Software Testing

Cohort 3 - Group 4
AJAJARA

Adam Fuller
James Goude
Abbie Spears
Rosie Sherwood
Artem Frolov
Alexander Gibson

Software testing methods

We have utilised unit, integration and manual testing to identify bugs in our code, so that they could be fixed before becoming part of a release of the game. These tests are run automatically on a push to GitHub, with test results and code coverage reports generated by JaCoCo and then added as a build artifact.

Our unit tests are designed to test the functionality of specific classes, or functions of classes. [1] These are used to ensure that individual components of our game are working as expected, which can help us catch bugs without needing to play through the game, saving us time.

Integration testing is designed to test that different components of the game can work together. [1] These tests will typically be larger, and span multiple packages within our codebase. For example, we might test that when a building is selected, a timer is registered.

Finally, we used manual testing to test UI elements and interactions. We can't automate these within the build pipeline because our tests are being run on GitHub Actions, within headless Linux instances. LibGDX encourages merging logic and GUI code with Scene2D, which we have used for most of our GUI. Scene2D relies on Stages, which need a SpriteBatch to operate. However, in a headless environment, we cannot create any ShaderPrograms, which a SpriteBatch requires. Therefore, whenever we instantiate a class that contains Scene2D elements, an IllegalArgumentException will occur. This doesn't stop our tests from proceeding, because it is an unchecked exception, but it does mean we cannot run any UI tests in code or launch our game's GUI.

Although we have 44 tests (all of which pass), because the UI classes inside of the screen package (which we can't test) tend to be our largest classes, we have a relatively low code coverage percentage of only 35%. However, as you will see from the code coverage report, we have thoroughly tested everything that is not UI-related. This lack of UI testing is offset by our manual testing procedures, which can be found here:
https://eng1g4.github.io/a2-website/documents/SoftwareTesting/ManualTestCases.pdf

The test report generated by JaCoCo can be found here:
https://eng1g4.github.io/a2-website/documents/SoftwareTesting/reports/tests/test/index.html
The code coverage report generated by JaCoCo can be found here:
https://eng1g4.github.io/a2-website/documents/SoftwareTesting/reports/jacoco/test/html/index.html

Tests report

Test	Type	Requirement	Description	Pass
testAssetsExist	Unit	N/A	Test that all of the assets required for our game exist	Pass
testBuildingCost	Unit	FR_CURRENCY	Test that each building type is instantiated with the correct cost	Pass
testBuildingCurrecnyGenerated	Uni	FR_CURRENCY	Test that each building type is instantiated with the correct currency generated	Pass
testSetMapPosition	Unit	FR_LOCATIONS_PLACEABILITY	Test that building is set to the correct position	Pass
testDistance	Unit	FR_SATISFACTTION_VARIABLES	Test that the correct distance is given when calculating the distance between two buildings	Pass
testBuildingCollision	Unit	FR_LOCATIONS_PLACEABILITY	Check that there is a collision with another building when placing.	Pass
testTerrainCollision	Unit	FR_LOCATIONS_PLACEABILITY	Test for collisions with terrain when placing a building	Pass
testOutOfBouds	Unit	FR_LOCATIONS_PLACEABILITY	Ensure that you cannot place buildings outside of the map boundaries	Pass
testBuildingCreation	Unit	UR_LOCATIONS	Checks that buildings of	Pass

			every type can be instantiated	
testPlaceBuilding	Unit	UR_LOCATIONS	Tests that buildings can be placed, and that the BuildingPlacementManager keeps track of those buildings.	Pass
testRemoveBuilding	Unit	FR_LOCATIONS_REMOVE	Tests that a building can be removed once it has been placed, and the size and contents of placed buildings list after removal.	Pass
testReset	Unit	FR_RESET	Tests that the placed buildings list is cleared when the BuildingPlacementManager is reset	Pass
testBuildingManagerPlace	Integration	UR_LOCATIONS	Tests that buildings are successfully placed with BuildingManager class and is kept equivalent with BuildingPlacementManger	Pass
testBuildingManagerRemove	Unit	FR_LOCATIONS_REMOVE	Tests that a building can be removed once it has been placed.	Pass
testBuildingManagerReset	Unit	FR_RESET	Tests that the placed building count associated variables are successful reset	Pass

			to null and zero values	
testSpriteCreation	Unit	UR_LOCATIONS_SIZES	Tests that sprites for each building can be loaded	Pass
testEventTimers	Unit	UR_EVENTS	Tests that event timers wait the desired time, and can be stopped and started.	Pass
testBuildingTimers	Unit	N/A	Tests that building timers tick 3 times.	Pass
testGameTimer	Unit	FR_TIMER_DISPLAYER UR_TIMER UR_TIME_TRACKER	Tests that the GameTimer can keep track of months and years	Pass
testTimerManager	Unit	UR_PAUSE	Tests that the TimerManager can register, start, stop and remove timers	Pass
testBlankSatisfactionGraph	Unit	FR_SATISFACTION_VARIABLES	Checks that the initial satisfaction score is 0	Pass
testSingleBuildingType	Unit	FR_SATISFACTION_VARIABLES	Checks that the satisfaction score is always 0 when one type of building is placed	Pass
basicSatisfactionTest	Unit	FR_SATISFACTION_VARIABBLES	Test that two buildings placed in the same place always give the same satisfaction score	Pass

We have another 13 test cases which are described in this additional document on our website:
https://eng1g4.github.io/a2-website/documents/SoftwareTesting/ExtendedTestReport.pdf

References

[1] Atlassian, ‘The different types of testing in software’, Atlassian. Accessed: Jan. 13, 2025. [Online]. Available: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing