Software Testing Report

Group Number: 10

Team Name: Decassociation

Group Member Names:
Tom Broadbent
Poppy Fynes
Owen Lister
Michael Marples
Lucy Walsh

Testing Methods

The testing methods we used for our code included both automated and manually verified testing, as not all of the elements of the code that we wanted to test could be verified purely by checking if something was true or false.

In terms of automated testing, we used the JUnit library of tests along with GdxTestRunner, as this seemed to be a commonly used method, and we already had some experience with writing tests in this format. The automated tests were used for things that we could verify to be true, which could easily be tested using the assertTrue() function, part of the JUnit library. This was used both to verify the presence of files and assets, but also to verify data driven functions, such as chef movement and station locations.

We broke down the automated tests into separate testing classes which would focus on specific sections of our code, with the sections being: “AssetTests”, “CookTests”, “CustomerTests”, “IngredientTests”, “LeaderboardTests”, “PowerupTests” and “StationTests. In these sections we had appropriate subsidiaries, for example, in AssetTests, we have testChefAssetsExist(), which will test if all the chef based assets are present and findable. This decomposition of the problem allows us to reduce the scope of our tests while the inclusion of these in a few master classes allows us to quickly run all the tests in a shorter time.

While automated testing was really helpful for things where we needed to verify specific values, we couldn’t really test things like drawing functions and proper display of entities in the world without using manual testing. Subsequently we made a “miniWorldTest” class for testing things that we could only really verify visually. In this world we tested the drawing of entities and made sure all the sprites changed properly when they should.

Overall we considered this hybrid of automated and manual testing to be the best for testing and also troubleshooting our issues. The small scope of the individual tests allowed us to find parts of the code that didn’t work, and fix them quickly and efficiently, whereas if we had used a bigger scope it may have been more difficult to find individual problems, and subsequently their solutions. We also found great benefit from the breadth of testing we carried out, as it helped us understand the code we had been given, which has made it easier for us to add to and adjust aspects of the code in accordance with the new user requirements.

Testing Details

As stated above, our tests were broken down into individual classes that encompass related tests, each with subtests to reduce excessive scope. There is no intrinsic order to these tests, so an arbitrary order will be followed.

Asset testing was broken down into the individual sections of: “testChefAssetsExist”, “testCustAssetsExist”, “testAudioAssetsExist”, “testItemAssets”, “testLeaderboardData”, “testMapAssetsExist”, “testParticlesAsssetsExist” and “testEIElementsExist”. In our current implementation all of these tests pass. This is unsurprising as it is basically just making sure that the folders contain all the necessary files to run the game, but it does make sure that they are there, and if we have any issues in coding that we think may have relation to assets then this test would help us diagnose issues.

Cook testing was a little more complicated than asset testing, as this was more based around testing the functional mechanics of the cooks. This gave us the test list of: “testMovement”, “testPickUp”, “testCreateCook”, “testDropItem”, “testFullCookStack”, “testCollisionAttempt”, “testCollision” and “testCookSwitch”. We chose these particular tests because we think it covers all of the mechanics that the chefs employ, which we feel is rather backed up by the coverage of the functions that are available to the chefs In their class. At this time all of the tests are passed by our implementation of the game. The writing of these tests did highlight various issues and ‘quirks’ about how the chefs interacted with the game, especially the ingredients, however it did force us to investigate and ultimately understand how the code worked. We feel that these tests do a good job of covering everything they need to while still being mostly in the realms of small scope, however tests such as the testMovement are quite large and without insignificant iterations. This does mean that we could be complete with the tests, but it does mean that they take a bit longer than they may otherwise have.

Customer testing was rather short, as there are not so many mechanics that need to be tested, which resulted in us only having: “testCustomerSpawn”, “testCustomerDeletion”, “testCustomerAtPosition” and “testIncorrectCustomerZone”. We also had a running “Rule” to make sure that no exceptions were being thrown anywhere. Right now not all of the tests pass, specifically testCustomerAtPosition fails, which we have recognised and have identified the route of the problem to be the customer controller function of isCustomerAtPos. This is something we hope/ expect to fix, but since the customers currently seem to work fine in general it isn’t the most pressing concern. Other than that issue, we feel that our tests effectively cover the customer’s functions and can make sure that everything is working as desired.

