Software Testing Report

Group 27 BlackCatStudios
Jack Vickers, Hubert Solecki, Jack Hinton, Sam Toner, Felix Seanor, Azzam Amirul Bahri

Summary

Having split into testing and implementation groups, we were able to iteratively test aspects of the code base that we wanted to keep, and later aspects of the base that had been modified to better suit our requirements. While the implementation team improved on the code base, the testing team adjusted the previous assessment's documentation, before moving on to testing the developing game. This started with setting up the testing environment, creating initialisation methods in a master test class to instantiate the world, stations and chefs, and later test these stations, assets and chef interaction. Once game implementation had undergone major changes, the implementation team also began testing the parts of the code base that they had modified and introduced. This strategy allowed for an efficient, iterative testing process that continued as implementation progressed, allowing everyone to understand the code base and adjust it as it grew and improved based on the outcome of the tests.

To set up the testing environment, our testing team leader, Jack Vickers, first modified the build.gradle so that tests could be added to the project. This was crucial for testing our methods and classes because we were then able to track our progress by obtaining line coverage as well as method coverage. Next, the gdx testrunner file is downloaded so that the libGDX headless backend for testing can be used and would make the testing clearer and easier to use. To begin testing, we followed the testing lecture and testing Q&A that were taught in the weekly materials. We found the lecture to be very helpful to start with testing given that we have not had any experience with testing beforehand.

We first generated a general test plan surrounding certain stations and chef interactions that worked as a base to cover the methods and lines within these classes. We separated our tests into separate sections to have a better understanding of what tests we needed to complete and to make it easier to cover the classes that we were testing fully. We separated core game functionality, stations, chefs, customers and assets into separate testing sections, before further separating them by class to test them fully. Once we were comfortable with how tests were written and how they functioned, we added further tests to the test plan to better cover these methods and functionality working off the results that the tests provided us with. With this improved knowledge of testing we then were able to ensure that our method and line coverage were as high as possible. Stemming from this, once we were comfortable with the types of tests that we had to run in each class, we modelled the tests of other classes from the ones that were already present. This loose plan allowed us to delve deeper into each class without any strict rules or boundaries, allowing us to obtain the highest coverage and testability possible. In addition to this, we later attempted to separate as much of the logic as possible from the user interface to also further test game development, increasing our test coverage by approximately 5%.
Unfortunately, not all of our code base was able to be tested automatically and we had to result in manual testing for any of the user interfaces and texture loading methods in classes such as GameScreen or MenuScreen since they attempt to load in shaders. Due to this limitation, we resorted to the manual testing of these methods including observing button functionality and displays on screen for the user interface. For these tests we used debug keys such as a feed first customer button, print statements to verify the correct data and stepping through in the debugger. The rest of the code base that was not tested included general getters and setters which testing would be superfluous.

B)

We decided to be as thorough with our testing as possible, testing as many components of the game automatically as we could. Some elements were either impractical or impossible to test. Such as some tests regarding animations or UI critical components such as the game screen. The majority of our automatic tests are regarding the interaction between chefs and stations, as it would be very time consuming to test every station for new versions. It also helped us to detect bugs we didn't notice or crept in due to merge conflicts. We ended up having over 160 tests and ended up having 63% of methods testing and 57% lines tested, with 100% of them passing. Roughly 10% of methods are getters and the remaining 20% is roughly untestable due to being part of a menu based class with unextractable logic. We used manual tests for these cases. This gave us a rough estimation of 93% tested code.

Furthermore, we did automatic testing on the utilities side of the game which includes; saving the game, loading a saved game, saving the high scores and loading the high scores from the json file into the high scores screen, soundframe tests, input checker tests, input keys tests. We also tested all the work stations in the game which consisted of assembly stations, chop stations, hobs stations, toaster stations and oven stations. Some other tests include trash can tests, chef and customer tests, customer counter tests.

In terms of correctness, they were done as similar to how they would work in the game as possible. We attempted to make the tests as detailed as we could make them. We took care with our assertions to make sure they were correct for each test case, and did not test redundant information. For certain cases, we did manual testing such as testing the timer and testing the UI because it was not possible to test it automatically. It was done by running the game and checking whether the functionality runs with no issues. Further explanation below:

UR_MENU cannot be automatically tested due to it requiring shaders to be loaded in. Thus Manual testing was done by seeing what was on screen.
UR_SCENARIO_MODE was manually tested, as we just need to start a scenario game to see if it exists.
UR_ENDLESS_MODE was manually tested, as we just need to start an endless game to see if it exists.
UR_CHEF_CONTROLS was manually testing for swapping as it was simply changing an integer. An automatic test would be redundant.
UR_CHEF_MOVEMENT was tested automatically by testing pathfinding with different inputs and obstacle avoidance. However user inputs were done manually. We test giving, fetching and receiving from all stations and food containers.
UR_NUM_OF_CHEFS this testing has been done manually as its as simple as adding new chefs in. There is a hard limit of 5 chefs that can be used at any given time. Due to the architecture of the game it didn't seem necessary to check if an int changed value.
UR_INTERACTION we test all interactions between all stations automatically.
UR_WORKSTATIONS all stations exist and are fully automatically tested
UR_COLLECT_ITEM we fully automatically test the food crates. As well as interactions between them and the chef. We also test picking up from all other stations and counters automatically.

