Testing

Team 5:
Kevin, Zuhur, Erin, Jack, Aidan, Ryan, Sam

General methods and testing approaches

Our approach for this game prioritised testing the game's core logic over rendering capabilities through a combination of automated and manual testing. We wrote an extensive test plan covering everything we needed to test, and followed this when conducting the final round of testing. This can be found on the website.

We used the JUnit framework to conduct our automated testing. The test results were generated by our Gradle project and we used JaCoCo for measuring our code coverage. We decided to use unit testing with the help of the JUnit framework so that we could test individual components. Our unit tests focus on the core gameplay mechanics such as timing, scoring, points and more. We wrote tests that allowed us to test isolated parts of code like the Timer classes and Score classes. We felt this method was appropriate because unit tests help enforce modularity and make it easier to show traceability since each requirement has a corresponding unit test. For instance, the FR_TIME_LIMIT requirement was tested using TimerTests.java to ensure an accurate countdown. An overview of the automated tests, including both the unit and integrated tests can be found below, along with results and analysis.

While automated tests provided extensive code coverage for the game's logic, there were still parts of our game that required manual testing such as navigating the game menu and responsiveness of the UI. For instance, FR_USER_INTERFACE could not be automatically tested to ensure that the game's UI wasn't cluttered and that all the buttons and input fields work. We tested these elements following a test plan, to ensure maximum coverage. This can be found covered in detail in the Manual Tests document on the website.

To summarise our testing approach, we used automated testing for ensuring correctness of core gameplay logic and manual testing for the more visual and interactive parts of the game.

Report of Testing

Automated Testing

As discussed in our testing plan we aimed to use automated testing as much as possible. We achieved this by using the JUnit framework to write our automated tests.

Unit Tests

We first focused on unit testing for smaller parts of the code base. In the end we had 5 key unit tests that aimed to automatically test our implementation for the following requirements: UR_STUDENT_SATISFACTION, FR_SCORE, UR_GAME_PROGRESS, FR_TIMER, FR_MAP, FR_BUILDING. A description of each of the testing classes is as follows:

AssetTests.java (FR_MAP & FR_BUILDING)
This is a unit test that ensures the presence of all graphical assets used in the game. It validates the assets.txt file by checking that each file listed exists in the project directory. If a file is missing or the path is invalid, the test fails, raising an assertion error and preventing the game from being built with incomplete assets.

TimerTests.java (UR_GAME_PROGRESS & FR_TIMER)
This tests the core methods of the Timer class (tick, hasFinished and reset). The tickTest() method checks whether the Timer shows the right time after 10 seconds have passed in the game. Each tick deletes a set amount of time from the timer. The updated timer is compared with a string of what the time is supposed to be. The hasFinishedTest() method checks if the timer indicates whether it still has time or has run out of time. resetTest() checks that timer can be brought back to its original state after being modified. Lastly, getTimeAsFloatTest() method checks whether time is returned as a float.

ScoreTests.java (UR_STUDENT_SATISFACTION & FR_SCORE)
This includes tests for validating the Score class. The initialScoreTest() method checks that the score is initialised correctly at the start of the game, and incrementScoreTest() method ensures score updates are accurate, including edge cases like exceeding 100% or negative increments. The resetScoreTest() method verifies that the score resets to its initial value, and the getScoreStringTest() method confirms the score's string representation is correct. The initialScoreValueTest() method tests whether the Score class correctly handles initialisation with different values. If a score is too high (like 500), it caps the Score at 100 while if the score is negative, it resets to zero.

PointTests.java (FR_BUILDING)
This tests the Point class which represents a point in a 2D space. The pointEqualsTest() method ensures the game correctly compares two points to see if they are the same and ignores invalid comparisons. The pointToStringTest() method checks that the point's coordinates are displayed in the correct string representation, eg. (x, y). Finally, the getNewPointTest() method confirms that a new deep copy of a point with the same values is created when requested.

