UNIVERSITY OF YORK

DEPARTMENT OF COMPUTER SCIENCE

ENG1 Assessment 2
Group 18 - Octodecimal

Software Testing Report

Group Members:

Izz Abd Aziz

Phil Suwanpimolkul

Tom Loomes

Owen Kilpatrick

Zachary Vickers

Michael Ballantyne

Summarise Testing Methods

We prioritised implementing a high proportion of automated testing, as manual testing expends an unnecessarily large amount of time/resources. Hence our tests were automatically run after every repository push/pull using continuous integration. Advantages of our focus on automated testing include spending less time debugging, having a safety net during refactoring, as well as encouraging the design of modular code with higher cohesion and lower coupling. These automated tests were of small test size (running one process on one machine) as larger tests are costly, complex and more likely to fail. Large tests can also fail with correct code, wasting development time. Overall, small tests use less resources and have higher determinism, so this was most appropriate for our small development team. Our goal was to align our test scope with the Google test pyramid model (80% unit, 15% integration, 5% end-to-end tests). Since our game's structure is simple, we determined integration testing was unnecessary, as we don't have multiple major components to test interactions between. Therefore, we primarily used automated unit tests, with some manual end-to-end testing where necessary.

We used Java's standard unit testing framework JUnit, allowing us to put @Test above a method to allow it to be recognised and run as a test. We ran tests using a type of test double called a fake, LibGDX's headless backend, this meant there were no graphics, which was appropriate for us as tests can run faster. Every test used assertions such as assertTrue to ensure the methods/classes were functioning as intended, these tests were run using a LibGDX test runner created by TomGrill. We used Mockito to make certain UI classes stop having functionality, enabling us to test functions of classes with reliance on other classes which have UI functionality. This was appropriate as our game's code contains a high proportion of logic-based code, not just UI-based code. We used a mix of sources of inspiration to create our tests, using a mix of black-box tests and white-box tests, as we felt both were equally important for this project (we needed to satisfy the client and maximise the game's playability). We ensured adequate verification with white-box tests based on the game's code. We ensured adequate validation with black-box tests based on our requirements.

We used input space partitioning to divide methods' input domains into equivalence classes. Inputs in one class can be treated identically, one input causing a fault means every input in that class will likely cause a fault (and the opposite is true for no fault). Focusing on this allowed us to test all key outcomes in far fewer test cases than random sampling. We had to ensure we considered and tested the boundaries between equivalence classes, as these are likely areas for faults to occur. Our automated tests were created to maximise our structural coverage, focusing on not missing any key elements (ensuring all key requirements are met). This involved focus on maximising statement coverage, branch coverage and mutation coverage, we used JaCoCo test reports to keep track of our test coverage.

Since we couldn't automatically test elements such as UI or sound, we wrote additional end-to-end manual tests. These were necessary as video games are largely reliant on their visual aspects, so we needed to ensure this was working to verify many of our requirements.

Testing Report

By the completion of the project we managed to write 84 automated unit tests across almost all of our classes. For code that resulted in ambiguous results (ones which relied heavily on visuals) we also wrote manual tests. We grouped our tests into testing classes which shadowed their game counterpart i.e Achievement would have an AchievementTest class to support it. The aims, working and results of each testing class is as follows:

AchievementTests: The first test is a testCheckNonexistentAchievement which checks to see if an achievement outside the index range of valid achievements exists in the player's save, if not there it returns false, passing the test. This test runs with no issues and lets us know that checking achievement doesn't return false positives. Next is testGiveAchievement, which checks to see if an achievement isn't given by default, and then gives said achievement by a .giveAchievement() command and checks if it's been added correctly. It also checks to see if adding the same achievement twice or an achievement that doesn't exist doesn't break the running of the game. This test passes. testResetAllAchievements gives each achievement to the player (a previous test showed this command works with no issues) and then uses the resetAllAchievements() command and then checks to see if each achievement is no longer owned by the player. This test runs with no issues. Following tests are for checking each individual achievement is not able to be given twice, not awarded unless achievement requirements are met and tests to see if each achievement is able to be given by commands within the Achievement class. All tests pass and with a statement coverage of 100% by the JaCoCo review tool we can safely assume the tests show us the Achievement class work as intended.

