# CPEN522: Assignment1

Jiahui Tang 77383255
Srikanthan Thivaharan 41497900

2019/01/31

## 1 Exercise 1.1

After we finished setting up the working environment for Git and Eclipse, we were able to pull our homework repo from GitHub to local. Then we followed the instructions on the assignment to compile, test, and run Jpacman framework. Since both of us wanted to set up on our own computer, we tried to add, commit, and push the framework files and changes to our Git repo separately. Jiahui set up the environment on Master branch, and Srikanthan created a branch called Sribranch, so we can work on our lab assignment together. The following is our answers for Exercise 1.2 to Exercise 1.4.

## 2 Exercise 1.2: Exploratory Testing

First of all, we want to introduce how did we explore the main features of JPacman Framework, and what kind of exploratory tour we took, then we will describe three main features (three scenarios) in charts.

Since it is a ready-made framework, so there are some documents providing the background and story about this framework. We started our "Guidebook tour" from reading those documents in the folder: **/docs**. Then we did a "Landmark tour" which means exploring the key features of JPacman. We started a game and played it for a while in order to know the whole story and to see if they realized the same functions as described in their documents. Finally, we tried to test the game by inputting some random and bad inputs and forced the framework to exhibit its capabilities. By doing those, we eventually understood the main features of JPacman.

As shown in the three tables below, we focused on three different scenarios during our exploratory:

a) The basic function to start, stop and exit a JPacman Game;

b) When a player moves over a tile containing food, the player earns points and the food disappears;

c) If a player and a ghost meet at the same cell, the player dies and the game is over.

Table 1: Start/Stop/Exit JPacman

| | |
|---|---|
| **Actor** | Normal Player |
| **Purpose** | JPacman game shall be started when pressing the start button after launching the game and shall be paused if the user clicks the stop button. The user is also able to exit the game by clicking the exit button. This test is to evaluate if the launch/start/stop/exit function works, and to make sure there is no any crash during the operations. |
| **Setup** | A Macbook Pro with 8Gb memory, the JPacman framework, the Eclipse IDE with all the related environment including JAVA, Maven and so on. |
| **Priority** | **High**. This is the fundamental function for the game. If the function fails, the user is unable to conduct other operations. |

| | |
|---|---|
| **Reference** | Not Applicable |
| **Activities** | 1. The user launches JPacman game. **Observations**: The game shall be open as a GUI application. The game window shall include the correct map, the status panel, a start button, a disabled stop button, and an exit button.<br>2. The user clicks the start button at the bottom of the window or presses "S" on the keyboard. **Observations**: The JPacMan game is started the ghosts move around. The status panel shows that the current status is "Playing". The "Stop" button becomes clickable while the "Start" button is disabled.<br>3. The user clicks the stop button at the bottom of the window or presses "Q" on the keyboard. **Observations**: The JPacMan game is stopped and the ghosts become standstill. The "Start" button becomes clickable while the "Stop" button is disabled.<br>4. The user presses the exit button at the bottom of the window or clicks "X" on the keyboard. **Observations**: The game window disappears. |

Table 2: The player eats the food and obtains the point

| | |
|---|---|
| **Actor** | Normal Player |
| **Purpose** | According to the designed story of JPacman game, the user shall be able to move over the tunnel and eat the foods on the tiles. As a result, the player will earn the corresponding points. The purpose of this test is to know if the user is able to move over tiles containing foods and eat that foods, and to know whether the point is increased after the moves. |
| **Setup** | A Macbook Pro with 8Gb memory, the JPacman framework, the Eclipse IDE with all the related environment including JAVA, Maven and so on. |
| **Priority** | **High**. As a basic user operation, this function is used very frequently during the game. If the test fails, the user cannot successfully finish the game. |
| **Reference** | Not Applicable |
| **Activities** | 1. The user launches JPacman game. **Observations**: The game shall be open as a GUI application.<br>2. The user clicks the start button and moves the Pacman over left side tails containing food by pressing left button or "H" button on the keyboard. **Observations**: The Pacman move over the left side tails and the point panel on the top of window increments by 10 points for each food the Pacman ate.<br>3. The user moves the Pacman back to the original position by pressing the right button or "L" button on the keyboard. **Observations**: The Pacman returns to the start position. The point panel remains the same. The foods on the left side tails disappear.<br>4. Repeat the same process as step2 to step3 for the right side tails. **Observations**: The foods on the right side tiles disappear and the point panel increments by 10 for each food the Pacman ate. |

Table 3: The player meets a ghost and loses the game

| | |
|---|---|
| **Actor** | Normal Player |
| **Purpose** | According to the story of this game, when a player and a ghost meet at the same cell, the player dies and the game is over. For this test, the purpose is to know whether the game will be ended when the play meets a ghost. If the test fails, then the game is infinite and meaningless to the users. |

| | |
|---|---|
| **Setup** | A Macbook Pro with 8Gb memory, the JPacman framework, the Eclipse IDE with all the related environment including JAVA, Maven and so on. |
| **Priority** | **High**. As a possible end of this game, this function is also used very frequently during the game, especially for the inexperienced players. |
| **Reference** | Not Applicable |
| **Activities** | 1. The user launches JPacman game. **Observations**: The game shall be open as a GUI application.<br>2. The user moves Pacman into the cell where the ghost is. **Observations**: The Pacman disappears and the game is stopped. Both "Stop" and "Start" buttons are disabled. The game status on the top of the window shows "You have lost :-(".<br>3. Restart a game and let Pacman stay at the start position. Wait for the ghost moves into the cell where the player is. **Observations**: The Pacman disappears and the game is stopped. Both "Stop" and "Start" buttons are disabled. The game status on the top of the window shows "You have lost :-(". |

Based on our exploration, we found there are some possible missing features for a game like JPacman. We listed all the possible ideas as below:

(i) If the user has extra life (such as three hearts), then the game will be more friendly for the beginners.