SettingsTests.java
This contains 1 test to check if the string value the user sets their username to is saved correctly.

Integration Tests

Also following our plan we used integration testing for requirements UR_EVENTS, UR_ACHIEVEMENTS and FR_ACHIEVEMENTS. We created the following test classes for these requirements:

Test 5: AchievementTests.java
These tests verify the achievement system within the game. The initTest() method ensures that achievements are initialized correctly by mocking various components of the LibGDX framework such as graphics and file systems, so that tests can run without actually rendering anything. The achievTest() method creates a mock game world and checks an achievement's state after simulating in game actions. The achievManagerTest() method tests the Achievement Manager to confirm that it can handle and track multiple achievements accurately.

Test 6: EventTests.java
The EventTests class has two tests that test the behaviour of the Event and EventManager classes in the game. The initTest() method, similar to the one in AchievementTests.java, sets up mock components like a mock world and a mock file system, for testing functionality of Events without requiring a display or file system. The eventManagerTest() method deals with the EventManager class and creates a mock world that cycles through events and ensures that two consecutive events aren't the same. The eventTest() method tests the Event class by creating two types of events - a "ticking" event and a "building" event, and by doing so, it checks event types, the properties of buildings and also the behaviour of other methods such as tick().

Results and Summary

In summary we created these automated tests by following our original plan and didn't have to make any significant changes. We started by implementing our planned tests and then improved using test reports after each implementation. An example of an improvement is our tests for our settings class. We initially didn't plan for this but as we implemented the leaderboard we had to add usernames and so we realised tests for these would be needed.

Looking at the test report produced on our latest build, and specifically our unit tests, (this can be seen HERE & HERE) we had no failed unit tests and coverage we were happy with. A lot of the missed instructions picked up by JaCoCo weren't tested as they would either be tested later in our manual testing or they weren't able to be tested. Also looking at our Gradle test results which can be found on our website, HERE, we can see that these unit tests were all completed in a small amount of time proving why they were appropriate to be unit tests.

Looking at the results of our integration tests in our coverage report (found HERE & HERE) and Gradle test results we can see that these tests took more time to run but all succeeded. Again there were some parts of the code that weren't tested. Some of these will be covered in manual testing and some of these it wasn't logical to test. For example the eventTick() method is missed in our EventManager but this must be working for our "ticking events" test to pass.

Manual Testing

When it came to manual testing we planned to cover anything that wasn't tested through our automated tests. We started with a general plan focusing on tests for requirements that wouldn't be covered with automated tests. Most of these involved the user experience i.e, UI displaying the correct information and user input responding correctly. This was thought to be the best option as it was reliant on the game being rendered (increasing complexity for automated testing) or the requirement was qualitative (e.g testing if the game was easy to navigate) We also made a slight modification to the game to ensure testing was consistent, this was ensuring the first event was always the same.

In order to keep track of our manual testing we created a tracking sheet which can be found on our website, HERE. This outlines each test with its own ID, what steps the tester will follow, the expected outcome, the actual outcome and if the test passed/failed. Following our test plan we broke each major test ID into individual tests that would cover anything missed by our automated tests. This left us with no user or functional requirements untested.

Results and Summary

Looking at our tracking sheet we can see that for our latest build all tests passed (any test that has no result recorded would have been completed in a previous test).

During the development of our game our tests evolved many times. Through exploratory testing we revealed tests that had been missed and improvements to our methods. Some examples of these are:

- Revealing the need to test how the project behaved such as saving of settings and resetting the game to its initial state
- Missed tests from our original plan like our "settings tests"
- Moving tests that were reliant on rendering to manual testing

Completeness and correctness

All 18 unit tests passed on the final release of the game, validating the core functionality of the game. All manual tests also passed. However, while reasonable code coverage was achieved, some gaps remain in navigation testing, and scoring algorithm validation, leaving the accuracy of the algorithm unverified. Despite these limitations, the overall testing results provide a solid level of confidence in the quality of the game.