4. Software Testing Report

Part a)

To test the code, we made use of Junit automated testing, and some Black Box testing for requirements that can't be tested automatically. Junit is a testing framework used to create and execute automated tests. We used this framework to write tests for all classes and methods, to ensure maximum traceability we made sure to create tests that would link most directly to the requirements. We aimed for around 60-70% coverage with our tests, to account for some methods such as those related to rendering that are hard to create tests for. We felt that this amount of coverage was suitable for the scale of the project, and any more would be overkill and detract from the quality of the code itself. We will generate coverage reports using the inbuilt tools within IntelliJ.

There are multiple reasons why it was beneficial to us to choose Junit for our tests. For one, it is very well integrated with the IntelliJ IDE that we were all using to code, which meant that creating the tests was quick and did not disrupt our workflow, and likewise executing the tests could also be done with the same ease as Junit also provides test runners. Creating the classes for tests could be done at the click of the button, and the same to manually execute the test if necessary, meaning that all we had to do was write the logic of the tests. Writing the logic for the tests highlights another strength of Junit testing: its assertions and annotations which make creating and reading tests as straightforward as possible. Another key benefit is that the tests run and provide feedback automatically. This dramatically speeds up our workflow and ensures that bugs are caught and as such solved as quickly as possible with minimal impact to the quality of the code. The underlying point of automatic testing, then, is that it greatly improves the quality of the code at every stage of development, with minimal disruption to our workflow. All of these benefits make Junit the best choice for our project: it is quick, simple and improves the quality of the code. In a small scale, non critical project like this, simplicity and ease of use is key, and Junit testing perfectly fits this goal. Given the game's reliance on graphics and LibGDX, we made use of a headless code runner to test many of the game's classes.

Where automatic testing wasn't possible, but we still felt it was important to test requirements, we employed manual testing. For example, we made frequent use of Black Box testing to test the non functional requirements outlined in the requirements document, as fulfilling these requirements is key to producing a successful product, however these requirements can't be tested using automated testing. We designed test cases and played the game ourselves to determine if the code successfully met the correct standards to pass the tests. You can find our NFR testing plan on our website for more details. We also employed black box testing for other functional and user requirements that we were unable to test automatically, often due to complications with the rendering of the game, detailed reports of these tests are again available on our teams website under the testing section.

Part b)

The first set of tests we would like to discuss are found in the AssetTests class. It is important to mention these first as they are separate from the rest of our tests which are organised by the class which they test. These tests check the existence of all assets which are used in the game, and are organised into testChefSpritesExist, testBurgerSpritesExist, testRecipeSpritesExist and so on. We ran these tests using the file GdxTestRunner, which makes use of the headless back end since it is dealing with assets. To implement this, we created the new GdxTestRunner class, which contained the logic to access the assets, and altered the build.gradle file to provide the correct path to the test assets source (the assets folder). It is important to create these tests as it ensures that if an asset has been erroneously deleted or renamed, the test will fail and point us to the offending asset, making debugging far simpler. Particularly with so many assets, all of which with similar paths and names, it is helpful to us to easily be able to find where errors are being caused. All of these tests, totalling 8 tests and 22 total assertions, passed. Test and coverage reports may be found on our website with the rest of the testing materials, here

The next series of tests can be found within the LongBoiBankTest class, and tests all methods found within the LongBoiBank class which was used to implement the money system. This relates to the user requirement UR_CURRENCY and more particularly the functional requirements FR_EARN_MONEY and FR_INVEST_EARNINGS, which are tested by testEarn and testSpend respectively. All 4 tests in this class passed, which took into account both valid and erroneous cases. Reports about these tests can again be found with the rest of the testing materials on our website, here. As we found when generating coverage reports for this class, we have 100% method coverage and 71% line coverage.

Following on from this, tests relating to the Chef class can be found within the ChefTest class. Many methods within this chef class were unfortunately unable to be tested given the scope and time restrictions of this project. However, where logic was more abstracted, we were able to test some functions, such as the class FixedStack.

