The technique we have adopted is to use Jest and Enzyme libraries. Jest is a library written by Facebook and it has become popular, and there is plenty of documentation. Jest will be used to do the assertions, and Enzyme will help us to render React components as well as provide us with useful selectors to select and define our components and DOM elements.

There are practices that we adopted when coding our React frontend, and that is to prevent nested React components as much as possible, which allows for simple selectors when writing our test scripts. More specifically, the shallow renderer from Enzyme. We have also separated our functionality and screens into individual components to further increase the usefulness of unit testing — this helps us to find and pinpoint errors quickly.

We have also made use of snapshot testing which makes sure our components render correctly, with no missing DOM elements.

4.3 Automated Testing Tools

To prevent a slow testing process with manual steps, we have made use of the husky JavaScript package. This package allows us to perform git hooks, but more specifically pre-commit hooks. This helps us to identify any rendering or functionality issues with our admin UI before committing, or submitting code for code review with pull requests.

5 Backend Team

5.1 Testing Strategy

For Game of Thrones backend that serves both game and teacher admin website data retrieval, we performed unit testing, which tests all individual functions. It works on the basis of a white-box technique and it was used to verify the correctness of code while developing for easier debugging to save costs.

5.2 Testing Technique

We adopted test driven development using the pytest package provided for automated unit testing. Test cases were scripted to make sure functions can be queried as required before allowing live implementation of the endpoint, which the frontend uses for querying. In the event of errors, stack trace also enabled the identification of errors quicker. Before every commit/pull request on github, all test cases are executed. Practices we adopted include:

1. Isolation of function
    a. This allows us to identify dependencies between functions of code and easier identification of location of bugs through display logs for failed test cases.
2. Frequent and continuous execution of automated test cases

5.3 Automated Testing Tools

- pytest framework is used for unit testing the API endpoints, CRUD operations with our SQLite database, and verifying the response and its body.
- Git pre-commit hooks to automatically run unit tests on each commit, preventing commits that fail any of the unit tests.
- GitHub Actions to automatically run unit tests on pull requests, and during deployment to Oketo, as part of our deployment pipeline.

5.4 Unit Test Cases

Refer to *CZ3003_TeamTWO_System Level Functional Test Cases (Backend).pdf,* in the same folder, for documentation of our unit tests.

6 Game Team

6.1 Testing Strategy

Functional Testing

Functional Testing involves system testing where the integrated system is tested and unit level testing where individual units or components of a software are tested.

Unit testing is used to validate that each unit and function of the game performs as expected. Unit testing allows us to isolate each unit of the system to identify, analyze and fix any defects present.

System testing will also be done when the game is fully integrated, to ensure that the game meets all the functional requirements set.

Non-Functional Testing

Performance testing is a non-functional testing technique that tests for the speed, stability, scalability, response time and reliability of the application under a particular workload. The game will serve mainly the students to answer questions set by their teachers and will be installed as an application on the device. Hence, performance testing will be used to test the robustness of the application to ensure software quality upon deployment.

Specifically, load testing was used to understand the behaviour of the application under anticipated user load. Stress testing helped identify the upper limit capacity (high traffic/ large data processing) of the system to know when the components will start to fail. Testing was executed on Windows 10.

6.2 Testing Technique

Behaviour driven development (BDD) is used to develop test cases and acceptance criteria. They were built based on non-functional requirements elicited in the Software Requirement Specification (SRS). Unity Profiler is run on all scenes in EditMode and further analysis done after observing the CPU/GPU and memory usage breakdown.

- Load Testing

  Right inputs and actions are stimulated during the testing. Test cases are created to measure response time of the functionalities implemented and load time for different scenes. Game profiling is used to determine device resources' usage and render time for each frame.

- Stress Testing

  Application will be tested with higher than expected load, with minimum additional workload stated in test cases. Dummy data will be added into the database to check on data processing speed, while traffic will be introduced by having multiple users run the game simultaneously to check on resource utilization and render quality.

6.3 Automated Testing Tools

- Unity Performance Testing Plugin and Unity Profiler used for scripting load test and measuring load time, response time and utilization.
- Unity Test Runner is used to automate the running of test scripts

6.4 Test Cases

Refer to *CZ3003_TeamTWO_Performance Test Results.docx* for details on results of the performance testing.

Refer to *CZ3003_TeamTWO_Component Design & Implementation.docx* for details on our Game Interface functional test cases.