Simon Fraser University
CMPT 276 : Introduction to Software Engineering

Phase 3 Project Report
Test Phase

Group 19
Arda Cifci, Nayeong Lee, Ellie Neufeld, Cole Thacker

Unit Tested Features

Input handler had to be tested to verify proper handling of the commands inputted by the user from the keyboard. When the WASD keys are pressed, the player character is expected to have moved, therefore testing this capability is of the utmost importance to facilitate an actual game with user interaction.

Collision checking for entities also had to be tested to verify that when any entity crosses paths with another, a collision is detected. Things like the player character colliding with blockers/walls, fry cooks, deep fryers, and door keys are important to the game mechanics and logic as specified in the requirements gathering phase. This class determines characteristics of movement across a variety of on screen entities and facilitates the being able to win/lose games.

Another feature that needed to be tested was the enemies' player tracking. The PlayerTracker class determines the paths that each FryCook takes towards the player character. We needed to verify that the FryCook moves in the correct direction based on the player's current coordinates, and that the FryCook is able to move around obstacles while maintaining its path towards the player. Testing this feature was a top priority as besides the player, the enemies are the only other moving objects in the game. Any bugs with the enemies' movement would be a glaring defect that would get noticed right away.

We also tested the functionality of the MapReader class, to ensure that it was properly reading the text files with the level maps and that there were no unexpected values in the resulting BoardTile array. It is important that the values in the BoardTile array are correct, because they represent the positions of obstacles, enemies, and rewards on the game board.

The methods in the Board and BoardTile classes needed to be tested to ensure that they output the correct results. The method getOpenTileCoordinates in Board is needed to determine the spawning location of the PotatoFriend, which is our bonus reward. The methods getDeepFryerPositions and getDoorKeyPositions in Board are used to create all of the DeepFryer and DoorKey entities.

The BoardSpriteManager class had to be tested to make sure that all of the images used in the game got loaded in correctly and doesn't cause problems in other classes.

The Music class needed to be tested to ensure that while the game is playing, the music gets taken from file and plays properly.

The GameState class needed to be tested to ensure the game's mechanics worked correctly. If GameState doesn't function properly, the game cannot be played successfully. GameState is responsible for the timer, score, the player character's coordinates, and rewarding and punishing the player. Game state acts as a data model for what could be considered as the metadata for the game containing renderable width/height and each individual tiles pixel width/height.

We needed to test the GameLoop class, however GameLoop is composed of other classes, acting as a unifying source of functionality acting on data models. GameLoop relies on component class logic to act properly. GameLoop also contains the ability to render the objects to the screen and display certain win/lose/pausing screens when those screens are required.

Lastly, we tested all entity classes in the Entities folder to ensure that each entity is created successfully with the given XY coordinates. For DeepFryer, we needed a unit test to check if it had been triggered correctly. For FryCook and PlayerCharacter entities, we needed unit tests to verify if they get the correct image of the entity, correctly move the entity and set the new image of the entity based on given x and y positions. For the DoorKey entity, we made a unit test to ensure that the setToCollected() method correctly sets DoorKey's data member hasBeenCollected to true. We also wrote a test to check that hasBeenCollected is set to false when setToCollected() hasn't yet been called. We also devised unit tests for EntityFactory to ensure it creates the object that is expected. More detailed descriptions for all unit tests are explained in the Test Feature Coverage section.

Component Interactions

We would have to do a small integration test across Collision Checking and moving the player character before a full integration testing of being able to take in input, move a character and check collisions, to reduce the potential complexity of debugging a full input-player-collision chain of tests. Integration testing between Collision testing and input handling is important to handle the taking in of a player's input and verifying that they are able to move in that direction, or that they are even able to move. To test we verify a single move, then verify a single prevention of movement with a collision against a blocker/wall. This allows end to end integration testing of the data model for player movement and collision detection.

In GameLoopTest, we test the interaction between moving between maps, and the interaction between GameLoop and GameState.On the changing of maps the GameState class should be updated by the GameLoop class, this is covered in testnextGameScreenNonFinish 1 through 6.

The MapReadTest class verifies the interaction between the game application itself and the external source of maps, in text files in resources, thus requiring an integration test to verify the class and the data source. Another class that's purpose is to read in resources from an external source, and therefore requires an integration test is the BoardSpriteManager class and is tested in BoardSpriteManagerTest to verify the existence of image resources that are able to be read in properly.

