Testing Report

Cohort 3 Group 6 - Carbon Goose

Members: Bailey Horsley, Owen Jones, Rory Ingram, Ken Jacob, Abishek Kingslin Raj, Louis Polwarth, Adam Johnson

a) Briefly summarise your testing method(s) and approach(es), explaining why these are appropriate for the project. (4 marks, ≤ 1 page)

Both unit and manual tests we employed in the testing of our game. Integration tests were not used as large parts of the inherited code from the previous group had limited separation between back end processing and the front end and . This meant that tests like that would have relied on automated UI that would have both been fragile and required significant development time we did not have access to. Changes to the architecture and code of the game that would have been required to facilitate these tests would have required time or resources out of the scope of the project. Unit tests were the primary method of testing used as it gives us the best way of testing the functional completeness and correctness of individual methods. Manual tests were primarily employed for requirements testing and to test parts of the code that couldn't be tested automatically.

The primary testing approach used was input space partitioning, this was primarily applied to black box unit tests. This was used because it allows us to test the full range of expected value easily. White box testing was also used alongside a fake version of "GameScreen.java" as the game event logic is largely implemented through "GameScreen.java" which cannot run on headless mode due to repeated use of the render() method and other similar methods.

b) Give a brief report on the actual tests, including statistics of what tests were run and what results were achieved, with a clear statement of any tests that are failed by the current implementation. If some tests failed, explain why these do not or cannot be passed and comment on what is needed to enable all tests to be passed. If no tests failed, comment on the completeness and correctness of your tests instead (10 marks, ≤ 3 pages).

In total 93 unit tests are run. We had an overall coverage of 44% and a missed branch rate of 49%. Largely we are happy with those numbers as the files we actually wrote tests for mostly sit at above 90% in both fields. Large amounts of the untested code sections represent things like UI features, camera control and audio control.

While we endeavoured to have our test's coverage of the tested classes and method be 100%, in many cases this could not be achieved. One example of this is in "LeaderboardTests", where due to the use of libgdx preferences to store the leaderboard, there is no way to test the "loadScore" and "saveScore" methods. The "addScore" function also required a refactor that added dependency injection to allow it to be testable. Another example is found in "TimerTests" where the "Poll" method that calls game events which has no test as we could not find a reasonable way of testing if a nonspecific "event" has take place and using a specific "event", due to them being functionally their own pseudo methods causes too high a level of brittleness.

For the testing of the building and the satisfaction systems, a testing version of the map json file that only had grass tiles was created. This was both used to ensure that devoid of all other features an empty map would have a satisfaction of zero and to avoid adding brittleness to the tests as any changes to the base map could lead to the relatively complex world setups used by the test starting to fail either by the configurations no longer being valid to place or base map features added that could change the output score.

For the testing of the satisfaction system we also had to create an additional new assert, "assertClose", to allow the comparison of two HashSets of floats while still allowing a delta within the individual floats.

The timer tests do have some minor issues with correctness. In its original version we had a timer of 1000 milliseconds and waited 100, the deltas for this version were 7 when running the tests individually and locally, an error margin of 7%. When the tests were run locally as part of a full run of all tests, both timer tests failed. When the delta and therefore the error margin was increased to 10 milliseconds the tests returned to passing. We believe that this is due to the timer system using system time as a base and so small things like context switches, other processes using the core after a wait and the processing time used to go in and out of thread sleep and to run the getTimeLeftmethod cause little bits of additional time to pass. This is supported by the fact that when the tests were run on Github they started to fail again, the amount by which the the tests on github failed was between 30 something milliseconds to over 130 milliseconds, the latter having an error margin greater than 100%. This supports my idea as it's likely that a server would

have greater additional processing time added due to higher demand for cores. Our solution this time was to increase the overall timer, wait times and deltas. We increased the timer to 100000 milliseconds, the wait time to 10000 milliseconds and the delta to 200 milliseconds. This gave us a margin of error of 2%, a significant decrease, it does however increase the real error from 0.01 seconds to 0.2 seconds.

The asset tests have a level of problematic fragileness. This exists because large parts of the game references assets in a way that is difficult or impossible to retrieve, for example the “BuildingType” enum loads as a “Texture” libgdx class directly instead of storing the file path and calling that later, this means there is no way to access the file path directly from code as the “Texture” libgdx class does not have a way to get the file path parameter back out of it. We have instead had to directly reference the file address, this means that if any file names are changed the tests will start to fail if they are not updated

There are some other issues with completeness in the unit tests: the satisfaction tests do not cover the branches for when the satisfaction goes down after a change; eventHander does not directly get tested, any of its coverage is gained by events being used in other tests and the calling of an undefined event is never tested; Vector2Int is also never directly tested, this largely happened due to time restrictions, the relative simplicity of its methods and the fact that many of its methods are never actually used; and in the game event code large parts of the ongoing code is left largely untested such as “tryForGameEvent”, “tickOngoingEvents” and many game event’s “ongoingEffect” methods. Beyond those examples and the areas explained in greater detail earlier in the report we are largely happy with the completeness.

We also had three manual tests that mostly exist for requirements testing. “Settings, timer and pause test” and “Build, move and achievements test” we are happy with the completeness and correctness of as everything they test is simply verifying that when you take a specific action a specific result occurs. “Leaderboard and playability test” on the other hand does have issues with correctness. Its leaderboard testing aspects are correct and help make up for not being able to test “loadScore” and “saveScore” but it’s playability tests measures 6 attributes of the game from the requirements all of which are both nebulous and subjective and so significantly reduce the correctness of the test. That is however an inherent issue with testing requirements of that nature. There is also an issue of completeness in the manual tests as they do not cover all requirements, only the ones that were easy to test quickly.

c) Provide the precise URLs for the testing material on the website: this material should comprise the testing results and coverage report generated by your automated testing tooling, and descriptions of manual test-cases that you designed to test the parts of the code that could not be covered by your automated tests (4 marks)

Found at this link downloads under “software testing”
https://carbongoose.vercel.app/