

## CMPT 276 Team 5: Project Phase 3 Report

Kyle Deliyannides 301459316

Patrick Shaw 301537454



















Chris Pinca 301396939

Bassim Ghasemzadeh 301474075

After completing the implementation of “FoodTruck Frenzy”, a thorough series of tests have been developed to ensure proper functionality within the game. These tests are comprised of a combination of unit tests and integration tests that cover every class of the game with the exception of abstract classes. The file structure of the game has been mimicked in a test folder containing the same file names with the suffix “Test”. This allowed an organized and systematic approach when designing and performing tests for the game. For a given test file (FoodTruck.java for instance), the corresponding test file (FoodTruckTest.java) would contain a series of unit tests for every method of the original class. Each method would then be tested for a combination of different parameters to ensure that it behaves as intended for various conditions. JUnit 5 and Mockito 5 were used for testing, along with Apache Maven to build everything.

Coverage testing with JaCoCo reports 96% line coverage and 89% branch coverage. The missing coverage is primarily the UI elements as this is not within the scope of things we learned in this course. Some game loop conditions are not tested because there are too many combinations for it to be worth the time. Refactoring was done to ensure that as much of the code can be unit and integration tested as possible, while leaving the game loop intact. Additionally, some invalid inputs/exceptions are not tested.

### foodtruckfrenzy

| Element                               | Missed Instructions   | Cov. | Missed Branches   | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---------------------------------------|---|------|---|------|--------|------|--------|-------|--------|---------|--------|---------|
| foodtruckfrenzy.SecondaryUI           |  | 91%  |  | 75%  | 7      | 33   | 18     | 155   | 6      | 30      | 0      | 14      |
| foodtruckfrenzy.Helper                |  | 96%  |  | 90%  | 6      | 57   | 6      | 129   | 1      | 17      | 0      | 5       |
| foodtruckfrenzy.GameFramework         |  | 98%  |  | 87%  | 16     | 118  | 4      | 301   | 3      | 68      | 0      | 7       |
| foodtruckfrenzy.Drawable.Item         |  | 88%  |  | 0%   | 2      | 14   | 4      | 33    | 1      | 13      | 0      | 8       |
| foodtruckfrenzy                       |  | 0%   |  | n/a  | 2      | 2    | 3      | 3     | 2      | 2       | 1      | 1       |
| foodtruckfrenzy.Sprite                |  | 97%  |  | 100% | 1      | 5    | 4      | 54    | 1      | 3       | 0      | 1       |
| foodtruckfrenzy.Drawable.Vehicle      |  | 99%  |  | 91%  | 8      | 87   | 2      | 185   | 0      | 36      | 0      | 7       |
| foodtruckfrenzy.Drawable              |  | 100% |  | n/a  | 0      | 9    | 0      | 44    | 0      | 9       | 0      | 2       |
| foodtruckfrenzy.Drawable.BoardElement |  | 100% |  | 100% | 0      | 11   | 0      | 20    | 0      | 7       | 0      | 3       |
| Total                                 | 141 of 4,373  | 96%  | 29 of 271   | 89%  | 42     | 336  | 41     | 924   | 14     | 185     | 1      | 48      |

Below are four sections, each labeled by which group member was responsible for this part of the project. All section's report content was written by the respective person labeled, and all testing and refactoring listed in the report section was done by that individual.

## Kyle

**BoardElementFactory** - I tested every parameter that can be passed into the board element factory, and all their interactions here. Every object that can be created which is displayed on the board (aside from the vehicles) are created through this class, and as such this is a vital component of the system. Boundary testing was also done here, which I learned was very beneficial. If an invalid row or column is entered, the system has been updated to throw an `IllegalArgumentException`. Tests were added to assert that this exception is correctly thrown for each boundary. While creating these tests, I had to add a get type to the `Drawable` class to assert that the correct images were being assigned to every object.

