# Testing 2

---

Cohort 1, Team 4:

Robert Watts
Adam Wiegand
Josh Hall
Travis Gaffney
Xiaoyu Zou
Bogdan Lescinschi

Throughout the project we determined that we needed to test the product regularly to make sure that the final release completed all of the specified requirements. The first part of the testing was the white box tests. These tended to be automated tests using Junit and Mockito. Each test was broken up into a class and we made extensive use of the @Before tag to set each test up in its own way. We set up the continuous integration to run these tests on every pull request.

Obviously we needed to be able to use LibGDX as a headless mode to be able to test. To do this we are using a class (based off https://github.com/TomGrill/gdx-testing) to start a headless session at the beginning of each test class. Using this helper class allowed us to test things such as the file structure, and graphics without loading a full window. We also used a module called Mockito that allowed us to mock particular classes, methods and variables. This meant that throughout the tests we could mock certain functions without the need to “render” or “load” them.

To achieve the black box testing we moved all of the project code into a folder named “main" in the core src folder. From there we were able to create a new folder called “test” which contains all the unit tests. We then adjusted the Gradle build file to reflect this change.

Our aim with the white box testing is to build a set of tests and get the code coverage of them. As we added new features we would create their tests, aiming to keep the code coverage at the same level or higher. The reason for this is that we wanted to write meaningful tests, and not just write tests to get the code coverage up. Our overall aim was to have a code coverage level of between 40 - 50%.

The second part of the testing was the black box testing. While normally white box testing is normally completed by a software tester without knowing how the code works, we elected to keep the testing in our team with the odd test being run in conjunction with our customer (during our team customer meeting). We did this to keep it simple and to be able to rapidly prototype and run the tests. As shown below it did have an effect on some tests. We used the black box tests for ui behavior and gameplay tests, as opposed to unit test specific pieces of code. The aim with the white testing was to test some of the requirements that we did not test.

The White Box test table is available here:
https://eng1-group4.github.io/Auber-2/assessment2/WhiteBoxTesting.pdf.

All of the tests in this passed. The tests that we run covered a wide range of the classes in the project, however our initial reaction to having completed them all was that they weren't overly detailed (as in they did not test all methods). For example one major admission from the unit tests was the render method in each of the sprite classes (for example the player or NPC). In these instances we struggled to write unit tests for the theme as they tended to lean themselves more towards manual testing and as such we pushed those towards the black box tests.

During the white box testing we run a combination of component, integration and unit tests. This the fact that many of the integration tests run correctly and passed shows us that the overall systems will integrate correctly when built. We used a variety of inputs including random inputs as well as boundary tests. This was particularly true during our tests of the Utils class and the pathfinding algorithms.

The code coverage for each file is as follows and a full code coverage report can be found here:
https://eng1-group4.github.io/Auber-2/code-coverage/

| **File** | **Code Coverage** | | |
|---|---|---|---|
| | **Class %** | **Method %** | **Line %** |
| AuberGame | 100% (1/1) | 0% (0/2) | 16% (1/6) |
| Utils | 100% (1/1) | 25% (1/4) | 16% (1/6) |
| World | 25% (1/4) | 16% (4/24) | 13% (38/277) |
| ui.Button | 0% (0/1) | 0% (0/3) | 0% (0/20) |
| ui.GameUi | 0% (0/2) | 0% (0/9) | 0% (0/101) |
| screens.GameOverScreen | 0% (0/1) | 0% (0/2) | 0% (0/24) |
| screens.GameScreen | 0% (0/1) | 0% (0/5) | 0% (0/121) |
| screens.LoadScreen | 0% (0/1) | 0% (0/3) | 0% (0/66) |
| screens.MenuScreen | 20% (1/5) | 8% (1/12) | 1% (1/97) |
| pathfinding.NavigationMesh | 50% (1/2) | 30% (4/13) | 18% (16/87) |
| pathfinding.PathNode | 100% (1/1) | 50% (2/4) | 69% (16/23) |
| entities.Civilian | 0% (0/1) | 0% (0/5) | 0% (0/12) |

| | | | |
|---|---|---|---|
| entities.GameEntity | 100% (1/1) | 20% (2/10) | 28% (18/63) |
| entities.Infiltrator | 33% (1/3) | 4% (1/21) | 1% (2/108) |
| entities.Npc | 20% (1/5) | 4% (1/24) | 0% (1/144) |
| entities.Player | 14% (1/7) | 22% (5/22) | 19% (42/217) |
| entities.PowerUp | 100% (5/5) | 68% (13/19) | 80% (75/93) |
| entities.Projectile | 100% (6/6) | 72% (13/18) | 62% (45/72) |

Our overall project code coverage was:

| **Overall Project Code Coverage** | |
|---|---|
| Class % | 42% |
| Method % | 23% |
| Line % | 16% |

The overall class code coverage is lower than a more standard 70-80% model. Ours was a little lower than this with the class coverage level being at 42%. We feel this is an appropriate level of code coverage for this project because most of the classes are quite visual in nature meaning that they lend themselves more towards a manual test rather than a unit test. While the overall class code coverage level is appropriate, the method and line code coverages are fairly low. The fact that the only 16% of the lines were hit during our unit tests says that they may not have been as effective as we believe them to be. That said all parts of the code coverage levels are above the minimum 10% and the class and method code coverages above 20%.

In terms of the individual breakdown per module, five modules had zero code coverage. This is particularly interesting for the following three tests: test_load_screen, test_game_screen, and test_GameOverScreenBecause these tests should have run these classes and therefore the code coverage should not be zero. On reflection we have reviewed the code for these tests and realised that we are mocking the implementation of these classes rather than actually implementing them. This means that the code is never actually run in these three files. On attempting to rectify this issue We realised that there was an error creating sprite due to the fact that we were using headless libgdx over the normal version. This issue has therefore not been rectified and future maintainers will need to deal with this.

The Black Box tests table is available here:
https://eng1-group4.github.io/Auber-2/assessment2/BlackBoxTesting.pdf

In total we ran 12 black box tests which is less than originally planned. They were all manual tests most of which also tested the system at the same time. All of these tests passed except for one, test_game_duration. This test was looking at the requirement that

the game must go on for at least 10 mins. The reason for this test was that the average was 8 mins not 10 mins. We put this lower value down to the fact that as a team we were doing the black box testing, as opposed to a seperate software tester. Because of this we became very good at the game, which decreased the time taken to play the game, even when the difficulty was set too hard. We believe that to correct this decrease one of two things could be done. First had someone new to the game tried it the time may increase. Second the difficulty could be increased (this has the disadvantage that someone new to the game would struggle causing the time to decrease; therefore this option is a double edged sword).

The Traceability Matrix is available here:
https://eng1-group4.github.io/Auber-2/assessment2/TraceabilityMatrix.pdf

In the end we did 53 requirement tests of the project with 13 coming from black box testing and 39 coming from white box testing. 13 requirement tests is lower than we would have liked from the white box. In the future we would develop more, as we did not cover all the requirements that white box testing as previously stated we would. That said while not all requirements had a test, however we felt that come where implicitly tested as part of others. For example the requirement FR_HOSTILES did not directly have a test, however the test test_user_can_arrest, test_slow_attack, test_confused_attack and test_random_action show that this requirement was satisfied as if there were no hostiles all of these test would fail. This was true with a lot of the requirements without tests - they can be linked to other tests.