Test Feature Coverage

Each entity in the Entities folder had to be tested to make sure that all the methods in each class work as expected. For the DeepFryer class, we devised a unit test that checks if the getHasBeenTriggered() method successfully returns True after calling triggerFryer(). For the PotatoFriend class, we devised a potatoFriendPosition unit test that checks if setting a PotatoFriend's position to certain XY coordinates and calling getX(), getY() methods return the expected XY coordinates. For FryCook and PlayerCharacter, we utilized a try/catch loop to resolve I/O exception errors. For the FryCook class, we devised 1) GetActiveImage unit test that checks if calling getActiveImage() method successfully returns activeBufferedimage, 2) Move unit test that checks if frycook entities successfully changes its image(left/right) and final XY coordinates after calling move(xPosition, yPosition), 3) GetSetForce unit test that checks if setForce() to a certain direction and calling getForce() method returns the expected direction, 4) GetSetCaused unit test that checks if setCaused() to a certain direction and calling getCaused() method returns the expected direction, and 5) GetXY unit test that checks if calling getX() and getY() methods return expected XY coordinates. For PlayerCharacter class, we devised 1) GetActiveImage unit test that checks if calling getActiveImage() method successfully returns activeBufferedimage, 2) Move unit test that checks if a player character successfully changes its image (left/right/up/down) and final XY coordinates after calling move(xPosition, yPosition), 3) GetXY unit test that checks if calling getX() and getY() methods return expected XY coordinates. Lastly, we devised EntityFactoryTest, which tests the entire EntityFactory class. Inside EntityFactoryTest, we made 5 unit tests: testMakeDeepFryers(), testMakePotatoFriend(), testMakePlayerCharacter(), testMakeFryCook(), and testMakeDoorKeys(). For each unit test, it checks if calling the make[entity] method in the EntityFactory class creates the same entity object as the one created from its own entity class. For makeFryCooks(difficulty, tileSize) especially, we used a switch statement and 5 different difficulty cases to make fry cook entities depending on the game's difficulty.

All methods in Board and BoardTile classes were tested in BoardTest.java and BoardTileTest.java, respectively. In BoardTest, we devised 7 unit tests for all getters in Board class to check if they return expected values. Some of these unit tests include getting width/height, tile type, positions, etc. In BoardTileTest, we devised 18 unit tests: 2 XY position getters, 2 IsOpen() checkers, 14 tile type getters (from type 0-10 and type A-D). Similarly, BoardSpriteManager.java was tested in BoardSpriteManagerTest with 16 getter unit tests that are in charge of testing if the sprite is successfully getting buffered elements without returning null pointer. Some of these unit tests include getting a buffered potato friend, buffered side wall, front chair, etc.

Collision detection testing begins with unit testing the CollisionCheck class via CollisionCheckTest class. Testing happens with open movement around an open tile, moving in each direction. Then we test movement in each direction colliding with a blocker/wall. Testing a blocker/wall is no different than testing a blocker AND a wall separately. Then testing happens with each entity type and player, testing collision and non collision by spawning an entity on top of player and off of a player. Finally we test door collision feeding doorOpen being false/true.

Input handler class is tested via InputHandlerTest via several functions that simulate key presses for WASD and gets the classes from the InputHandler class under test, and verifies the type of CharacterCommand concrete class created.

InputHandlePlayerCharacterIntegrationTest does integration testing across Input handling and the player character by taking in a KeyEvent, moving the player character, then asserting the final value that the player character should have moved based on the event produced.

Collision Checking, Player Character movement, and input handling is tested in InputHandlerCollisionCheckerIntegrationTest where the player character is spawned one unit of movement away from the other entity/blocker/wall, and then moves exercising input handling, then moves into the entity/blocker/wall to test movement/collision detection.

The Music and MapReader classes are tested with MusicTest and MapReaderTest. The Music test makes sure that music is playing. The MapReader test makes sure that the x and y coordinates of each BoardTile are correct, and that the tileType of each BoardTile is a valid value.

GameState testing is done by GameStateTest. We tested getters and setters by comparing the set value with the expected value. We tested the score by calling methods to increment and decrement the score, and then compared it to the expected score. The timer is tested by testGetTime, testGetTime2, testCheckIfTimeElapsedFalse, and testCheckIfTimeElapsedTrue. GameState's resetting methods were tested by changing the game state, calling the reset method, and then comparing the current game state with its expected initial value. We also test whether or not Potato friends spawn within the proper time frame with testCheckIfPotatoFriendRespawnTRUE and estCheckIfPotatoFriendRespawnFALSE. The TRUE version tests if the Potato friend spawns within the right time frame, so beyond some number of seconds, and the FALSE version tests if the Potato Friend is spawning too quickly.

