COMP3211

Software Engineering

Group 1 – Unit Tests Line Coverage Report

Monopoly

Hong Kong Special Edition

Group 1	
Student ID	Student Name
22107416D	Kent Max CHANDRA
22055747D	Cheuk Tung George CHAR
24010799X	Tommaso Fabrizio COSTANTINI
23096373D	Xin DAI

CONTENTS

1. Introduction

Purpose of the Unit Tests Line Coverage Report

The main purpose of this **Unit Tests Line Coverage report** is to declare the percentage of line coverage achieved using Python Unit Test, additional information on running Python Unit Tests as well as examples showcasing the general implementation of the Unit Test code for our game **Monopoly Hong Kong Special Edition**.

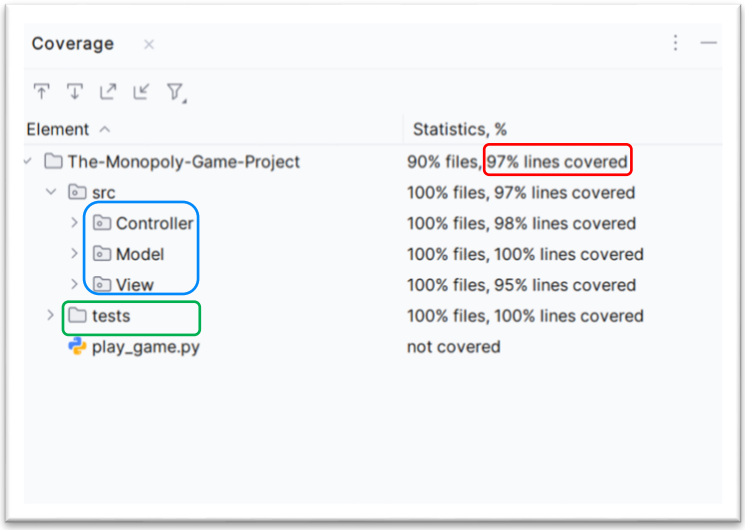
- The percentage line coverage achieved from Test Codes
- Instructions with regards to replicating the line coverage result using Python Unit Test using PyCharm IDE
- The Unit Test code examples written using Python Unit Test

II. Percentage Line Coverage

Percentage Line Coverage Result Declaration

This section provides a detailed statement of the results achieved by running Python Unit Test under PyCharm IDE using Python 3.11. I will also provide detailed statements with regards to the results and provide explanations for the percentages that the Python Unit Test had achieved.

The image below shows the result of line coverage displayed in PyCharm



The figure illustrates that in the source code of **Monopoly Hong Kong Special Edition**, there are multiple folders which are separated into 'Model', 'Controller' and 'View'. Outside of the source code, there is a file named 'tests', and this is where all the Python Unit Tests resides in. The 'play_game.py' acts as a executable file or in a another word, a run file, where the user would run the whole application(program).

From the figure, our Unit Test codes have achieved **97% line coverage** for the entirety of the program folder, which in the figure is called 'The-Monopoly-Game-Project', marked using a red square.

Notably, the line coverage for the 'Model' folder is 100%, while the others, namely, 'Controller' and the 'View' folder have 98% and 97% percent respectively.

The Unit Test has 100% coverage across all the folders under the 'src' (Source Code) folder, which are boxed using blue, overall achieving 90% file coverages in the project folder. The 'play_game.py' file, boxed by green is responsible for the rest of the missing 10% of file coverage. The reason why this file is not covered, as explain before, this file is equivalent to an application executable file, where it encapsulates all the: logics and data(Model), game management (Controller), and displays(View), hence

II. Percentage Line Coverage

the MVC model; essentially, all the tests implemented in the 'tests' folder applies directly to the implementation, logic flows, and management of the source code resided in the 'src' folder, which is the same entity that the file 'play_game.py' encapsulates.

