Testing methods and approaches

When developing our game, we decided that the majority of our tests should be automated. This was decided on due to the fact that we decided our game should follow test-first development **[1].** This approach to development was recommended by professionals in the field and was also mentioned frequently in much of research material. In this approach, tests are written for aspects of the project before development starts on that part. This is done to break a problem down into easier to manage logic, that when combined, creates the finished functionality **[2].**

We also decided that a combined approach would suit the project the best. Alongside test-first development, we also decided to follow responsibility-driven design. Research into the methodology of organising group projects suggested responsibility-driven design would provide more reusable code **[3]**.

To follow these design approaches, we used Junit tests, as this was a well documented method of automated testing. With continuous integration in place, we could make sure tested features of the game remained functional after each new build. These tests made ensuring past functionality remained intact throughout the project easy, as we could run the test whenever we made changes to key parts of the game.

Our unit tests followed the advice of “DAMP over DRY”, in which tests are explicit in what they do by making the logic as self explanatory as possible. This method helped to improve the readability of the tests. We also split the testing into sections, such as movement, collision or interaction. This largely grouped together tests relating to the same or similar requirement, which made finding tests easier.

Not all parts of the game were testable through this method however. This was due to issues with graphics being needed for certain parts of the game. In these cases, we had to use manual testing. This was never the preferred method for testing, but was used to ensure the code acted as we expected in areas that automated testing could not reach.

To ensure we had as much coverage as possible, we aligned all of our tests up with our functional requirements. This included comments that referenced which requirement was being tested for. This ensured we had tests that were traceable to the requirements.

Results of Testing

Testing on the game was primarily done through running automated unit tests using a headless backend version of the game. These were split into categories of which aspect of the project they referred to. Examples of these files included “AssetTests”,”InteractionTest” and “MovementTest”. We decided on formatting the test files in this manner in order to make finding failing tests an easier experience. If we had decided to make all of our tests in one file, or very few files, finding out which aspect of the game was causing the error would have been harder.

The test in the “AssetTests” file uses “AssertTrue” in order to make sure that all assets used in the game remain in their respective directory whenever we update the project.This test uses a txt file of all the assets used in the game in order to czech whether the necessary graphics exist.

Tests in the “MovementTest” file are designed to ensure the player character's movement is working between updates. We do this by creating a chef and placing them in an empty world. We then tell the chef to move in a certain direction once and use an “AssertTrue” to ensure the chef is in the position they should be in. This folder contains tests for all four cardinal directions, as well as tests to make sure multiple inputs for the movement still result in the chef ending up in the position we would expect. Currently, this file contains the “TestChefSwitch” test, which should test whether a user attempting to switch chef works. This currently fails, and will be talked about more later.

The “CollisionTest” file works in a similar way to the movement tests. It also creates and places a chef in the world, this time with a collidable object in their path. It tests for all four cardinal directions, as well as ensuring the chef slides against objects when two input keys are being pressed at once. It retrieves the chef’s position and ensures it is the same as their expected position. All tests in this file currently pass.

Tests for the general gameplay mechanics are contained in the “GameplayTest” file, which ensures all the recipes are accepted by the assembly station. It also contains tests for the new power-ups, as well as the tests to make sure serving stations work and checking the games reputation system.

“GeneralTest” covers specific functions in the game code, such as getting the x or y coordinate of a serving station. These don’t directly relate to specific requirements but were tested to ensure all areas of the game acted according to our expectations. They also include the tests that make sure general setter and getter classes work as intended, as well as utility methods. For this reason, they are not included in our traceability matrix.

Tests we classified as interaction tests were about the different things the player could interact with in the game. Tests in this area included picking up and putting items down, as well as making sure the player stack worked correctly. We also tested the interactions between the player and the various items and stations in the game. We have extensively tested interactions with certain items and stations, including testing unintended interactions such as attempting to chop fried meat, serving the customer the wrong order etc.

All tests in the “MenuTest” file currently do not work. We did not make many menu tests, as it was clear that none of them would be automatically testable. They are not present in our traceability matrix. We will talk about these in more detail later.

To ensure all aspects of the requirements were covered by our tests, we created a traceability matrix that compared both the requirements and the tests we’d made. We made sure that every test (represented by a column in our matrix) contained at least one ‘x’, meaning we had tested for this requirement. The requirements that a specific test relates to can be seen in this traceability matrix: traceability matrix, as well as throughout the testing code in our project. In a normal traceability matrix, each requirement only links to one test. However, in our matrix, this is not the case. Due to the rigorous testing we undertook, tests ensure separate logic in the game functions as intended. This logic builds into our

requirements, hence why multiple tests fall under one requirement. This matrix covers all the automated tests that relate to requirements and manual tests.

