Testing

“Lucky” Team 13

Team 13

Yuxin Wu
Bailey Findlay
Elizabeth Edwards
Chenxi Wu
Alex McRobie
Yihong Zhao

Methods and Approaches

When approaching testing our project, I decided to use a mixture of both static and dynamic tests. The Dynamic tests (whitebox) are written code, automatic JUnit methods that test the inner workings of the project's code. Each test is written ad-hoc for each requirement and works on a basic system of expected result versus actual result. If the project code is working as intended, our tests should pass as the expected result equals the actual result. For tests that for specific or obvious reasons cannot be automatically tested by code, we tested things via human inspection (blackbox). These tests are more arbitrary but work in a similar way. A human runs and plays the game and thoroughly tries to accomplish the requirement being tested, essentially manually checking an expected result versus an actual result. After some research and planning, we decided this was the most appropriate way of testing the project. There is no single technique of testing that is perfect, and so a mixture of static and dynamic testing is required. Exhaustive tests are impossible and therefore we decided to prioritise testing each individual requirement as our requirements inform the game's state and therefore are directly linked to the overall success of the game.

It's important to note that, as our tests focus on functional completeness as well as correctness and appropriateness, we decided not to test the Non-Functional Requirements in their own unique tests as these are very basic requirements that usually all of the User and Functional Requirements inform, so testing all of those will, by extension, test the Non-Functional Requirements. Thus the Non-Functional Requirements have been tested by proxy and are not included in the statistics. (see Test IDs vs Requirements Matrix)

It was also very important to us to try to test how the disperate requirements interacted with each other and therefore how we could merge different tests. Unfortunately the time restrains meant we could not fulfil that to a level we felt was satisfactory. Given more time we would have written numerous integration tests including things like testing if we can operate the in-game UI whilst completing orders, burning items and re-making them to complete multiple orders in a row, mixing on-time and running out of time, and many more ideas we had to implement.

Overall Report & Testing Statistics

There are 27 dynamic (whitebox) tests, and 15 static (blackbox) tests. 27/27 (**100%**) of the dynamic tests pass without failure.13/15 (**87%**) of the static tests pass without failure. A full report of both can be found below, along with a matrix identifying which requirements are tested by which tests and a full automatic coverage report of the dynamic tests (also on the website). In terms of actual code-base tested, JaCoCo measures we test **52%** of the classes, or about **26%** of the total lines of code just with automated testing. Overall, just from automated testing, this feels like sufficient coverage to ensure the game is as the customer intended. Along with the non-automated testing we encompass all of the game's functionality under separate black-box tests.

The tests themselves utilise GdxTestRunner (licensed under Apache 2.0) which is a public, free-to-use open-source testing tool to allow LibGDX classes & methods to be tested as they run naturally instead of as isolated classes. I've also written my own *Utility* class that aides the testing by initialising some of the class Engines so the GdxTestRunner runs smoothly, most of the tests call *Utility.initialiseGame()* to prepare a fresh game run to ensure tests are running on a completely new start of the game and therefore that the tests are correct, appropriate and complete.

I've grouped the tests into categories, some basic constructor and initialisation tests are under "BasicConstructorTests". These are then grouped further into sub-categories that are titled after what they test and they are simple method calls that assert an expected value to an actual value just to check we are initialising the base-game as we expect. There are then the tests under "RequirementTests" which are the tests that explicitly test the requirements and are the main indicator to how the game runs; if any of these were to fail (under current implementation they all pass), it would mean that the game would not meet customer expectations, so they are a fantastic tool to ensure the implementation matches the customer requirements. There are also wider "FunctionalityTests" which at the moment contains only the *CollisionTest*, this sub-class is just for wider tests that cannot fit under one requirement name. They test integrating parts of the code.

A type of test that we simply didn't have time to implement was a large-scale integration test which would measure if different aspects of the game, which were already individually tested, would work whilst integrating with another. For example whether or not we could -in one large test- open the game, select a scenario, complete the scenario as a player would and win. However, we did manage to implement smaller integration tests such as Chef_InteractionTest which tests the chef's ability to move, interact with different worktops test worktop timings and functionality, and interact with food objects. The same can be said for tests like RecipeTest and BurnTest. Given more time, we would have thoroughly tested that the fundamentally disparate parts of the code do indeed work integrated with each other.

The main black box test that failed was actually expected. The second test for UR_UX (the UI design) reflected an error we were already aware of due to inheriting it from the group before us. Essentially the main menu UI is not interactable when fullscreened. This is a bug that we are aware of, visibly documented and given more time we would have fixed it. The only other test that failed was testing for FR_Alert. This was simply an implementation that didn't precisely fit how we initially intended (explained in the Implementation 2 Deliverable).

There was one test which we inhereted from the previous group NFR_audio_visual_compatability that we thought was excellent practise as a non-functional requirement so we kept it. However, we never implemented any audio as it wasn't part of any other requirement, so we left this as N/A in our report & statistics as it wasn't applicable, but given more time it would be something we would implement and therefore involve this requirement.

For all of the tests to pass without fail, we would only need to fully implement the FR_Alert function as well as fix an inherited bug. Other than these two small issues and given the time restrictions in place, our testing is complete to a satisfactory level. Not only do we cover over half of the classes in-code just with automated tests, we cover one hundred percent of the requirements that inform the game directly from the customer's wants and needs. In addition, a great indicator of the completeness of our tests is that the *only* bugs/failed implementations are raised to us by our tests. This is also a valid indicator of the correctness of the tests, none of our tests give false bugs or render different results on method calls, they all operate exclusively and completely as the game does and therefore our testing is entirely correct.

Extras

Test IDs vs Requirements Matrix Matrix to relate tests to requirements.
Testing Report Testing Report of the whitebox & blackbox tests, also automated here.