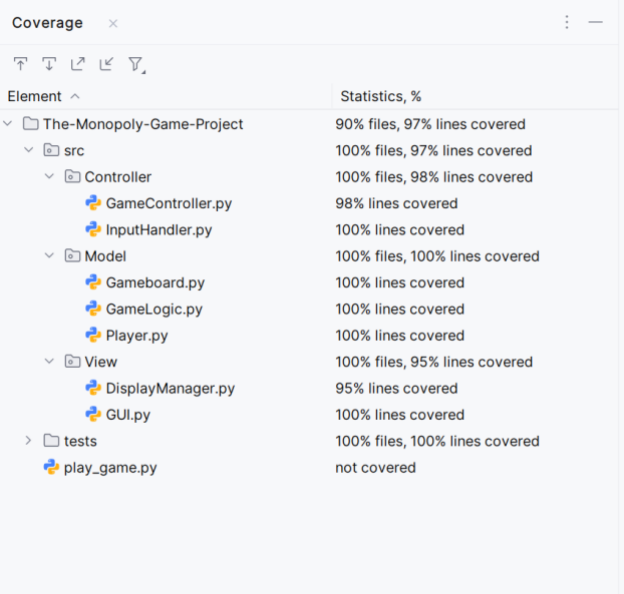
This figure shows the implementation of the file 'play_game.py'

```
from src.Controller.GameController import *

# Run the GUI
gui = GUI()
controller = GameController(gui)
gui.mainloop()
```

As you can see, this is the implementation of the file 'play_game.py', which only contains 4 lines of working code, encapsulating the whole source code of the game. This means that, the testing on this file is redundant as, firstly, there is no variables and outputs to check for its correctness, and the result of this program file is ambiguous, as it covers the entire program, and secondly, the other parts of the program are broken down into smaller, more distinguishable, testable, and the ones with definite outcomes. These are residing in the 'src' folder, tested by the codes in the 'tests' folder, which is the Unit Test. Therefore, testing the codes in the 'src' folder is sufficient, as it reflects the behaviours that a running 'play_game.py' file should have.

This diagram below shows more detailed break down of the line coverage in each separate folder



To be more specific, 'Model' file has every file covered, and all with 100% line coverage. 'Controller' folder has 2 files in total, and 'InputHandler.py' has 100% line coverage, while the other, which is 'GameController.py', has 98% percent. The last folder in 'src' folder is 'View', and it has 2 files a well, with 'GUI.py' reaching 100%, and 'DisplayManager.py' having 95% lines covered.

III. Replicate Python Unit Tests

Instructions to Run and Replicate Python Unit Tests

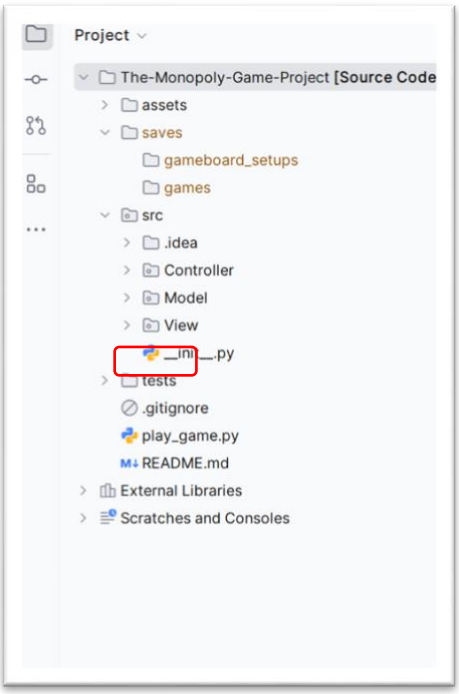
In this section, 2 methods' detailed instructions for replicating the Python Unit Tests that reproduce the result presented about are listed.

IMPORTANT: the following instructions are carried out in **PyCharm IDE Community or Professional Version.** To be able to replicate the result exactly, please consider using this specific IDE. The download link is provided below had you haven't downloaded the application already.

PyCharm Community and Professional Download Link:
https://www.jetbrains.com/pycharm/download/?section=windows

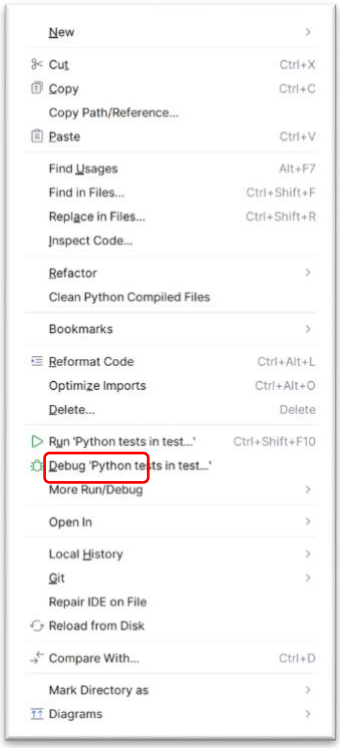
Method 1:

This method is more recommended as it is platform independent.

