Software Testing Report

Galin Dzhumakov

Aisyah Firoz Khan

Faran Lane

Samuel Nicholson

Jack Polson

Alana Witten

4a).

The team took a test pyramid approach, closely sticking to the requirements of the game, on testing as we evaluated that most of the code could be tested using unit tests with the addition of a few integrated tests. A mixture of different methods were used, taking into account the requirements the previous team had on the game as well as focusing on overall Line and Branch coverage. Tests were automated where possible, however some private methods (such as the chef movement calculation and collision methods) that could not be accessed were tested manually.

A simple test plan was created that summarised the goals of most classes and how they should be tested, which parts can and must be tested, when they must be tested, who was assigned on testing them and then who analysed the results. Despite this, the initial plan wasn't perfect as more and more specifications were added and more possible test branches were discovered which led us to further developing it as testing continued. Additionally, as testing progressed, the team found out that many classes and methods couldn't be tested using unit tests so more integrated tests were carried out than expected.

Following the test pyramid, the automated tests were kept as short as possible with the minimal possible scope so the testers could easily keep track of what is being tested without worrying about anything unnecessary. Test code was also kept "DAMP" by each having a descriptive title that shows exactly what is tested, clearly initialised variables and methods, and a docstring that describes the test and its purpose. Alongside this, to simplify some tests even further, or even allow some methods to run without the need to render within the Headless environment, stubs (doubles) were used to simulate the behaviour of some methods/classes such as the U.I Controllers within the Station or Chef tests using Mockito as well as some fakes were used to replace the need for Texture Managers or Texture Regions for the Food and Station tests when initialising the classes.

In terms of the actual data tested, simple getters/setters/initialisers as well as, basic renders and draws that don't have an impact over the logic of the game weren't tested. The testers used methods such as partitioning the input space for methods that require an unknown input, always taking into account normal, boundary and erroneous data. For example, the chef's ingredient stack was tested using a normal input (any number between 1 and 4) of ingredients, the boundary inputs (0 and 5) as well as 2 invalid inputs such as 6 and trying to remove and ingredient when the stack is empty (-1) covering all possible inputs for the stack, in this case as it is a max of 5.

In conclusion, the pyramid approach, the use of a plan, the use of "DAMP" test code, replacing some data within tests with doubles and the method of partitioning of the input space were all used to make testing simpler, more efficient and get the team's tests closer to functional correctness.

4b).

Tests were grouped according to their respective folders/groupings of classes within the main source folder. The original source code tested a total of 46% of classes, 33% of all methods and 31% of all lines (including branches). Although these numbers seem low, the tests cover every functional and logical specific part of the game, as most of the classes not tested are related to UI which only impacts the design of the game and not how the logic actually works which means the way the UI classes are presented/act cannot break the game. Additionally, a big percentage of the methods and lines missing are getter/setter methods, initialisers (such as the initialisation of all the stations on the map in the GameScreen class) and methods that relate to applying textures/drawing or rendering assets. Some of these classes (apart from all UI classes) include the “Game” and “Home” Screens as well as, a lot of classes had parent methods they derived from where the parent’s methods/functions were not used at all such as the base Station and Ingredient classes.

Overall, there weren’t many opportunities to test some of the methods as unit tests, for example, most of the tests on the “CustomerManager” class had to be done as integrated tests through the StationTests class as there is no way of checking/creating a singular order, same thing goes for some Chef methods as they required the initialisation of the chef manager which required the initialisation of many UI Controllers. Additionally, for a lot of the code to be tested efficiently, lots of doubles had to be used, such as creating stubs of UI Controller classes or fakes of variables/classes required to verify results such as a LinkedList<StationAction.ActionType> to mimic the results provided by the Stations as there was no way of getting those either.

The tests classes are AssetsTests which related specifically to the UR_GRAPHICS and UR_UX user requirements, ChefTests which dealt with the FR_CHANGE_PLAYABLE_CHARACTER, FR_MOVE_PLAYABLE_CHARACTER and FR_GRAB_ITEMS requirements, StationsTests which related to FR_FLIP_AND_CHOP, FR_PLACE_ITEMS, FR_REMOVE_ITEMS and FR_SERVE_CUSTOMER and finally FoodTests which only related to the user requirements UR_INGREDIENTS and UR_COOK_FOOD. It is important to note that some requirements were not implemented in the game, and so could not be tested, these include; FR_GUIDE_USER, FR_COLOR_BLINDNESS, FR_LOADING_SCREEN, FR_SAVE_CHANGES, FR_VERIFY_SETTINGS’_CHANGES and FR_MUTE_SFX.

Starting with the “Chef” folder, it contained the classes Chef(81% methods, 42% lines), ChefManager(45% methods, 62% lines) and FixedStack(100% methods, 100%lines). This folder includes tests making sure that the chef can pick up ingredients, making sure that the chef’s ingredient stack is not exceeded and it doesn’t drop below 0. Additionally, it included ChefManager tests such as checking if 2 chefs can be loaded and the ability to switch between chefs. Finally, some manual tests had to be carried out to confirm that the chef’s movement was working as intended and to also check that the chefs were colliding properly with objects, as these methods were private. The tests confirmed that the ingredient stack and chef movement implementations were working properly as well as the initialisation of the chefs was correct.

The “Food” folder contains all the Ingredient classes, Recipe classes, the ChefManager class and the FoodTextureManager. Both the “Salad” and “Burger” recipe class were completely tested alongside all the Ingredients: “Tomato”, “Lettuce”, “Patty” and “Bun” except the shared “getTexture()” which simply gets the appropriate asset for the respective ingredient. The “FoodTextureManager” class was tested within the “AssetsTests” only to validate the existence of the assets as the methods “getTexture()” and “dispose()” are once again simple getters and don’t have an impact on the logic of the code. Finally, the “CustomerManager” class was tested with 83% of methods as well as 72% of lines covered. Tests within the FoodTests class included checking if food was named properly and presented as the correct type, checking if tomatoes and lettuce could be chopped and if the patty can be raw, half cooked and cooked. Recipes were checked to see if they take the right ingredients and return the correct recipes, or if no recipe is given when the ingredients are wrong. The tests for the customer manager class were entirely done as integrated tests within the StationTests class as there was no obvious way to test methods such as CheckRecipe() and NextRecipe() using unit tests. The tests confirmed that all the recipes worked as intended, only forming when the correct ingredients and their variations were present. Tests also demonstrated that the orders are a set queue, however checking the actual orders/recipes and passing onto the next worked as intended.

The “Observable” folder contains 2 interfaces that could not be tested.

The “Screens” folder was not tested as the Game Screen and the Home Screen classes only initialise everything else (i.e Stations, Food, Chefs, UI) into 1 class called when the game is ran.

The “Stations” folder consisted of the following classes; ChoppingStation (75% methods, 87% lines), CookingStation (75% methods, 89% lines), IngredientStation (75% methods, 80% lines), RecipeStation (57% methods, 74% lines), Station (13% methods, 22% lines), StationAction (33% methods, 45% lines), StationCollider(0% methods, 0% lines). The StationsTests class tested if the stations returned the right possible actions at different given scenarios, if the chopping and cooking stations did their respective actions on the ingredients passed in by the chef, as well as checking if only the right ingredients were accepted by a specific station. The recipe station checked if burgers and salads can be made given the right ingredients as well as if submitting recipes worked properly. The Ingredient Station was checked if it passed the right ingredient as well as if a chef can grab an ingredient from it if his ingredient stack was already full. The station class is a parent class and it was only tested if it can update a station with the right chef as all the other methods were overridden by the children classes. StationAction class wasn’t tested as it only contains the “getActionDescription()” method that only returns a string equivalent to the action passed through. Finally the StationCollider class was also skipped as its only methods were simple getters/setters that add chef subjects to a station. Overall, all tests were passed, which mean that all the stations worked as intended, accepting the right ingredients and returning the correct recipes/ingredients cooked/chopped.

The “UI” folder was also not tested as it consisted mainly of code that solely affected the design of the game such as rendering buttons, textures, fonts, etc. There was no significant, and/or potentially game-breaking, piece of code that had an impact on the actual logic of how the game works.

PiazzaPanicGameClass was not tested as it simply, only calls the game class and initialises some of the UI Controller classes

In terms of code completeness, we think that the tests cover most of the functional requirements that were implemented by the previous team with the exception of FR_FULL_SCREEN and FR_TIMER. However the Non-functional requirements are quite vague and some couldn't really be tested, e.g NFR_OPERABILITY as there is no correct method to unit test that a user with no previous experience can play the game. This overall makes the testing close to being functionally complete. The test code is also very correct as it covers every possible type of input for every test through partitioning (testing for normal, boundary and erroneous data) as well as, covering every possible branch for every line of code ensuring that testing is precise and the system always works as intended.

Unfortunately, because testing was done so late into the project, after merging testing with the implementation code for Assessment 2, a lot of the tests failed. This is because of a mix of a lack of time, miscommunication and the rewriting/adapting of more efficient methods and classes. A lot of tests that had constructors with specific data passed into them such as tests on the ChefManager and CustomerManager classes failed as they had to be changed due to new requirements and design implementations the team didn't account for, as well as, most tests that used any derivative class of the Ingredient class also failed as these were changed later.

4c.
https://eng-25.github.io/assessment2.html#tests