# Testing

Group 21
Generic Games

Josh Thomas
Andrew Palombo
Oscar Gunn
Scarlet Desorgher
Immanuel Ghaly
Madeleine Nielsen

## Testing Method/Approach

Our aim with testing was to do as much automation as possible because we are a small team. Automation would help us to cut down on time and people-power needed to test and debug our code thoroughly. Therefore, we decided to write our code so that unit testing could be easily implemented. Variables were carefully named, scopes were controlled, and we were careful not to link outcomes to graphics. However, much of the code we were taking on had been written relying on graphics not enabled in the unit testing environment. This limited our testing coverage as one of the main game logic classes couldn't be initialised in a testing environment. To cover this, we did manual tests on things that couldn't be automated. We allocated the testing to the programming team as they had the most experience with the code and could make any fixes as they went.

Our main approach to testing was proactive. This way the test design process was initiated as early as possible,  which was when we got the code from Team 13. This allowed us to mostly find and fix the defects before the project was finished. This also meant we thought about the future development in order to prevent potential problems. To recognise these future threats we thought about the risks the new requirements might give and what code would be the hardest to implement.

Our literal testing approach for each existing method was to first identify if we can use unit tests to test them. After writing and running the initial tests, we checked the Jacoco reports to see any missed branches/missed coverage in the methods we were testing. After identifying these we would adjust our tests to include them or create new unit tests to test the branches. This approach to testing was perfect for our team structure, the implementation team was small so when a person added a new method they could write the unit test for it at the same time.

# Report of Tests

## Introduction

As mentioned earlier, tight coupling between the UI and game logic has meant a significant proportion of our testing has been manual. Our automated tests were written after only the signatures of the classes and those of their respective tested methods had been written and at this stage, the automated tests failed as one would expect. Once the class and method logic had been defined, their respective tests passed 100% of the time, verifying the correctness of our code and forming a useful checklist as classes were fleshed out.

Our manual tests also proved very useful as despite having a very high passing rate of 95%, some end-to-end tests failed, highlighting important corrections to be made. This is because some of these end-to-end tests highlighted that although blocks of code/classes may work in isolation, they sometimes fail when combined.

An example of this was the MN2.0 manual test which first showed the correct behaviour of restoring gameplay when the game started was one of scenario mode. However, when the game started in endless mode, after restoring the session, the system would falsely recognise the game as completed after just one dish. Upon investigation, we discovered that this was not due to a fault in the game logic, but instead one of the logic that saves a game to be restored. This error may have otherwise gone unnoticed if end-to-end testing had not been utilised.

We did however encounter some issues with automated testing within the development of the game due to the design of the previously written sections. We discovered that it was not possible to test code relating to the HUD.java class.

Further details to refer to of the automated and manual testing carried out, including the actual tests, their results, how they were carried out and details of the Jacoco code coverage report can be found on our website here: [Link]. It may be useful to view this alongside this report for better understanding of our approach.

## Report of Automated Tests

*Comment on any failed, completeness, correctness and coverage.*

Our automated tests cover 18 % of our code, as shown in the coverage report, however, this is because the following packages contain classes that are tightly coupled to the UI: ‘com.team13.piazzapanic’, ‘‘com.team13.piazzapanic.Screens’ and ‘Tools’. When this is accounted for, our automated tests cover 52.2% of our code.

As previously mentioned, 100% of these tests passed in the first instance after their respective classes/methods had been written. Please note that instead of using a traceability matrix, we have included the information of which requirement each unit test is verifying on the testing document on our website.

Please find the test plan for our automated tests itemised below.

The **AssetTests** class tests the ‘availability’ property of assets used by our game. These tests were necessary to prevent any other errors from arising in our code that were due to missing assets. As the loading of our assets depends on manually entered filenames, and because there are a large number of them, the use of assets is particularly prone to human error. Ensuring that all necessary assets are present at compile time prevents a potential runtime crash when an asset is requested but cannot be found.

The **ChefTests** class tests the correctness and usability of chef objects and their integration with other classes such as station classes. These specific tests were used because the functionality being tested does not necessarily produce an output, but rather an outcome that needs to be verified. As such simple assert statements were used to check for these outcomes. The functionality of chefs (as the only moving objects in the game) were tested individually to simplify error detection and then later tested in an end-to-end environment in manual testing.

The **OrderTests** class tests the correctness of order objects in isolation using tests for expected conditions, boundary conditions and erroneous conditions. This approach was chosen because orders must generate certain attributes depending on conditions. Therefore this output must be verified in a range of possible conditions that may occur during gameplay.

The **StationTests** class tests the correctness and usability of station objects, ensuring that their methods are correctly processing requests and generating the appropriate outputs. These specific tests used a simple assertion design technique because the expected output only needed to be of a certain type or match a previous value.

For all the aforementioned test classes, they are tested every time the ‘./gradlew build’ or ‘./gradlew test’ commands are entered by a developer and every time a git push/pull operation occurs (occurs on developers’ local machines and on GitHub Actions runners). These automated tests were written by the implementation team, run as previously described and analysed on an ongoing basis by the implementation team.

## Report of Manual Tests

Our manual tests cover the remaining 82% of our code and are divided into two sections covering old and new requirements for this assessment. These manual tests covered anything from ensuring that the correct number of order tickets spawn to checking if the game actually finishes once it is supposed to. Many of these test criteria were quite abstract in definition, written with words such as “quickly” or “mostly”. Without any quantifiable number to compare these criteria to, we could not complete the tests automatically.

As previously mentioned some end-to-end tests failed on the first attempts are running them. This highlighted several key points in the code where sections that we believed to be completely detached from each other would interact in unpredictable ways and cause errors during gameplay.

Please find the **test plan** for our manual tests below.

The manual tests and the actual testing of the old requirements were written by the implementation team. As a whole team we all helped do the actual testing of the new requirements and analysis of the results.

Regarding boundary testing, for MO6.0, we made sure to test all inputs that may happen. We had to make sure that all valid inputs were followed by their appropriate response and that invalid inputs do not affect gameplay and have no response. For this test we included boundary inputs like ‘F’, ‘x’, and ‘4’. We also need to test mouse clicks on the screen and that the area around the buttons does not react.

The manual tests verified the usability, accessibility and correctness of the project. Additionally, we chose these specific tests as they encompass the entire user experience from start to finish (an end-to-end approach).

Regarding **when** these tests were implemented, tests were written before any implementation was programmed in order to ensure that the implementation follows the project requirements. This is because by providing a step by step process of what behaviour of the system is expected, it becomes easier to understand functionality that needs to be implemented. These tests were then carried out by the implementation and testing teams after the respective feature was believed to have been implemented. Additionally, the manual tests were rerun after additional features had been implemented to ensure that new functionality does not break previous functionality.