Engineering 1 Group Assessment 2
Software Testing Report
Cohort 3 Group 23

Team Members:
Harry Draper
Seif Hussein
Tikhon Likhachev
Thomas Maalderink
Joshua McKean
Sebastian Armstrong

Testing methods/approaches

The approach we took to testing involved creating a wide range of unit tests, covering as many methods as possible in all classes. Unit testing is done by testing individual components of our Piazza Panic game, like testing individual methods of our game and testing individual classes of our game and functions of our game, testing was done in isolation as every class in our game had its own testing class and every method in our game had its own testing method that was located inside the testing class that is made for the class of our game, that is how isolation of testing was done to ensure that every class and every method was correctly functioning. Junit testing for unit testing was used as it has numerous benefits, it was very easy to set up and also easy to use, so as it was our first-time doing unit testing, that really helped us a lot. Also, JUnit enabled us to automate our tests and made it easier for us to run them - this automation ensured that testing was done with good reliability and consistency. It was also simple and easy to run tests which enabled us to run the tests regularly in order to catch errors and bugs early and easily. Moreover, Junit testing provides a way to organise tests into test suites to avoid discomposure. It also provides a range of assertions that were used to ensure that the functions behaved the ways we wanted. Additionally, Mockito was used to mock objects and create a double for them that had graphics in their initializations but these objects were necessary for other objects that we were testing for.

We began by testing as much of the inherited assessment 1 code as possible, in order to allow us to detect any new issues or bugs during development of new features. Whilst continuing development for the assessment 2 features, new methods and classes were continuously added to the tests package and unit tests were written for them.

We found this to be an effective method of software testing for our project as it enabled us to integrate newly developed code with the inherited code, and to detect any new issues in the old features that our new code caused. As we regularly tested our code throughout the project, bugs became apparent quickly enough that they could be fixed without affecting development time significantly.

Another reason this was an effective method of testing our code was because we were able to integrate it with our continuous integration pipeline. During this, the tests are automatically run, and we are easily able to see if any of them failed by looking at the final status of the build - if it failed that means that one or more of the tests failed.

Certain parts of the project had to be manually tested by us rather than done with unit testing, either because it was easier/more convenient to, or because unit/automated testing was not relevant for that specific element of the software. One example of this was UI elements such as the various screens in the game (Instructions, menus etc.) or the various text elements displayed in the main game screen. This was not necessary to unit test as any changes made to these elements could quickly be seen by launching the game and navigating to the necessary section, playing for a short period of time if necessary. Any changes made would be hand tweaked so that visually it looked and acted the way we needed it to. Furthermore, any values being used by such UI elements were already being unit tested elsewhere.

Another element that was necessary to manually test throughout development was the gameplay/balance of the game. We had to continually ensure that the game remained at a good level of difficulty, without being too easy or too difficult. When difficulty selection was added later on in development, we had to balance the different settings, alongside the other gameplay elements which were not affected by the difficulty level. We ensured that testing the gameplay was a high priority throughout development, as it was necessary to the success of the project.

Test Report

Automated unit tests were run on most of the classes and methods in the project:

AssetTests.java:
- testStationAssetsExists() tests if all station assets and related assets exist
- testCustomerAssetsExists() tests if all possible customer assets exist
- testIngredientAssetsExists() tests if all assets for ingredients exist
- testChefAssetsExists() tests if all chef idle/moving assets exist

BakingStationTests.java:
- TestBakingStationInteract() tests that all possible interactions with the baking station work correctly

BinTests.java:
- TestPlaceItem() tests whether the player is able to interact with the bin when holding an item, and if the bin deletes it
- TestTakeItem() tests to make sure the bin is never holding an item

CustomerTests.java:
- TestCustomerFulfillOrder() tests whether fulfilling an order gives the correct reputation, score and money to the player, and whether it makes the customer leave
- TestCustomerGetOrder() tests to make sure the method returns the correct order
- TestCustomerConstructer() tests the constructors with different difficulty levels to ensure it changes correctly
- TestFulfillOrder2() tests to make sure the method removes the customer from the list correctly
- TestOutOfTime() tests whether the customer leaves after waiting longer than their allowed time, and whether it affects the players reputation points correctly
- TestAct() tests whether the act method correctly makes the customer leave when their timer reaches 0
- TestSetTimeLimit() tests whether the difficulty affects time limits of customers correctly
- TestGetOrderString() tests whether the correct string name of the customer's order is returned

CuttingStationTests.java:
- TestCuttingStationInteract() tests that all possible interactions with the cutting station work correctly

DishTests.java:
- TestAddIngredient() tests whether the addIngredient() method works correctly
- TestIsComplete() tests whether a dish is complete when all necessary ingredients are added
- TestIsEquals() tests whether the equals() method for dishes works correctly
- TestGetRecipe() tests whether the getRecipe() method returns the correct String
- TestCheckComplete() tests whether the recipe is completed if completed it sets the texture of this dish and removed the ingredients of them
- TestSetRecipeBurger() tests whether the setRecipe() sets the recipe to burger when ingredients of burger are added
- TestSetRecipeSalad() tests whether the setRecipe() sets the recipe to salad when ingredients of salad are added
- TestSetRecipePizza() tests whether the setRecipe() sets the recipe to pizza when ingredients of pizza are added
- TestSetRecipeJacket() tests whether the setRecipe() sets the recipe to jacket potatoes when ingredients of jacket potatoes are added
- TestSetRecipeError() tests whether the setRecipe() returns false when trying to add ingredients that don't fit in the recipe.
- TestSetRecipeCheese() tests whether the setRecipe() does not fit the recipe to neither pizza or jacket potatoes when cheese ingredient are added as it fits both recipes