Our implementation of IngredientTests was a little limited, as some of the ingredient functions required the passing in of a SpriteBatch parameter, which we couldn’t find an efficient way of simulating for the testing. These tests would be mostly interactions with stations however, and we have managed to cover most of these in other tests, so we don’t feel that there is too much of a hole in our testing of the code as a whole. The tests that we do have in this class include: “testIngredientClone”, “testFlip”, and “testEquals”. These tests essentially encapsulate the creation of, and some related functions related to the ingredients, focusing on the relationship between “Ingredient.java”, and “Ingredients.java”. At the time of writing, all of these tests pass.

For the updated assessment, we added the function of a leaderboard, which had been half implemented already, but was terribly broken, and if the player tried to use it, the game would crash. For this reason we thought it especially important to make sure that we had a

test for this feature, and so we created one. We tested both automatically and visually the adding of values to the leaderboard when a player completes the game, and that passes and works. We also made sure that you can navigate to and interact with it, which works.

Power Ups were a new addition to the game, so we built these ourselves, however we feel that adding tests for them would be good practice, so that we can demonstrate complete coverage of code with testing. In this set of tests we have “testCollision”, and “testActivate”. These tests make sure that the chefs are able to collide, and thus interact with, and also actually activate the powerups. We struggled to find an effective way of automatically testing the actual effects of the powerups, however using manual testing, we managed to make sure the tests worked, which will be covered later.

A section of testing we really struggled with at first was StationTests, as the code we were provided with was rather confusing, however we did manage to work it out and implement relevant tests. This actually turned out to be very useful as it helped us understand how the stations worked and how we could manipulate them for the purpose of adding more, and also fixing the bugs in the old code. The subsections of this class are: “TestCreate[Serving/ Baking/ Frying/ Cutting/ Ingredient/ Prep]Station”, “test[Baking/ Frying]place”, “testCookingStationLockCook”, “testCuttingStationLockCook”, and “testPrepSlotsToRecipe”. We feel that this covers all of the functions regarding creating and verifying the stations. We also have multiple ways of creating stations implemented into the tests, so that we can 1) demonstrate the function and 2) demonstrate our ability to manage the code. Currently all of these tests are passed, which we feel is a good display of correct and complete testing of the stations, including but not limited to making sure that error cases are taken into account, such as the user attempting to input the incorrect recipes into either the stations or to the customers.

In terms of statistics of our tests, results of our tests can be found at: https://decassociation.github.io/project_eng1_team3/htmlReport/index.html. All of our test classes: AssetTests, CookTests, CustomerTests, GdxTestRunner, IngredientTests, LeaderboardTests, PowerupTests, SaveLoadTests, StationTests ran successfully but not every single test methods passed (3/48 failed - see below for specifics). This means that some lines of code did not run; the CookTests class for example had 78% of it’s lines run. Moving onto the statistics of the classes that were tested, 4/6 of the packages had 100% of their classes tested. The screen package has 11.6% classes tested and the util package has 66.7% of the classes tested. The reason for this is due to the screen package having classes (such as CreditScreen) that use lots of shaders and renderer objects so couldn’t be unit tested in the headless environment. For method percentages, there is a range of values for the same reason(s) as just described for the classes. The station package has the highest percentage of methods tested with 75% and the lowest was the util package with 9.1%. Finally, for percentages of lines tested, the util package has the lowest with 5.5% and the entity package as the highest with 72.5%.

Failed Tests:

- testCookSwitch() in CookTests.java
- testAdd() in LeaderboardTests.java
- testCutsomerAtPosition() in CustomerTests.java

We are unable to fix the testAdd() test - we haven’t been able to understand what the real issue is and what would fix it. The leaderboard feature of our game does actually work however (seen from manual testing). Also, the testCustomerAtPosition() test would take too

much time to fix and potentially cause issues within the game that didn't feel worth the time and effort for the benefit (customer movement is easily tested when running the game and in manual testing). This is because there is an issue with how isCustomerAtPos() works in CustomerController.java and we didn't create that method so we didn't feel confident in changing it without breaking other parts of the game. testCookSwitch() has also caused issues that we have struggled to fix. Therefore, to enable all tests to pass we would need to spend a lot of time as a team working on all three test methods and we would likely need a more experienced developer to help us in fixing testAdd().

Overall, we believe that our unit tests cover a sizable amount of our game code so cover lots of essential parts of our game and where unit tests weren't possible we have manual tests so that we were able to test that the game works as expected as a whole.

Links to relevant part c documents for the software testing report - the test coverage report and the details of manual test cases - can be found on our website at:
https://decassociation.github.io/project_eng1_team3/htmlReport/index.html and
https://decassociation.github.io/project_eng1_team3/Manual%20testing.pdf