The next tests are in the AssetTests class, these are all asset tests. Each function goes through each folder of assets and just makes sure each file exists. These are here so that if anyone were to accidentally delete or move an asset from the asset folder we would be able to immediately identify which one. All tests run correctly and as all visuals exist in the game, we are not missing any assets. If one were to go, however, the tests would fail.

The next test class is EnergyTests. There are three tests, the first being testSetEnergy. This tests if the .setEnergy() command functions as intended and that a max value of 100 can only be reached no matter how high a number is entered. All tests pass ok so this function is working as intended. Next is the decrease energy test. This tests if decrease Energy is functional and once again will not return a value below 0 for total energy. Test passes so function works as intended. testGetEnergy tests the getEnergy function. Tests pass so functions as intended. This class had a 100% branch coverage score with the JaCoCo tester.

The next is the EventManagerTests class. EventManager is where the vast majority of logical operations happen for the game so EventManagerTests is full of unit tests and automation. The first test is the testTreeEvent which tests if .event("tree") properly adds the tree to your list of achievements and updates your final score. This test passes so this gameplay feature works as intended. Next is testFriendsEventNoEnergy, this checks to see if interacting with an event with no energy means the event will not occur, as shown by time not changing (which always occurs with events). This event passes with no errors showing the key gameplay feature of not being able to interact with events with no energy is working as intended. Next is testFriendsEventWithoutTopic, this checks to see once again that the event will not begin if the player has not selected a topic of discussion, no test errors, working as intended. Next is testFriendsEventWithTopic, which tests that time passes and energy is used (showing the event takes place) when interacting with friends with a topic chosen. This test passes with no issue,

so this function is working as intended. The check event with no energy and energy theme of tests is continued for events : Ron Cooke hub, Piazza, walk, bar and ducks. Much in the same way as before for the friends event, all tests pass which shows the features are working as intended. This class was able to be tested with 67% coverage on the JaCoCo coverage reporter.

The next testing class in the GameObjectTest class. GameObject in this game is used to store positions and dimensions of Tilemap properties and so testing is as follows: the testPut test tests to see if adding a key value pair to the gameObject is possible. Test passes with no issues. testGet tests to see if the value of a key value pair can be obtained with only the key. Test passes with no issues. containKey does similar but finds the key form the value, test passes. testCentreX/Y tests to see .setCentreX/Y method works. Tests pass with no issues. Finally testSuper tests to see if constructing the gameObject is able to be done with no issues. Test passes. Overall the class has a 49% JaCoCo coverage report.

GDX test runner is a class we did not create but is necessary code for the testing framework we are using.

Next is HustleGameTests, this class contains a large amount of subjective outputs so manual tests were mostly employed for it. However we are able to unit test the file reading aspect of this class. In testReadTextFile, we test that the class is able to correctly open the right file it needs to and returns the proper text it should. Test passes with no errors so we assume the class can read its text file. Next is testReadTextFileNonexistentFile, this returns an error message when the given text file can not be found. Test runs without errors so in the case our text file is missing/misnamed we are able to determine this from tests. This class has 100% JaCoCo branch coverage, but only 18% statement coverage, as most of this section is covered by manual tests.

Next is LeaderboardTests, the first test, testReturnFileName tests if the leaderboard is in fact using the file path we assigned to it and storing/reading data from the correct place. Test passes so we can assume it is. Next is testScoresStoredAndReturnedCorrectly which tests that saving scores is possible to the file, and properly orders the leaderboard based on score size. Tests pass so we can assume this is a functional part of Leaderboard. testOnlyTopTenScoresKept does just that, tests to see that if the player in 10th gets knocked to 11th they'll no longer appear on the leaderboard as their position has been usurped. It also checks to make sure only 10 players are ever shown. Test passes so we can assume the leaderboard is fully functional. This class has a 97% statement coverage on the JaCoCo coverage report.

