SEPR 2019/20 Assessment 2

Team CheatCodez

Jonathan Grout, George Lesbirel, Cheuk Wang Wu, Lauren Quarshie, Muaz Atif, Lillian Coultas

Testing Report

Testing methods and approaches

At this stage in the project there are three types of testing we will be doing - unit testing, integration testing and manual testing. In unit testing we will be testing individual functions to see if they perform as intended; these individual functions will make up a larger system. Integration tests allow us to see whether interactions with the database are working as intended. Finally, manual tests allow us to test the game whilst it's running to see if everything works from the user side. For this project we have opted to use JUnit to perform unit tests for the program.

JUnit is a unit testing framework for Java, it is a popular testing framework with wide appeal. Each JUnit test will check one aspect of the code and return whether it passed successfully or if it failed.

To test the game using JUnit, we must use a library called Mockito to run a Headless version of the game with a blank ApplicationListener to emulate the game. We can then run a custom gradle script "gradlew tests:test" to run all our tests.

We are using GitHub workflows to build and test our project automatically. Here we can run our basic test scripts prior to the source building upon pushing to our master branch. This will test the code for major, generic errors and will prevent them from joining the master branch should there be an issue. This way none of the code we test from the master branch will have any obvious errors.

We can test manually by launching the game, and either playing it in a specific way or performing certain actions that will allow us to check a specific aspect, e.g. moving the firetruck near a base to check the water and health both go up. Many of these will test similar things to unit tests, so if they work on both the development and the user side then we can assume they are working correctly.

We wrote JUnit tests to test the movement of the firetruck. However due to compatibility issues between Box2D and Mockito any test that required Box2D did not work (no forces were being applied to objects). Therefore we decided not to include any more Box2D related tests as we knew that they would fail so decided to test these manually.

Test report

Unit tests

Our unit tests provide **41%** class coverage. Report of all tests can be found at https://teamcheatcodez.github.io/project-kroy/testing/

Test Name	Test Description	Test Result	Result Clarification
test_highscore_score()	Tests to see whether a score (200) can be added to the highscores.	Pass	The score is added successfully.
test_highscore_short_name()	This tests to see whether a name less than 10 characters can be added to the highscores.	Pass	The name is added successfully.
test_highscore_long_name()	This tests to see whether a name greater than 10 characters can be added to the highscores. It should only save the first 10 characters.	Pass	Only the first ten characters are saved to the highscores, this passes the test.
test_firetruck_move_forwards()	This tests to see whether the fire truck can move forwards.	Fail	Due to Box2D not working alongside Mockito
test_firetruck_move_backwards()	This tests to see whether the fire truck can move forwards.	Fail	Due to Box2D not working alongside Mockito
test_firetruck_turn_right()	This tests to see whether the firetruck can rotate right (clockwise).	Fail	Due to Box2D not working alongside Mockito
test_firetruck_turn_left()	This tests to see whether the firetruck can rotate left (anti-clockwise).	Fail	Due to Box2D not working alongside Mockito
test_firetruck_can_shoot()	This test to see whether a projectile is fired when the fire truck shoots.	Pass	Firetruck shoots a single projectile which passes the test.
test_firetuck_ammo_decreases()	This test to see whether the fire truck's water decreases when a shot has been fired	Pass	The water lowers after a shot has been fired which passes the test.

test_firetruck_ammo_refill_low()	Tests to see whether the fire tuck can be refilled when on less than full water	Pass	The water refills when low which passes the test.
test_firetruck_ammo_refill_full()	Checks that the fire truck will not be refilled when it is at full ammo	Pass	The water does not increase when at max capacity which passes the test.
test_fortress_shoot()	Checks that the fortress can shoot	Pass	The fortress shoots a projectile which passes the test.

Integration Tests

Test Name	Test Description	Test Result	Result Clarification
DB - retrieve test	This tests to see whether a specific amount of records (we tested two) can be retrieved from the database.	Pass	The test successfully retrieves the required amount of records.
DB - insert	This tests to see whether a new record can successfully be added to the database	Pass	The test successfully manages to add a new record to the database.

Manual Test Summary

All of the manual tests we conducted on the game passed. The things we tested include:

- The main menu buttons (start game and high scores)
- The movement of the firetruck in all directions
- The firetruck's shooting capacities
- That the firetruck can refill at water stations
- That the firetruck can repair itself at water stations
- The alien bases can be destroyed
- Switching between the firetrucks
- Whether firetrucks can be destroyed
- The game can end (win or lose)
- Whether the player can zoom in and out of the game
- The map boundaries

A detailed report on precisely which manual tests were conducted can be found on the website. (https://teamcheatcodez.github.io/project-kroy/supplements/manual_tests.pdf)

We believe these manual tests to be complete as they test every aspect of the game from the user side. Everything that is implemented into the project at this stage which can be accessed from the user side has been tested, which means that the manual testing is complete.