UR_REMOVE_ITEM we automatically test if the player can drop an item and give items to the bin.
UR_CUSTOMER_ORDER has been fully tested automatically as we test if all foods can be generated and ordered.
UR_GAME_MAP we test if the map can be loaded in with automatic tests, however we rely on manual testing to make sure it's all correct.
UR_PREP we test whether all stations are functions and will produce the correct results
UR_TIMER has been manually tested due to its unfortunate interaction with games UI that is critical.
UR_PANTRY food crates are littered across the map and interactions between them are automatically tested
UR_ENJOYABLILITY due to the nature of this requirement it is unable to be manually tested automatically, so we enlisted the help of friends to play-test our game alongside the development team to ensure the game was fun and enjoyable.
UR_POWERUP we have tested all powerups automatically. However we had to test if they turn off manually.
UR_DIFFICULTY is tested manually as it just shifts different parameters around to make the game harder. Play-testing was done to make sure the game was fun and playable.
UR_SAVE_GAME has been mostly tested automatically to ensure the game remains saveable. Some elements of the saving system that involved the GameScreen class remain untestable. However core elements have been tested automatically.
UR_LEADERBOARD has been tested automatically to ensure the scores are saved and top 5 scores will display in the high scores screen.
UR_PAUSE_MENU involved loading shaders so we have to test this manually by visual inspection
UR_EARNINGS is tested automatically, as it involves changing an integer so it was relatively simple.
UR_SPEND_EARNINGS has not been tested automatically due to using menu features. However all elements that it touches have been tested.
UR_REPUTATION_POINTS we have tested automatically that these function as live and decrease when a customer leaves.
UR_CUSTOMER_TIME_LIMIT we test when customers leave through frustration automatically
UR_SCENARIO_END we test this automatically when all customers leave and when the player runs out of reputation points.
UR_ENDLESS_END we test reputation reaching zero automatically

Generally functional requirements have the same requirements a user requirements, so they have been omitted.

Non functional requirements has been included due to their difference to user requirements

NFR_SCENARIO_TIME was manually tested, but the length of the game depends on how many customers the user selects. However the default value under the “stressful” difficulty can be completed in 5-6 minutes.
NFR_PLATFORMS we have tested the game on windows and mac. The game is able to be played however on MAC it is difficult to get running but possible due to MAC security. This has been manually tested. The game is fully functional on linux as well.

NFR_MEMORY the game tries to keep memory footprint low, and will reuse images into the game where possible. The memory footprint may rise during the course of gameplay due to new images being loaded from disk. However this will reach a maximum memory footprint as it aims to only load in necessary images. This has been manually tested.
NFR_LAUNCH_TIME the game launches in about one second when the JAR is clicked. This was manually tested
NFR_INSTRUCTIONS the game contains instructions when playing. This has been tested manually due to its connections to menu buttons. Playtesting was required to make sure it was appropriate.
NFR_LEADERBOARD the game contains a leaderboard and and displayed as soon as the player click on the button to open it up

Completeness and Correctness

Interactions were a major focus point in our testing as it affects all critical components of our game. Such as **UR_INTERACTIONS, UR_COLLECT_ITEM, UR_WORKSTATION ect.** We test every type of interaction between chefs and stations using the interaction interface. Such as picking up an item, trying to pick up an item when there is none, trying to interact with a station ect. On most stations we ended with 100% method testing. Correctness in this case has been determined by the item whitelist on each station and through the interaction interface. This is done by checking every item in the whitelist and trying items outside of it. The interface has been used as it can very easily let us try each type of interaction and requires an expected output from them. Allowing us to very easily determine if the test is producing the correct outputs.

Chefs are the players main “port” into the game so they must be tested very carefully and work smoothly. **UR_CHEF_MOVEMENT** was tested manually, however a large component of chefs contribute to **UR_ENJOYABLILITY** so manual testing is required for some sections, as it comes down to a feeling rather than “hard code”. Such as for **UR_CHEF_CONTROLS, UR_NUM_OF_CHEFS, and** parts of **UR_INTERACTIONS.** For some it wasn't worth using automatic tests to change an integer and for the selection of the correct station needed to be tested manually to make sure it felt correct. This method is complete as we test almost all of pathfinding, and manually test the rest of the chef's features via playtesting. Correctness is more loosely defined for this group of requirements due to them being based on how it feels for the player. We used the debugger heavily to verify the correctness, such as in determining the closest interactable object. However we do use automatic tests for determine if the pathfinding algorithm produces a sensible path.

For the new requirements and our additional requirements from ourselves, we have automatic tests set up for almost all of them. E.g. **UR_SAVE_GAME, UR_DIFFICULTY** and **UR_LEADERBOARD.** We have performed manual tests for the rest such as endless and then scenario mode via playtesting and using the debugger. In terms of correctness we have done several playthroughs of the game on different modes and difficulties. We used the print statements to help verify this. For automatic tests we aimed for complete line coverage and were careful to consider every case possible.

C)

To see our Testing Report
To see our Testing Results
To see our Manual Testing