Our automated tests often followed the idea of covering large areas of code by exercising the different scenarios that the classes could experience. We used methods such as “AssertTrue”, “AssertFalse” and “AssertEquals” to be able to determine whether the expected outcome had occurred in these tests. Using coverage reports, we have tried to ensure that all parts of the code that can be tested using a headless backend, have been.

In terms of coverage, we are happy to report that our automated tests cover 77% of the classes in the game (126/162 classes), 73% of the methods that the game uses (978/1330 methods) and 73% of the lines of the code (10394/14136 lines). From research we conducted into testing, we knew that “there’s no silver bullet in code coverage” **[4]**, but we still attempted to cover as much of the code base as we could. We have also realised that only attempting to increase the coverage number could lead to poor quality tests or us missing key components of the system. Because of this, we have prioritised creating robust tests for key areas of the codebase. The method and line coverage is lower than the class coverage due to the use of render code in most of our classes. This code is untestable as it serves to produce the graphics that the given classes need to appear in the game. This has caused our coverage in these areas to suffer.

Despite our best efforts, we have not been able to test every part of the game using automated testing. Out of 150 of tests we ran, 147 passed. In particular, classes in the “game” folder have not been tested in this manner. We initially attempted to create tests for the classes in this folder, but quickly found that it would be a fruitless endeavour. This was due to classes in this folder relating to the different screens of the game, such as the credits and the main menu. These areas of the game have a high reliance on graphical methods, which cannot be tested easily through the headless backend approach. Therefore, the following tests currently fail, not necessarily because the logic behind them is wrong, but because of the issues with graphics in a headless environment. The areas we tested manually include the various game screens and menus, the ability to switch between chefs, buying items with gold and the pause powerup. In addition to this, graphics to do with the chefs and customers (such as stacks), testing screen sizes and game difficulties were all manually tested.

Currently, “TestChefSwitch”, which tests whether you can switch between the different chefs, fails. The error is because the game screen handles the collective group of chefs, and since gameScreen has a high reliance on graphics, this is also untestable currently. A work around to this could be to create a separate class which handles the collective group of chefs, and allow gameScreen to access this class. This would allow us to test the class, whilst keeping the original intent intact.

Additionally, all tests in the “MenuTest” file fail. These were created early into the project and were intended to test whether the logic behind moving between the different screens in the game worked correctly. However, we stopped creating tests in this manner once we realised that testing areas that rely so heavily on graphics using automated testing would be infeasible. As stated before, the tests in this file currently do not pass due to the limitations of the headless backend testing environment. For all tests that fail because of the heavy

reliance on the graphics, we have attempted to rectify these shortcomings by creating manual tests for these areas, as explained later.

At the start of the project, we refactored pivotal areas of the game such as the chefs, interactions, movement etc. The previous team's project relied heavily on the main screen of the game, which needed graphics to work. As explained before, one of the limitations of the headless backend we are running the tests in is that it cannot run code that load graphics. Therefore, these changes were necessary. Due to the constraints on the project, refactoring all areas such as the above examples was not a viable option. We deemed it appropriate to create manual tests for these classes. We did this by creating a table with five components. This is available at: https://uoyollie.github.io/UnderCookedWebsite/assets/documents/Manual_Testing.pdf. The first was the test name. Second was the requirement it relates to. Third was detailed steps to be taken in order to complete this test. The fourth column of the table was the expected result of the test, which was to be followed by the actual result in the fifth column. After consulting with an industry professional, as well as research into this issue, we decided this would be the best method to test these classes, as it allowed us to continue with the project and minimised the time spent refactoring old code. Another reason for manual testing was the human factor that automated testing is unable to replicate. For example, user experience is important to test for, but unable to be covered adequately by automated testing, as the "human brain is irreplaceable" **[5]**.

We currently have manual tests for testing the menus in the game, interactions using gold (such as purchasing power ups, stations and chefs), switching chef and using some of the power ups (such as pausing time).

Bibliography

[1] I. Sommerville "Agile software development" in *Software engineering* , New York:Pearson Education, 20-08-2015, pp. 72-100

[2] Katie *The benefits of test-driven development (TDD)*, *Northcoders*. Available at: https://northcoders.com/company/blog/the-benefits-of-test-driven-development-tdd [Accessed: March 23, 2023].

[3]S. Moreels (2016, Nov. 21). *Responsibility-Driven Design (In Practice)* [Online]. Available: https://www.codit.eu/blog/responsibility-driven-design-in-practice/?country_sel=be [Accessed: April 12, 2023]

[4] S. Pittet *What is code coverage?* , Atlassian. Available at: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage#:~:text=With%20that%20being%20said%20it,fairly%20low%20percentage%20of%20coverage. [Accessed: April 9, 2023]

[5] Gamecloud *Importance of manual testers in video games* , Gamecloud. Available at: https://gamecloud-ltd.com/importance-of-manual-testers-in-video-games/ [Accessed: April 9, 2023]