Test Driven Development Process for "Escapade" - CBL Project Group 176

Huy Hong 1935569

Long Pham 2000954

Introduction

Test-driven development (TDD), developed by Kent Beck, is a design principle for guiding software development (Freeman & Pryce, 2010). It involves writing tests before the code that makes the tests pass before implementing the methods for the actual code.

The TDD cycle consists of following the three main steps repeatedly (Freeman & Pryce, 2010). This is repeated until the final code is finished:

1. **Red:** Write a failing unit test for a new feature or function you want to implement.
2. **Green:** Write the functional code to make the test pass.
3. **Refactor:** Refine the code while ensuring the code stays functional and well-structured.

Firstly, we start with a list of test cases. Pick one test, apply the three steps, and once we are done we continue with the next test and apply the same principle again. If any new tests come to mind, we can add them to the list of tests during the process. Subsequently, we then refactor the code once again and make sure it passes the acceptance tests.

Implementation of the TDD Process

We decided on test-driven development (TDD) and version control as learning goals with version control being our main learning goals. Because these two goals are closely linked to each other, we will explain our process of research on these two topics and how we utilized them while working on the CBL Game Project.

Unit testing

At the beginning of the project, we first decided on implementing **failing acceptance tests** (Freeman & Pryce, 2010). The acceptance tests (i.e. outer feedback loops) would naturally fail at first, in which we would refactor the code for the final product and test it rigorously so that the game passes all acceptance tests. These outer loops focus more on whether the product serves the needs of the player (Freeman & Pryce, 2010).

The acceptance tests are:

1. The **core gameplay mechanics** (e.g. movement, collisions). This is implemented in the file PlayerMovement() and its update history can be seen from the commit history in the Git repository (see References).
2. The **User Interface (UI)** built in Swing (e.g. buttons work as intended).
3. **Game objective** (e.g. the player can complete the stage).

We chose to only do unit testing for the inner loop of the core gameplay mechanics, as the UI and Game Objective do not rely heavily on unit testing. For the acceptance tests, we could use end-to-end testing (E2E) (Freeman & Pryce, 2010). However, a large downside of E2E is that it requires complex environment setups and large time resources, which we found to be rather redundant, given that our game mainly revolves around the player's movement and the collision between the player and other game elements such as walls, obstacles, and monsters. Therefore, we decided to simply use **real user testing** in a controlled environment where we follow specific steps. Additionally, the user for the tests will be us since this allows us to instantly verify the test while instantly obtaining valuable insight and feedback on usability, like when the elements or objectives do not work as intended in the real environment. Furthermore, this also ensures the game is functional in the real environment. Ultimately, we found the cost-to-benefit ratio of real user testing to be worthwhile compared to other testing methods.

For such reasons, we opted for JUnit testing (inner loop) and then the acceptance test (outer loop) for the core gameplay mechanics, while settled with just real user testing (outer loop) for the acceptance test of UI and game objectives.

Version Control

In the next step, we created a separate TDD branch (`tdd`) in our CBL Game Git repository, which allowed us to isolate the TDD process from the main branch (see References). Originally, we planned to write the tests in the TDD branch, then write the code in our respective branches (i.e. `t-h-hong` and `l-m-pham`) and push it to the TDD branch after the code is finished. However, we found that keeping both the tests and code in the same branch (TDD) provided a more seamless and consistent workflow with the TDD cycle.

Timeline of the TDD Process Implementation

We first wrote the unit tests for the player's movement, which included the directions up, down, left, and right. Although we did write separate tests for each direction, we decided to group them into 'one' large test since the logic for the directions is essentially the same. First, all the unit tests fail. We then write the functional code and pass all the tests related to movement directions. Finally, we refactor the code and implement it into the game so that it passes the first acceptance test (core gameplay mechanics), while simultaneously ensuring that the code remains functional and well-structured.

Next, we wrote the unit tests for player interaction (that first fail), which involves checking for collision and death. After writing out the unit tests for player interactions, we realized that we need to add methods that mark the player as dead, then check if the player is dead, make sure that the player

stays still after colliding with the wall, etc. We then create tests for those methods and add them to our list of unit tests. Following that, we write a functional code that passes all of the aforementioned tests in sequential order.

After passing all those tests, we realized that we needed to check for things like the player remains dead even when die() is called multiple times. We then add those tests to the list of tests in the player movement. From there, we once again refactor the code just enough that we can pass the new tests. After sequentially passing all those tests, we refine and clean up the code and implement them into the final draft of the game, while also ensuring that it passes the first acceptance test.

We agreed on testing the acceptance tests towards the end of development, as a lot of things depend on the game mechanics to work first. Before starting, we first set some expectations and objectives for all three acceptance tests (e.g. the input only accepts usernames with alphabetical letters while limiting the length to between 1 to 20 characters). Since we worked on the project ourselves, we can **instantly** pinpoint the issues such as if certain buttons do not work or if the game does not end when the player reaches the goal (denoted by a green square on the stage map when playing the game). We test each acceptance test rigorously, make adjustments and retest if needed. A lot of our real user testing involves verbal communication between the two of us since we found that to be more efficient.

Finally, we simply make sure that all aspects of the game work as intended to pass all three acceptance tests. Additionally, after refactoring for the acceptance tests, the JUnit tests were no longer needed and thus, will not go to the `main` branch, but this can be seen in the `final-draft` branch, which is a branch where we commit files where we think not a lot of changes can be made further, but it won't go in the `main` branch. The `main` branch serves as a fixed branch where we do not change anything that is in the main branch except for initialization of the Git repository and the submission of the Git repository for the CBL Project.

References

Git Repository: `https://github.com/huyhong04/2IP90-CBL-project`

Freeman, S., & Pryce, N. (2010). Growing Object-Oriented Software, Guided by Tests. Pearson Professional.