**KeyboardHandler** - did not require any refactoring to make this more testable, as it already has a very simple and optimal design. I used Mockito to create a mock key event, which is passed into the keyboard handler as "pressed" and "released". After each of these mock key events I assert that the system is sane.

**MapLayout** - was refactored to no longer be a static class which allows the behavior to be controlled for testing purposes. I learned here that testing static classes is not as feasible as if it were to be designed non-static. Additionally, I added a few `RuntimeExceptions` to the map layout class. The reason I added these is so that when I test initialization, if one of these is thrown the test will fail. Exceptions are now thrown if there is a discrepancy between the layout file size and the grid columns and rows, or if there is an error loading the file. I also wrote a test which tests every element's get function to ensure nothing is returned that isn't a layout enum (this is caught with the try catch `IllegalArgumentException` on conversion from file to layout enum which is then thrown as a runtime error which will be caught by the test with the row and col of the invalid entry logged).

**Obstruction** - class is tested to ensure its constructor works by using the getters and that the interact function throws an error as it should not be ever called on this object, if it is called it is indicative of a major system problem. This class is pretty simple so no refactoring or bugs needed to be solved.

**Road** - class was a bit more complex. It contained the same stuff as Obstruction, except I had to test the interact functionality. I took care to check that interacting would remove an item and return proper `ScoreValue` objects. No refactoring was done, but I did find a bug that the Glitter object was unnecessarily returning a blank `ScoreValue` object which threw errors when I first started testing this. Getters were not tested here as they have been extensively tested in the board element factory, which tests the interaction as a whole for most items.

**Item** - each individual item class is tested (Food, Recipe, PotHole, Speed Trap, Glitter) to ensure that they have either a positive or negative score, correct score type, correct positions on creation.

**SpriteLoader** - the sprite loader has a test which loops through every single `Drawable` enum to ensure that each one returns a valid `ImageIcon`. I learned the importance of printing out debug messages here, as one image was actually missing. This bug was found because it logs which exact `Drawable` was missing.

**Grid** - started off by having a refactor to accept both the board element factory and the map layout classes in the constructor, instead of having them create their own. This follows the factory design principle that we learned in class, and I learned its importance for testing here. The reason I had to move these to the constructor was so that I could replace them with

modified mock versions for testing purposes, something that would have been impossible with its original design. The grid plays an extremely vital role in this application, so its interactions were tested extensively. I check the get cell by having its initialization mock the board element factory to return a special mock board element which I test is contained in every cell. I then go on by using Mockito to fill up the entire grid with Vertical Road, Horizontal Road, or Obstruction and test the isObstruction function for every single grid element. Finally, I test the system's ability to count the ingredients and recipes by testing 0 and max. To do this I had to refactor the Food and Recipe classes to no longer contain static instance counters and move the counting to the grid class.

**GameConditions** - class was created to separate the win check, loss check, and pause functions from the Game class where they originally belonged. Win and loss checks are tested for every scenario, and the interactions with pausing between classes is tested. This exposed a bug that could cause the loss screen to appear twice, which was rectified.

**Game** - class has some simple unit tests which assert the functionality of pausing and resuming, unfortunately I learned that JUnit is not the right tool for testing UI interactions with these so it is not done here, it just isn't feasible. While doing this, the game loop was simplified and moved to its own function. The constructor was significantly cleaned up to be more readable too. Game conditions were separated from this class to make them more unit testable. After checking the original JaCoCo report I noticed that the branch coverage was only 50% here. Turns out the game loop wasn't even being tested hardly at all. So learned the importance of creating mock objects with Mockito and created some override functions to pass in mock objects. I then test the game loop content by overriding the keyboard handler, game conditions, and food truck. I use these mocks to simulate key press behavior and assert that the food truck move functions are called. Additionally, I assert that when game loss or wins happen I handle them. This was probably my biggest learning experience while testing, and by learning Mockito fully here I increased the branch coverage from 50% to 80% for this class. Any more and it would be redundant for the time required.