Next is NonPlayableCharacterTests. Most of the NPC class relies on screens and animations, and while it’s useful for a unit test to say something displays/formats correctly, it’s more visual so it is more appropriate for it to be covered more by manual testing. Still, the unit testing done in NPC is done as follows: CheckX/Y/Dir which checks the getX/Y/Direction() methods respectively. Each of these returns true so we know that checking different aspects of an NPC is working as intended. We also have testGetCurrentFrame which tests getCurrentFrame(), it passes so we know this method can be relied upon when it’s called later for animation specific code. checkSetters also checks if we are able to set the direction of an NPC which we are able to do so as the test passes. This class has a 0% coverage from the JaCoCo coverage report, however this is expected as key aspects of the code are identified to function properly and other more subjective tests are handled by manual.

Next is PlayerTests, first test is testSetPos, this checks player’s position after using Players inbuilt commands to move players location. This test passes so the player is able to be reliably moved, next is tests setFrozen and setUnfrozen. These tests pass so the player is able to be location locked when in

menus/talking events. The following tests add colliders to the player and set up various scenarios where a player may be out of bounds and checks to see if the game corrects the players position in these situations. In all axis of movement, these tests pass so we know a player may not be able to leave bounds and can reliably have its position and collider updated. Key functionality we know the player needs and is shown to have. This class has an 80% JaCoCo coverage report.

Next is ScoreTests, with the first test being testAllActivitiesExist. All keys relating to activity I.D 1-5 are present and accounted for and the test passes accordingly. testNonExist also checks that the game is not derailed when an activity code that doesn't exist is used by .containsKey(). This test passes so score manipulation based off of activity should encounter no obstacles. testAddAndReadScore tests that .incrementTotalScore() and .getScoreByActivity() .getTotalScore() all function as intended. Tests pass so they do. testResetScore also proves resetScores() functions as intended so the key tools needed for keeping track of score are shown by these tests to be fully functional. This class has an 80% coverage on the JaCoCo coverage report.

Next is SoundManagerTests, testGetVolume and testSetVolume for Music and SFX are used to check that volume is a readable and alterable value by only using SoundManager commands, tests pass so we have proven this to be true. The remaining tests are used to test that the footstep sound is not able to be played constantly. It ensures that through the use of footStepTimers and processTimers which ensure enough delay is presented between each playing of a footstep sound. These tests pass, however while we can prove delay takes place between each footstep we cannot assume this results in a streamlined experience for the player so we used manual tests to ensure this feature is present in its full capacity. This class has a 67% Jacoco coverage report score.

Finally is the TimeTests, half of which are format tests. testFormat and testGetMeal test that correctly formatted text is returned on a time input, a digital clock time reading and a meal which is eaten at the time of day the player is in. Tests pass so we can know that these methods function as expected however we cannot assume they are displayed correctly for the player and must resort to manual testing to see if the dialogue boxes present the (correctly) formatted time properly. Next is testPassTime which checks that time can be added to a day through the .passTime() method and that after a point the time is reset to 0 and the day number is incremented. This test passes so it should function as expected in game. Then it's down to testGetters which tests various get() methods within Time. All pass so we can assume all other classes will have no issue interacting with the Time class. This class has an 84% Jacoco coverage report score.

Overall we received a Jacoco coverage report score of 30% for structural coverage and 37% for branch coverage. Most classes related to screens and visuals were not unit tested as things like audio/visuals are subjective, however we believe anything that was possible to unit test has been done. Our end-to-end manual test checks all audio, u.i and visual aspects of the game are working as intended, this was done as certain classes (such as screens) cannot be initialised with the headless backend.

URLs for Testing Material

Our most recent testing results can be found here, these are generated by dorny test reporter:
https://github.com/eng1team18/HesHustle2/actions/runs/9198238231/job/25300561343

Our most recent JaCoCo coverage reports can be found here, under “Workflow Artifacts”, note that there is one for each OS as explained in the Continuous Integration Report:
https://github.com/eng1team18/HesHustle2/actions/runs/9198238231

Further information on our manual test-cases (including further description of the process to create and run through them) can be found here:
https://eng1team18.github.io/website-2/Assets/Assessment%202/Manual%20Tests.pdf