

# Software Testing Report

Group 26

devCharles

Ross Holmes  
Sabid Hossain  
Thomas Pauline  
Joel Paxman  
Andrey Samoilov  
Louis Warren

# Method Summarisation<sub>(A)</sub>

As we began assessment 2 and started learning about creating and designing tests, we quickly realised that the best way forward was to create as many automatic tests as possible, which meant we strove to solely create unit tests. This meant testing components individually would be very efficient and simple. Unit tests are also widely used in industry and therefore have plenty of online resources to aid us learn since nobody in the group knew how to do testing before assessment 2 began.

Integration tests were avoided since this would involve testing classes/methods instead of specific components which may lead to random behaviours not dependent on our code. Because the code was well designed and we took care to not tightly couple classes/systems/components it meant we managed to avoid using these tests.

To create the tests we used JUnit running in a headless environment as this would allow the tests to run with no graphics and therefore the tests run faster since tests can run without needing to open/run the game.

JUnit allows us to perform all our unit tests and it introduces a `@Test` function, which allows our machines to understand which code is actually a test. It also allows us to use very helpful functions such as `assertEquals` and many more, allowing us to check the behaviour of methods/components and check if they have functioned as expected. JUnit works by creating a test runner that makes a “mock” version of OpenGL (through Mockito) which makes all UI classes stop having functionality (creating a headless environment).





Mockito is used since it allows methods of mocked classes to be called but have no effect, allowing us to test methods in classes that would otherwise try to render, causing the headless environment to crash. This works well for the project because it allows us to test - for example - the movement of chefs. This movement in our implementation requires a `ChefManager`, which in turn requires an overlay - this doesn't directly affect the movement of the chef but is required to “exist” for that movement to happen. In cases like this, Mockito enables testing of an entire class allowing us to test a lot more behaviours.

A small quantity of features could not be tested even with Mockito being used, such as, everything relating to UI and saving. This then led us to have a group discussion on whether to spend a time designing and implementing manual tests for UI classes, and due to a sizable quantity (20-30%) of our code is UI related this meant after some discussion we decided to do this. Though we were a small project without making sure our UI is fully functioning it may lead to the whole game being unplayable.

Since most tests were unit based this meant Black-box testing was only impossible for UI related manual tests and so team members who had less input in code base were assigned to completing that document. Therefore all other testing was white-box testing as team members would be required to look at the codebase to create tests since tests were derived from the codebase and the expected behaviours of that code.

B)

The team made a Manual test table with a small preview :

ID	Description	Related Requirements	Procedure	Expected Result	Status
1	Once loading the gaming check that when clicking the UI "Start" button the stage then switches to a new screen (as described in the expected result)	UR_UX, UR_GRAPHICS, UR_SETTINGS, UR_MODES, UR_SETTINGS, NFR_USABILITY	Load the game and then immediately click on the button saying "Start"		Pass
1.1	Once 1 completed then check scenario mode works (after inputting a value)	UR_GRAPHICS, UR_ENVIRONMENT, UR_MODES, UR_UX,	Once doing the procedure of 1 then add a value to the text box next to scenario mode and then click the scenario mode button		Pass
1.2	Once 1 completed then check endless mode works	UR_GRAPHICS, UR_ENVIRONMENT, UR_MODES, UR_UX,	Once completing the procedure of 1 then click on the endless mode button		Pass
1.3	Once 1 completed then check you can return to the main menu	UR_UX	Once completing the procedure for 1 then click on the back button		Pass

Along with this our continuous integration pipeline also makes a test report:

## Summary

### ▼ Summary

Generated on:	05/03/2023 - 09:06:03
Coverage date:	05/03/2023 - 09:05:46
Parser:	JaCoCo
Assemblies:	24
Classes:	101
Files:	101
Line coverage:	64.8% (2886 of 4449)
Covered lines:	2886
Uncovered lines:	1563
Coverable lines:	4449
Total lines:	0
Branch coverage:	54.8% (572 of 1042)
Covered branches:	572
Total branches:	1042

## Coverage

- ▶ cs/eng1/piazzapanic - 61.8%
- ▶ cs/eng1/piazzapanic/box2d - 76.9%
- ▶ cs/eng1/piazzapanic/chef - 70.5%
- ▶ cs/eng1/piazzapanic/customer - 72.3%
- ▶ cs/eng1/piazzapanic/food - 84.3%
- ▶ cs/eng1/piazzapanic/food/ingredients - 92%
- ▶ cs/eng1/piazzapanic/food/interfaces -
- ▶ cs/eng1/piazzapanic/food/recipes - 92.8%
- ▶ cs/eng1/piazzapanic/observable -
- ▶ cs/eng1/piazzapanic/screens - 0%
- ▶ cs/eng1/piazzapanic/stations - 80.7%
- ▶ cs/eng1/piazzapanic/ui - 6.4%
- ▶ cs/eng1/piazzapanic/utility - 87.6%

The code report was made so that we could have multiple metrics instead of relying purely on solely code coverage. Since this does not accurately portray the thoroughness of tests. All tests are located in the test directory of the github repo as well as this they are located in the source zip file. This then can be run by any team member at any time, to check whatever change they make does not break the tests.

Our continuous integration set up is designed such that every time you commit and push something to git, a testing report is created (as shown above) showing the current amount of lines tested and which tests have passed and failed. The coverage currently shows around 65% coverage which is almost all of the testable code.

The statistics also show branch and path coverage, which are lower than line coverage, but since branch and path coverage demand significantly more time and complexity to completely cover we choose to focus on this less. This is mainly due to the time constraints of the assessment, which meant we had to prioritise. As a group we decided it was more important to check the base behaviour of the codebase (such as working out their equivalence partitions), rather than test every possible branch. This allows the tests to be thorough but doesn't take up disproportionate amounts of time.

During our manual and automated testing we came across several issues such as unexpected behaviours and functions behaving differently to expected. However due to the fact that testing was completed in parallel to implementation, when bugs were found this was communicated to the development team and so this has ensured our game runs without errors and passes every test.

This is all documented on this link: [Actions · dislocated-su/ENG1-2 \(github.com\)](https://github.com/dislocated-su/ENG1-2/actions)

Which shows all the pushes with each one being tested, if and when tests fail they are then fixed by the team.

Our manual testing process was rigorously done, testing every UI element that we have created since these cannot be tested through automated testing. To document these tests, a Manual testing report was created including screenshots of expected behaviours and in depth descriptions of how to individually test every visible UI element. A link to the full report can be found here:

<https://dislocated-su.github.io/team32-website/assets/other/Manual%20Testing.pdf>

Each test is given an ID representing a specific test and then more complex tests reference previous ID's using their methods to get to specific places within the game screens, and then adding actions to then test a new UI element.

Every time there was a major update to the game, somebody would manually fix any broken test that appeared, and add any additional tests if required. With the current implementation each test passes.

# Links

(c)

As mentioned earlier all of our automated tests can be found using this link:

The manual test report can be found here:

<https://dislocated-su.github.io/team32-website/assets/other/Manual%20Testing.pdf>

The latest software test report can be found here:

<https://github.com/dislocated-su/ENG1-2/actions/runs/4870157515/jobs/8685602958>

This redirects you to the code coverage report, including things such as line coverage and branches. This includes both an overview and specific statistics about each class making it a helpful and organised way of finding out where tests are lacking or what has been tested completely. Typically classes at 0-20% are untestable, with a few side effects from other tests occasionally appearing in these classes.