**GameFrame** - has been refactored to take in its panels in the constructor instead of creating them itself. This is for testability as we can create mocks of these panels, which do not contain functionality. While doing this I discovered that I needed to make the GamePanel not display items if they are null to make testing easier, so I can give the GameFrame a GamePanel with nothing in it. I tested the constructor, show pause, show game, and refresh systems. The refresh system uses Mockito to test, and this is an integration test between the GameFrame and GamePanel.

**GamePanel** - has been refactored to include getters for all three constructor arguments. I then use Mockito to create mocks of these to test if the constructor works, all three arguments are tested separately. I unfortunately learned again here that it's not really possible to test if the visual aspect of the game is correct, it is more useful to test the underlying components instead.

## Bassim

**CopCreatorThreadTest** - The tests for the copCreatorThread contained a testGetCop test which verified that the returned object is not null and that it is of type Cop. Next, the Cop instance that was returned is then tested to ensure it contains the same properties as intended when initialized. This is where the testGetRow, testGetCol and testGetGrid each verify that the returned value for the corresponding method is equivalent to the values given when the CopCreatorThread is initialized.

**CopTest** - Testing the Cop class involves verifying that with every movement (up, down, left and right) the corresponding move method returns true when successful and updates the vehicle position accordingly. To test this the row and column of the vehicle are stored before each movement and then checked after a successful movement to verify that it moved correctly. Failed movements are also checked to ensure the row and cols remain the same before and after an unsuccessful movement. Note that each of the described tests have made for every possible movement (up, down, left and right). In addition to testing the inherited movement methods from the Vehicle class, the overwritten portion of the cop movements were also tested to make sure the correct DrawableEnum is assigned for each successful movement at the correct orientation. For instance if moveRight() is called, the cop should have a drawableType of DrawableEnum.COP\_RIGHT. After testing the move methods, the cop also needed to have its chaseTruck() method tested. To do this, a failed or finished chase test was done to verify the cop would not move when chaseTruck() is called if the cop's directions array is empty. Next, the cop's row and col positions have been checked before and after chaseTruck() is called to ensure the cop moves in the correct direction based on its provided Direction enum.

**VehicleSpawnerTest** - This consists of testing whether the VehicleSpawner correctly creates instances of Cop and FoodTruck as intended. To do this, the getFoodTruck method checked that it returned an instance of FoodTruck and not null. For getCops the returned ArrayList was iterated and each element was verified to be a Cop instance and not null.

**PositionMapTest** - The PositionMap is responsible for creating a HashMap of Position objects with a method to add Position objects and a method to check if a given Position is in the PositionMap. Hence, both the put and contains methods of the PositionMap needed to be tested together. An empty PositionMap was tested, next a combination of tests were made to ensure the contains method returns true only if the Position's col and row values match regardless of the value of prev value.

For the most part these tests proved that the classes were functioning as intended. However, for the VehicleSpawner, it was discovered that it was able to return null objects in the cops ArrayList. Writing the junit tests for the VehicleSpawner instantly revealed this bug since the testGetCops() failed right after writing it. Initially this was thought to be a mistake writing the test case, but upon further inspection it was actually a bug within the VehicleSpawner. During development the VehicleSpawner added a null cop as a reminder for us to add more cops, however this was a terrible thing to do since it led to a bug that almost went unnoticed if not for the VehicleSpawner unit tests.

## Patrick

### **Scoreboard**

For the Scoreboard, I took a straightforward approach and focused on writing unit tests for each method. I made sure to check each getter and setter in unison and tested the timer, making sure it worked as intended. Additionally, I implemented the ability to pause and resume the timer.

### **FoodTruck**

When I first started working on the class, I noticed that it was too tightly coupled with the Scoreboard class. This meant that it was difficult to test and modify without affecting the scoreboard as well. To fix this issue, I refactored the class by removing a large amount of code and unnecessary methods. I also updated the constructors and changed some of the logic in the Game class to make it work properly.

Once I refactored the class, testing it became much simpler. I wrote unit tests to ensure that each method was working as expected. I also wrote integration tests to test the movement of the food truck on the grid, its interaction with the cops in the game, and how it passed data to the Scoreboard class.

