Software Testing Report
Group 30 Triple 10
Team Members:
Kelvin Chen, Amy Cross, Amber Gange, Robin Graham, Riko Puusepp, Labib Zabaneh

For this project, we chose to follow a reactive testing approach. As explained in other documentation, our team was unfamiliar with testing a project of this scale and so decided to focus on this together at the end of our project. This also allowed us to know the scope of the testing and to tackle it methodically. This approach to testing also followed our methodology throughout the project (waterfall), where testing is the second to last stage of the process.

However, recognise that this does limit our project in a few ways. Mainly it doesn't allow us to refactor our code throughout the development. This means that any errors in the code can propagate through the code if not closely managed. Had we tested throughout our development, this would have been partly managed by unit tests.

As a team, we decided to focus on unit tests for as many of the classes as possible. This allowed us to individually scrutinise each class and ensure they all function properly. It also allowed us to identify where the particular bugs were in our code, as the classes were tested independently of each other. Testing these classes allowed us to ensure that we had completed the requirements of the project. Unit testing however can't test how a user experiences the code, something covered in our requirements, so another method had to be found for parts of these requirements.

For some areas of the project, unit tests were not suitable, for example for any methods involving calling render() methods or any UI aspects which involve user inputs. In these cases, we chose to complete manual testing, allowing us to focus on more complex class features. There is also the advantage of testing a close imitation of user experience, as we can perform actions and provide user input as if we were playing the game. With this, we were able to cover a few additional requirements which unit tests would not be able to achieve. Nevertheless, manual testing requires more time than automated testing, so unit tests were favoured over this approach.

Unit tests were written for the following classes:

- Chef/ChefManager
- CustomerManager
- ChoppingStation/BakingStation/CookingStation/IngredientStation/RecipeStation
- Powerup/PowerupManager
- Money
- Reputation Points
- Timer

Every asset used in the game was also tested.

As was briefly mentioned previously, some of the tests that we initially thought we could write were unable to be made into unit tests. The majority of these cases were due to the requirement of user input, which we were not able to mock. As such, these must be manual tests instead, as shown on the website.

Once unit tests were implemented it was possible to make test for the listed classes above. While we didn't have a strict rule for percentage coverage of the tests we were aiming for in most parts to have a test coverage of over 50%. What we were more looking at was that any functions within classes that were entirely logic based had been tested. If this was the case then we felt we could successfully say that such a class had been tested.

Our main limitations during unit testing were any classes or functions that called or were passed a PiazzaPanicGame object as this would call render and load functions which we coudln't run in headless mode for testing purposes. The other roadblock we found was that some functions and especially UI elements required user input to function upon the logic within them. To this end we extensively manual tested those functions and classes which coudn't be access by the headless mode of unit testing.

Going down the list we can analyse the statistics in greater detail. First was the Chef class as well as the ChefManager class. On the chef itself we got a 65% method coverage but only a 26% line coverage. This was due to the extensive collision detection and movement systems which all relied on user input and therefore could not be tested via unit testing. The other 65% of methods which were tested simulated things such as picking up and putting down items. In the ChefManager class we got only a 36% coverage as this class had more visual and UI functionality which couldn't be tested in headless mode. Although from the 63% lines tested we can see that the heavier logic has been tested.

In the CustomerManager class we were able to get a method coverage of 75% with a similar percentage on line coverage. Once again this was limited due to the various UI elements implemented in this class but thankful we could get a large coverage over the logic elements of the class such as registering customer wait times and accepting orders.

Once we got to the stations we found the unit testing to be very similar among them. Percentages ranged from 80% - 66% for the functions in these classes. These functions were mostly the action functions that a chef can take when interacting with the stations. Interestingly enough when creating the tests for the stations we found some that had bad logic that could have caused a bug should we not have unit tested them first displaying the usefulness that testing provided us in this development cycle.

The powerups and subsequent powerupmanager were quite difficult as some of their functions we would have liked to have uniti tested but were unable to thanks to the fact that they interact directly with UI elements such as money and reputation points. This means that while we had to manually test those we were able to create unit tests covering almost 90% of powerupmanagers functions and 60% of powerup's functions leaving us with a good coverage except for the 2 stragalers.

Money and reputation points we managed to cover for a full 100% functionality meaning that we were secure in their functionality thanks to the testing.
Timer on the other hand ad 1 function tied to the UI that meant we couldn't test that however all of the other logic functions were testable giving us a healthy 88.9% coverage.

Finally there are those classes with 0% coverage. These are the UI overlays or screens that couldn't be tested thanks to their relation with the PiazzaPanicGame object. This meant that any user interaction or logic within them required manual testing as detailed below.

<u>Manual Testing</u>

- Downloading Game - test that the download from the website downloads the correct version of the game
- Loading Game - loading game startup from the jar file
- Chef movement - The chef moves with the correct keys
- Chef Collision - The chef collides when interacting with an object
- Buttons - buttons follow through to the correct menus
    - Main menu
        - Start - allows the user to start playing the game
        - Load Game - loads the game screen
        - Tutorial - opens a tutorial page to help explain the game basics
        - Settings - opens a settings page
        - Exit to desktop - quits game
        - Mode selection - the mode of the game reflects the button picked, and only one can be selected at a time
        - Customer - chooses the number of customers that need to be served in order to complete scenario mode, ranges from 1 - 5
        - Difficulty - chooses the difficulty level of the game, applies to both game modes
    - Gameplay
        - Home - saves the current state of the game and takes you back to the main menu screen
        - Star - powerup is applied
        - Shopping trolly - correctly allows players to buy items
        - Grab Item - allows chef to pick up items
        - Place Item - allows chef to put down items
        - Chop - if applicable changes item status
        - Bake - if applicable changes item status
        - Cook - if applicable changes item status
        - Make _ - if applicable asseblems dish

        - Submit Order - if applicable submits completed dish
    - Tutorial
        - Done - takes the user back to the home screen
    - Settings
        - Full Screen - a check mark allowing the user to turn on or off the fullscreen option
        - Back - this button takes the user back to the home screen
    - End Game Screen
        - Return to Home Screen - takes user back to the home screen
- Assets - All assets rendered on the screen correctly

In order to compensate for the classes and functionality that couldn't be tested using unit tests, we used the method of manual tests. This includes UI elements, and functionalities that require user input. For classes that required user input such as Chef, we would run the game and press the input keys W, A, S, and D, and would monitor what the game is displaying to ensure the class is handling user input correctly. To ensure collision detection was functional again we would run our game and control our chef to come into contact with our collison borders, we observed to make sure our chef is unable to make it pass these borders, therefore we know collision detection is functional. The UI elements can be tested by running the game and checking that each element is displayed correctly.
Navigation and gameplay buttons are tested through observation, we make sure each navigation button is functional and performed correctly, the gameplay buttons are tested by having multiple playthroughs of the game. We made sure each button is displayed at the right position and functions correctly. Since these elements couldn't be automatedly tested, we were very meticulous when carrying out these manual tests, and would repeat them often to ensure all problems and bugs are discovered and fixed.