FryingStationTests.java:
- TestFryingStationInteract() tests that all possible interactions with the frying station work correctly

GameDataTests.java:
- TestGameDataAddScore() tests whether score is added correctly
- TestGameDataAddMoney() tests whether money is added correctly
- TestGameDataSetScore() tests whether score is set correctly
- TestGameDataSetMoney() tests whether money is set correctly
- TestLoseReputation() tests whether reputation is lost correctly
- TestAddReputation() tests whether reputation is added correctly
- TestSetReputation() tests whether reputation is set correctly

IngredientStationTests.java:
- TestTakeItem() tests whether picking up an item works correctly
- TestIsEqual() tests the equals() method for ingredient stations
- TestPlaceItem() tests to make sure you cannot place an item on an ingredient station
- getIngredientStationAsset() tests that the correct asset is set for the type of ingredient station

IngredientTests.java:
- TestIngredientCut() tests whether the correct types of ingredients can be cut and that their texture and other internal data is updated correctly
- TestIngredientFry() tests whether the correct types of ingredients can be fried and that their texture and other internal data is updated correctly
- TestIngredientBake() tests whether the correct types of ingredients can be baked and that their texture and other internal data is updated correctly
- TestIngredientGetCuttingProgress() tests whether the correct value is set for cutting progress
- TestIngredientGetFryingProgress() tests whether the correct value is set for frying progress
- TestIngredientGetBakingProgress() tests whether the correct value is set for baking progress
- TestIngredientIdenticalTo() tests whether the method identicalTo() returns the correct boolean value given certain conditions
- TestIngredientEquals() tests whether the equals() method works correctly for Ingredients
- TestIngredientConstructer() tests whether the constructor for ingredients works correctly
- TestIngredientBurn() tests whether the correct change is made to an ingredient when it burns

IngredientTypeTests.java:
- TestGetName() tests that the getName() method returns the correct name for the IngredientType
- equals() tests that the equals() method for IngredientType works correctly

LevelTests.java:
- TestLevelConstructor() tests that the constructor for the level creates the correct objects at each tile in the tilemap
- TestGetLevelName() tests that getLevelName() returns the String of the level name correctly
- TestGetStation() tests that the getStation method returns the correct Station

ObstacleStationTests.java:
- TestHasItem() tests whether the hasItem() method works correctly for an obstacle station
- TestTakeItem() tests that the player can pick up said item

PhysicsTests.java:
- TestPhysicsIsSolid() tests that collisions work correctly
- TestPhysicsPlayerMovement() tests that player movement works correctly

PlayerTests.java:
- testPlayerPuttingDownItems() tests that players can place items correctly on relevant stations provided their inventory has an item
- testPlayerMovement() tests that the position changes when the player has had a velocity change
- TestPlayerPickUp() tests that players can correctly pick up items from a relevant station provided their inventory is empty
- TestPlayerFacingStation() tests that the game knows correctly which station a player is facing

- TestPlayerTop(), TestPlayerBottom(), TestPlayerRight() and TestPlayerLeft() test whether the bounds of the player's hitbox are correct
- TestPlayerAct() tests that the act() function for the player enables the user to use all possible actions
- TestPlayerSetSpeed() tests that the speed is correctly set by setSpeed()

PowerTests.java:

- TestPowerUp() tests that all the available power ups give their desired effects to the player and display them on screen correctly

SaveLoadTests.java:

- TestSave() tests that the save() function works correctly and writes the correct data to the file
- TestLoad() tests that the game state is set correctly by the load() function
- TestOrderInt() tests that an order String can be correctly converted to an int

ServingStationTests.java:

- TestServingStationServeOrder() tests whether the serving station accepts the correct orders and fulfils a customer's order properly
- TestServingStationPlaceItem() tests that the player can correctly interact with the serving station
- TestServingStationTakeItem() tests to make sure the player cannot take anything from the serving station

StationTests.java:

- TestStationPlaceItem() tests that the player can place an item on a station properly
- TestStationHasItem() tests if the station inventory works correctly
- TestStationTakeItem() tests that the player can take an item if a station has one
- TestStationIsEquals() tests the isEqual() method for Stations
- TestStationConstructerGridXY() tests that the station is in the correct location

Vector2Tests.java:

- TestGetAbsoluteX() and TestGetAbsoluteY() test that the correct value is returned for their respective functions
- TestGridUnitTranslate() and TestGridUnitTranslateDouble() test that the gridUnitTranslate() method works correctly

In total, this amounts to 84 unit tests and the tests took 1.243s to complete. The coverage of the tests was as follows: Class (60.8%), Method (67.8%), Branch (62.5%), Line (80.7%). A full breakdown of coverage is available on our website. None of these automated unit tests failed in the final build of the project. We believe that these tests provide a mostly complete solution to unit testing our project, as the majority of the methods that are used in development are fully tested, besides the few that are left to manual testing as mentioned earlier. Further statistics can be found on the relevant section of our website.

Things that some of the tests did not cover in methods are not possible in the code as our implementation restricted cases of it happening by using if statements, returns, switch case statements etc., so some cases may not be shown to be covered, but it is impossible to be covered as no test environment inside the test can be created to cover them, other than that, the tests covers the things that can be done and all the branches and lines. Our testing ensures completeness and correctness as it covers most of the possible scenarios and outlines cases and special cases that could happen in code, it ensures correctness as it accurately identifies and analyses bugs and errors in the code and provides accurate and precise feedback and results.

URLs to the automatically generated results and coverage report can be found on our team website.