The GameLoop class is tested in GameLoopTest with 10 unit tests. We devised 2 unit tests for checking whether the game score and level have been resetted to default value and whether player character's XY coordinates have been set to its default position. These unit tests verify underlying data being changed in the GameLoop class, and more functional testing is used to verify proper GUI elements. We test resetting with testReset, where the entities and values are reset to starting positions and values and player and key being reset via testEntityAndKeyReset to verify those values associated with the player character are returned to a starting position and start value. We then test the default value of gameLoop regarding the direction that we can get a non-null arrayList of fry cooks with testgetFryCooks. We then test various screens, which are the maps, via the testnextGameScreenNonFinish tests numbered 1-6, each corresponding with a map type.

The enemy player tracking testing is tested in the PlayerTrackerTest.java class. In PlayerTrackerTest there are 8 unit tests designed to test the various movement tracking scenarios the enemy could come across. The first four unit tests (lines 31, testFryCookMoveLeftToPlayer() to 81, testFryCookMoveDownToPlayer()) test the base enemy tracking and directional movement without any obstacles in the enemies path. These base tests ensure that the PlayerTracker.java class logic properly knows which direction the enemy has to move to reach the player in the shortest distance. The next four unit tests (lines 83, testFryCookForcedFromRight() to 164,testFryCookForcedFromDown()) tests to see if the enemy can change directions to move around obstacles in its path to reach the player. These tests ensure that if the enemy hits an obstacle, then the enemy's logic has the ability to move around the object and resume pathing to the player.

Test Quality

Tests should be granular enough to discern logical units, but not so granular that they are trivial. The tests functions should be as small as possible to exercise as much as it can, instantiating only what it needs to get the test done. Unit tests should focus on functionality within classes, which are designed to hold a cohesive amount of logic within the functions. Attempting to test across UI classes could result in poor test cases with JUnit structural style of testing due to the fact that JUnit would need some results to compare to, and Swing typically does not return anything to compare.

To ensure the quality of our test cases, we made sure that each test covered only one method or functionality. We also gave each test a clear, understandable name that makes it plain which method is being tested.

Code Coverage

According to code coverage reporting tools in Intellij, we achieved 46% line coverage and 36% branch coverage, mostly due to UI classes and functionality regarding rendering drawing entities or UI elements to the screen. Excluding the UI package and classes, we get around 64% line coverage and 43% branch coverage, with further drawing and UI related functions weighing down the numbers. Furthermore, since GameLoop is composed of classes from other packages, testing those other classes is effectively testing the GameLoop class itself.

Some features created in Java Swing could not be used with certainty due to JUnit tests only testing by method/function. Code coverage in those classes was low, due to a lack of focussed testing in them. Some functionality in InputHandler was not exercised since they were simple stubs required by the KeyListener interface. Functional testing would be more effective in testing these classes than structural testing especially regarding UI elements and how they look, and the level of usability.

Findings

Before we entered the testing phase, we had done a fairly thorough refactoring and a lot of functional testing to verify that the requirements laid out before were being met by the game. Functional testing had elicited many bugs in terms of how the game felt and played. Refactoring had elicited many issues/bugs and areas that had its code quality improved.

From the testing phase, the only production code we directly needed to change was that the DoorKey class was in the wrong package and needed to be moved into the Entities package. Additionally some getter and setter methods were needed in order to test parts of the production code more thoroughly.

We learned that the downstream importance of encapsulation is that it creates easier testing. More loosely coupled classes with clear boundaries and tight cohesion, communicating over well defined interfaces, make for easier creation of tests. We also learned that structural testing is useful, but should be used in conjunction with functional testing, especially regarding UI elements, as structural testing has difficulty in revealing issues with GUis.

We also learned about how to write JUnit tests, with unit tests and integration tests. Sometimes it was difficult to test private methods or well encapsulated classes, however in conjunction with other methods of testing, the encapsulated code would be exercised.

Another lesson learned was to ensure our created methods are implemented in a test friendly way. Because of the design of some of our methods, testing them became challenging and time consuming due to not having any returns, or they didn't manipulate code in a way that a JUnit test could verify. Going forward, by taking the time to make methods test friendly we will save time in future testing phases and ensure proper code test coverage for all methods.