Testing

ENG1 Group 29

Adam Hewlett
Ani Thomas
Dan Kirkpatrick
Matthew Crompton
Niko Chen

Software Testing Report:

When it came to testing the project, we opted for a very simple testing approach for the vast majority of the project. We opted for Narrow Testing in order to cover most of the project, Utilizing JUnit5 to carry out Unit Tests on the code. We took advantage of autonomous testing for the vast majority of the unit testing in order to be more efficient with repeat testing. This was particularly important as we made the mistake of not completing the implementation with testability in mind, leading to testing being exhaustive and time consuming to begin. We designed out testing around the Test Pyramid with the majority of tests being Unit Tests to provide a more accurate testing suite.

Manual testing was imperative for some more complicated tests, particularly involving the power-ups and spending money to gain more staff members. This is due to the increased complexity of these interactions and that we had some trouble isolating the methods that enable this.

Below is an example of the Test Plan we began to make in order to test the first User Requirement:

ID	**PURPOSE**	**INPUT**	**EXPECTED**
1.a	If pressing a movement button returns the correct direction to move the chef	“W” “A” “S” “D”	“W” - inputVector.y = 1 “A” - inputVector.x = -1 “S” - inputVector.y=-1 “D” - inputVector = 1
1.b	If the inputVector can handle opposing input directions correctly	“W”,”S” “A”,”D”	inputVector.y = 0 inputVector.x = 0
2.a	If the inputVector translates to the chef moving in the correct direction (y axis)	inputVector= (0,1) inputVector= (0,-1)	movement.y > 0== True movement.y < 0== True
2.b	If the inputVector translates to the chef moving in the correct direction (x axis)	inputVector= (1,0) inputVector= (-1,0)	movement.x > 0 == True movement.x < 0 == True
2.c	If the system can handle diagonal inputVectors	inputVector= (1,1) inputVector= (-1,-1)	movement.y > 0 && movement.x >0 == True
3	…	…	…

Test Report

UR_CONTROL_COOKS:

When testing for this requirement we primarily tested the Chef and Chef Manager Scripts. This primarily involved testing the following methods:

- getInput() - 8/8 Tests Passed
- calculateMovement() - 6/6 Tests Passed
- setCurrentChef() - 3/3 Tests Passed
- addIngredientToStack() - 2/2 Tests Passed
- removeIngredientFromStack() - 3/3 Tests Passed

This covered the movement controls of the Chef, Items joining the stack and swapping to multiple chefs.

UR_COOK_FOOD, UR_SERVE_FOOD:

Due to the similarity in gameplay with which these occur (food is “made” and then almost immediately served at the very same station immediately after it is “made”) it was decided to be optimal to test these together. For these requirements we tested primarily the Ingredient.java file as well as the individual recipe files and some from the various stations. Here we tested the ability to make each of the required recipes and serve them.
This involved testing the following methods:

- toString() - 4/4 Tests Passed
- getTexture() - 1/1 Tests Passed
- fromString() - 7/7 Tests Passed
- getTexture() - This was across all the ingredient’s individual files - 18/18 Tests Passed
- isCorrectIngredient() - This is across the 4 cooking station’s individual java files as each has the method near the same - 8/8 Tests Passed
- getActionTypes() - From the 5 stations files, these tests show whether the each station can correctly determine which action it can take at the moment - 36/36 Tests Passed - An issue I noticed here is the inability to dispose of incorrect recipes on the Recipe Station meaning making the wrong recipe can soft-lock a recipe station as there are no actions available. An easy fix to this would be create a Station Action that
- doStationAction() - From the 5 station’s files, these test how each station can perform all of their available actions- 54/54 Tests Passed

These tests show the basic functionality of being able to cook and serve dishes.

UR_MODE and UR_CUSTOMERS:

These tests pertain to the functionality of the 2 game modes required to be present. The files most tested here are the CustomerManager.java, HomeScreen.java and GameScreen.java. These are the following methods that were tested:

- init() - From the CustomerManager, we tested here to see if clicking the correct button on the home screen would actually impact the generation of recipes in the game. - 4/4 Tests Passed

- nextRecipe() - Also from CustomerManager; here we tested whether new recipes would be continually be generated for Endless Mode. 2/2 Tests Passed

UR_POWERUPS:

In order to test the 5 power-ups, the method we tested was from the UIOverlay.Java file. We opted for manual testing in this case as it would be simpler than implementing an autonomous test.

- Manually testing allowed us to test multiple pieces of code that allowed the powerups to work from both the buttons on the GameScreen to their implementation in the UIOverlay - 15/15 Tests Passed

UR_MONEY:

Being able to generate money is done in 2 specific places, in the recipe station to actually gain, the Game Screen to correctly display it and Money.Java. We tested the following methods:

- doStationAction() - From the Recipe Station, this test was to show that only a correct recipe would cause money to be generated. - 2/2 Tests Passed
- addMoney() - From Money.java - 1/2 Test Passed - negative numbers would allow this to detract money
- removeMoney() - From Money.java - 1/3 Test Passed - There is no cap to prevent this from falling below zero
- doStationAction() - From the 4 different non-ingredient Stations, testing that the Unlock case did actually unlock the station and cost money - 4/4 Tests Passed
- getActionTypes() - From the 4 different non-ingredient Stations, testing that the Unlock Case is only available when the station is already locked and that money is available. - 12/12 Tests Passed
- Similar to testing the power-ups, testing the buy-button for a new staff member was done manually. 2/2 Tests Passed

UR_FAILING_STEPS:

The Failing Steps are an important part of the project as they add failure conditions into the game.

- nextRecipe() - From the CustomerManager, testing here is to make sure that the timer must be above 60 for it to reduce the life count. - 4/4 Tests Passed - Might have been better game design to have the lives tick down the instant 60 seconds has elapsed on the order
- act() - Tested the method on both the Cooking Station and Baking Station - 6/6 Tests Passed

UR_SAVE

Creating a Save and Load System was the toughest part of the implementation. The main

files that needed methods to be tested from were UIOverlay.java and HomeScreen.Java. The methods we tested for this were:

- UIOverlay() - Here we are testing how well the state of the game is stored when the save button is pressed - 16/16 Tests Passed - An improvement here would have been the current recipe being stored too as that would more accurately replicate the state of the game
- init() - From UIOverlay.java, testing here how accurately the game can be rebuilt from a saved state - 16/16 Tests Passed

While we experienced fairly successful testing, the testing was not as complete as we would have liked. There remained many methods that were not tested as they did not as directly correlate to the requirements we were testing for. Another issue that led to the lack of completeness is the initial difficulty in testing that came with the code we inherited and we did not refactor the code well enough early on to support automated testing. The methods we did test however were done thoroughly, using Mockito in order to better replicate the game environment. Overall, our coverage was not the best but we were thorough with our testing of the key logical components of the game that were necessary to meet the user requirements.