Relating to the function of the Chef, FixedStack is an important class, which is tested using the TestFixedStack class, which contains two tests linked to its methods: TestPush and TestHasSpace. These tests confirm that UR_INGREDIENTS (“the player shall be able to collect ingredients”) and FR_GRAB_ITEMS have been met. Both tests were passed successfully. Our tests in FixedStack have resulted in 100% method coverage and 83% line coverage. The appropriate test report can be seen on our website here

One small yet critical class we felt would be important to test is the StationAction class, which determines what action is presented to the user at a station. We tested the method getActionDescription by confirming that the method produces the correct string by making use of Junit’s AssertSame assertion. This test was passed successfully and relates to the requirements FR_FLIP_AND_CHOP, UR_COOK_FOOD and UR_SERVE_FOOD. Our coverage reports show that we have achieved 100% method coverage and 61% line coverage for this class. Again, here is the corresponding test report

However, due to limitations of the system we were using to test our code, not all features were able to be tested automatically, and not all tests were able to be passed. For example,

the tests involved in testing the requirements UR_INGREDIENTS did not pass. This is because the construction of an instance of the Ingredient class requires an instance of TextureManger as one of its parameters. Our testing system, using Junit and a LibGDX headless back end is unable to test any functions which would require the rendering of graphics, as with the TestIngredient class we created. We looked to test the getters and setter methods, which play an important role in achieving the requirements UR_FAILING_STEPS and FR_FLIP_AND_CHOP. When running the tests, an error is raised as JUnit is unable to access the required. We ran into this problem early into development, when we were beginning to understand the strengths and limitations of the testing platform, which meant that from this point onwards, we were able to understand whether or not a test would work for any given method. We concluded that any methods which involved the rendering of graphics would unfortunately not be able to be tested without drastically altering our testing system, which given the scope of the assessment and the strict time constraints, we felt would not be possible unless development finished particularly early. Regardless, we still generated test reports for these failing tests, which can be seen here

We designed similar tests for the requirement UR_COOK_FOOD, which specifies that the user must be able to prepare different recipes. We tested this by looking at the Recipe class, which further recipes “Pizza”, “Salad” etc implement. We designed tests for this class, looking at the getType method and the getRecipeIngredients method, which returns a linked list of ingredients required to create the recipe. By testing these methods, most aspects of the game which interact with the Recipe class are therefore also covered. However, these tests again did not pass, which you can also see in the test report available on our website, due to the previously mentioned complications with the graphics rendering. If we were able to separate the logic from the FoodTextureManager which is causing the problems, then our tests would run and we believe that they would pass as the logic is sound. Instead, we also tested these features manually using black box testing, which we will discuss shortly.

One of the last major requirements which we felt would be important to test is UR_POWERUPS. Implementation for this feature was one of the largest tasks we undertook, and as such it was important to design tests as thoroughly as possible. Unfortunately, we ran into many of the same issues as we did with previous tests, the frequent integration classes which handle graphics (such as ChefManager) meant that we were unable to test large portions of the code with our current system. Therefore, some tests did fail. Details of these failing tests can be found on our website with the rest of our testing materials. However, we were able to test some elements of this requirement, given that we were conscious during development to separate logic from graphics in order to facilitate testing. We were able to test this class's integration with LongBoiBank, which is a testable class to us. The tests were passed.

Where we were unable to implement automatic testing due to these limitations with the software, we designed manual black box test cases in order to ensure that all requirements were still tested. To do this, we considered each requirement in turn, and designed tests for each one. We then played through the game, ensuring that by the end of the game the passing criteria set out within the test we were considering were passed completely. This way, we have full confidence that all requirements have been met and tested, even where our automated solutions were insufficient. Our website has a page where we describe these tests here . We also outlined tests for the Non Functional Requirements, which must be

tested manually. Ideally, these tests should be carried out by those unfamiliar with the game in order to simulate the open day environment. So, we each got friends and colleagues to give us feedback based on these NFR tests. Outlines and results for these tests can be found on our website at this link

We generated coverage reports using the built in tools in the Intellij IDE. This tool was useful to us as it shows both Class and Method coverage, and breaks it down per class. This made it easy to understand and see where further testing was required. Our overall coverage was dependent on our ability to test sections of the code given the constraints posed by the headless backend. Overall, we were only able to achieve a class coverage of 8%, this is disappointing given the target we were aiming for, however we did the best we could considering the difficulties surrounding FoodTextureManager and other such classes. The tests we were able to implement, such as assetTests and testLongBoiBankhad much higher method coverage than our average. These high quality tests coupled with our thorough process of manual playtesting and black box testing mean that we can still strongly say that most requirements have been met. You can view the coverage report here, though its format has been altered when converted to a html document.