### **FoodTruck, Scoreboard, Grid Integration**

One of the challenging parts of testing the FoodTruck class was making sure that it passed the correct data to the Scoreboard class and that the Scoreboard class displayed the data correctly. To do this, I had to learn how to use the Mockito class. I created a new class called ScoreBoardFoodTruckIntegrationTest to carry out these Mockito tests. These tests involved creating simulated (mock) objects of the scoreboard and the grid. This allowed me to test individual components of their interaction in isolation, providing me with the necessary control to ensure that Scoreboard, Foodtruck, and Grid worked together as intended.

### **Position**

Position class only required simple unit testing. I made sure the constructor was performing correctly and that new positions were being generated correctly in the generateKey method.

## Chris

### **Screen**

The functionality of the Screen class responsible for creating the Title Screens, Pause Screens, Game Won Screens, and Game Lost Screens went through thorough testing. Working on the screen class exposed that it needed some heavy refactoring to eliminate copy and pasted code and blobs of code. The components of the UI screens and the overall layout were then heavily tested to ensure proper function. I wrote unit and integration tests to make sure that image files are loaded properly and that the file paths match for the background and button images. The screen components are also tested for proper initialization, layout and display. The background JLabel, button JPanel, and JButtons themselves are all tested to ensure that GridBagLayout, the image icons, and the components are properly laid out. The createBackground() method is also tested with unit tests to ensure the background image has the correct dimensions, that it makes the background with the correct layout, and that the buttonPanel was added to the backgroundLabel. The createButtons() method is tested to ensure that there are 2 actual buttons with correct image icons and listeners attached. Finally, the resize() method is tested to ensure it resizes images to the specified dimensions. These methods were created to remove

the blob and create methods with a particular function, resulting in far more readable and efficient code.

### **Frame**

We tested the functionality of the Frame class and the associated Screen objects that it creates. We decided to add this class after building the Screen classes to make it easier to create a certain type of class using pre-determined enums. The Frame class is responsible for constructing a Frame object with the specified screenType and Scoreboard object. The creation and display of the frame is thoroughly tested, ensuring that it has the proper dimensions (with a slight discrepancy for different frame sizes due to the different displays of unique systems). We figured out that just passing tests with specific dimensions on a single device wouldn't work on different devices, so we added this leniency in values tested. We ensured that the width and height must be within 15 pixels of the specified sizes, the frame title has to be correct and the frame itself must be visible. We also tested that the unique classes like TitleScreen, GameWonScreen, and GameLostScreen are actually instances of the specified types. Refactoring our code and adding this class made it very simple to add new screens for each scenario of the game where we needed a Secondary UI screen.

### **GameOverScreen**

We tested the functionality of the GameOverScreen responsible for adding the end scores and time onto the Screen displayed at the end of the game. Unit and integration tests ensure the label displays the final scores and time at the end of the game and is initialized with the proper values. This is done by ensuring the correct values are passed, the correct text is created and the label is not null and is actually added to the screen. We decided that this class should extend the Screen class, but couldn't just be another type of Screen in of itself, it had to have the functionality of displaying the final results and display the specific Win or Loss screen that were created as children of the GameOverScreen class.

### **Dimensions**

We tested the functionality of the Dimensions class responsible for containing the height and width to be passed through a Dimensions object to the Screen class. This was a new class added during the refactoring stage to remove the need for hardcoded height and width parameters to be passed through. The getHeight(), getWidth() and the dimensions object itself are tested to ensure that the object is not null and contains the correct values.

### **ImagePaths**

We tested the functionality of the ImagePaths class responsible for containing the background, button1, and button2 image file paths to be passed through an ImagePaths object to the Screen class. This also was created during refactoring to eliminate hard coded parameter path values. The getBackgroundPath(), getButton1Path, getButton2Path, and the object itself are tested with unit tests to ensure the correct value and that the object is not null.