(ii) Record the time the user plays the game and present the best result (the minimum time the user used to finish the game).

(iii) For the current version, the users have to exit the game and restart it again after they lost the game, so it is a good idea to add a "restart" button.

(iv) It is better to add some sound elements into the game to make it more interesting. For example, a new sound is played when the game is successfully started and the sound is paused when the game is stopped. This will help the user to understand the status of the game. Another example can be this: A new "dong" sound is playing when the Pacman eats the food to notify the user that the point is increased.

(v) When the user finishes the game, it is better to show a pop-up window with some words like "Congratulations" or "Try Again!".

(vi) The game is supposed to realize multiple levels and the user will start a new level when they finish one. However, currently, this version of JPacman does not have different maps.

# 3 Exercise 1.3: Running the Test Suite

As listed below, there are 10 test classes and 39 test cases in this unit test suite. We can also get the number of test cases directly on Eclipse as shown in Figure 1.

(1) AbstractAcceptanceTest (Acceptance test): 1 test case

(2) MoveGhostStoryTest (Acceptance test): 5 test cases

(3) FactoryIntegrationTest (Integration test): 1 test case

(4) GameTest (Unit test for Model): 12 test cases

(5) SpriteTest (Unit test for Model): 5 test cases

(6) BoardTileAtTest (Unit test for Model): 8 test cases

(7) PointManagerTest (Unit test for Model): 3 test cases

(8) MainUISmokeTest (UI Smoke test): 1 test case

(9) ButtonPanelTest (UI Smoke test): 1 test case

(10) ImageLoaderTest (Unite test for View): 2 test cases

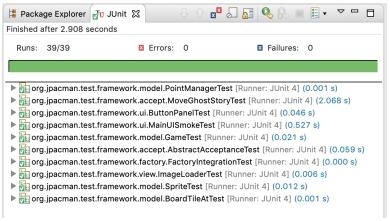


Figure 1: JUnit Running Result - All Pass.

This time, we decided to change the test case testPlayer in ImageLoaderTest class. This test case is to test whether images for players are properly loaded or not. Normally, there are different types of player images (4 images for each "UP", "DOWN", "LEFT" and "RIGHT" direction). All those images should be loaded and resized properly. For this test case, it tests whether the loaded PacMan1Up and PacMan1Down images are the same or not. The desired result should be not the same, so it used "assertNotSame(up, down);". Then we changed it into the same images, so it will fail. The code changes are shown in Figure 2. The results are depicted in Figure 3. Failure Trace shows that "java.lang.AssertionError: expected not same" and it also gives the line number of the failure.

```
/**
 * Are images for player properly loaded?
 */
@Test public void testPlayer() {
    Image up = imf.player(Direction.UP, 1);
    Image down = imf.player(Direction.DOWN, 1);
    assertNotSame(up, down);
}
```

```
/**
 * Are images for player properly loaded?
 */
@Test public void testPlayer() {
    Image down = imf.player(Direction.DOWN, 1);
    Image down_1 = imf.player(Direction.DOWN, 1);
    assertNotSame(down, down_1);
}
```

Figure 2: Code comparsion - before and after.

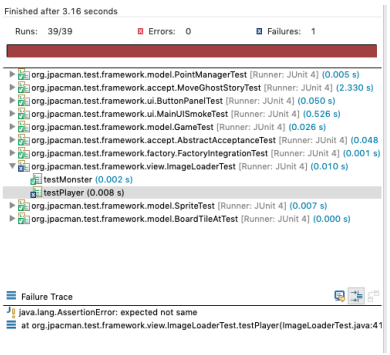


Figure 3: JUnit Running Result - 1 failure.

# 4 Exercise 1.4

For this exercise, we chose to look deeper into PointMangerTest, which is a unit test class for pointManager. According to the comments, the original purpose is to test the basic behavior of a point manager.

The coder writers wanted to test three different behaviors of a point manager: testAdd, testEatSomeFood, testEatAll. In the initialization phase, they created a point manager and mocked a player by using MockitJUnitRunner.class.

For testAdd, it is based on the fact that after we added food in the setup(), we still have enough food to eat, so allEaten() should be False in this case.

For testEadSomeFood, the story changes into this: we ate half of the food, then we still get the other half to be eaten, so allEaten() should be False and getFoodEaten() should equal to half of the total points.

For testEatAll, this test case assumes that the player ate all the food, then there is no any food left, so allEaten() should be True.

In our opinion, this is not a good test class. **First**, the starting point for this test class is from a behavioral perspective. Generally, it is difficult to fully consider all behaviors. For example, in the test class, they assumed that half of the food has been eaten, so why don't we test one third of food. This test class always miss some possible behaviors of pointManager. **Second**, this test class used addPointsToBoard() directly in the setup() without testing it. Moreover, there are 7 methods in the pointManager class (4 methods for pointManager and 3 override methods of IPointInspector). In this test class, they only used 4 methods (only used them instead of testing all of them). **Last**, it is better to add some reminding messages into assertion methods.

To our understanding, if we want to conduct unit tests for pointManager class, it is better to write one test case for each method in PointManager class (or maybe we can possibly combine some method together). At least we need to test all the methods in our test class. The place worth learning in this test class is the use of Mock. Because consumePointsOnBoard(Player p, int delta) requires a player, this test case uses Mock to initialize a player. However, in the pointManager class, there is also a consumePointsOnBoard() which does not need to use Mock, so we can directly test just as what we do for other methods.