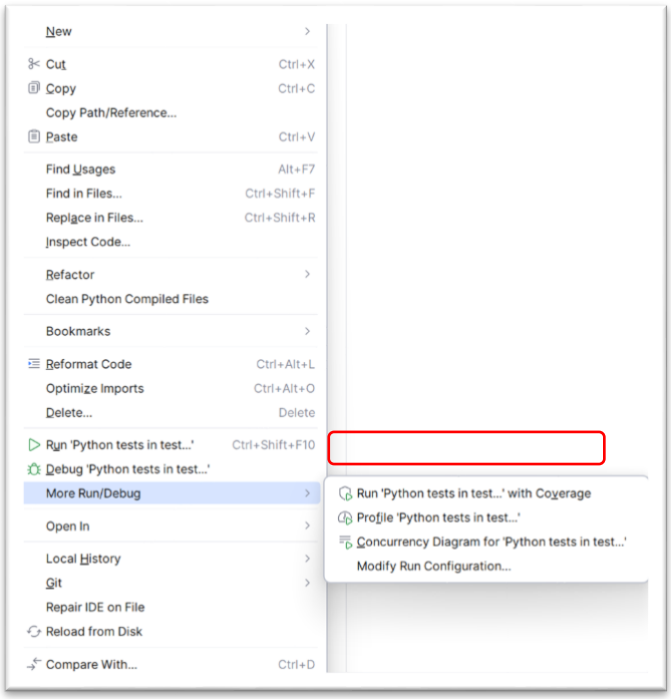


First, go to the Folder Navigation Bar and **<Right-Click>** the 'test' folder, boxed and highlighted using red in the image.

III. Replicate Python Unit Tests



Second, you will see a pop-up table like the one shown in the image on the left. <Left-Click> the highlighted region using red in the image which is called **More Run/Debug**.



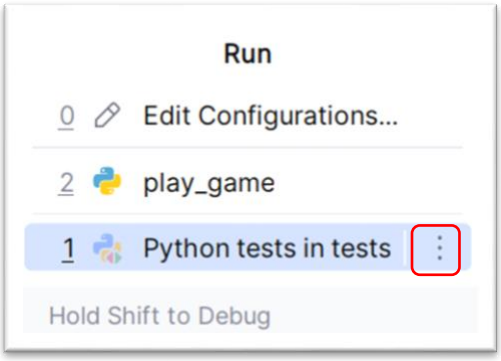
Finally, <Left-Click> on the highlighted area in red in the picture called **Run 'Python tests in test…'** to replicate the result of the Python Unit Test show in section 2 of this report earlier.

III. Replicate Python Unit Tests

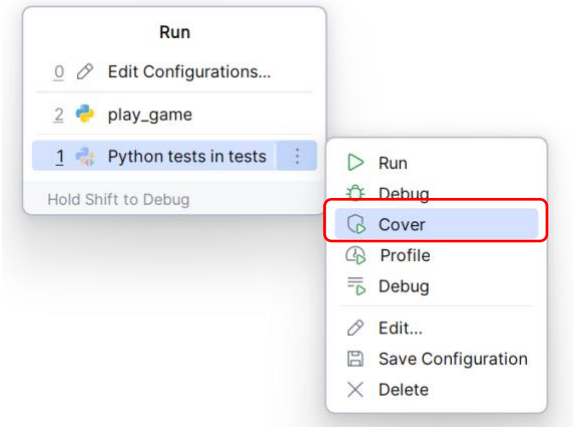
Method 2

This method requires you to use a **Windows Operating System** that supports the download of **PyCharm IDE** either **Professional** or **Community Version**.

Open the source code using PyCharm in Windows computer, and **first** use keyboard and enter <Shift+Alt+F10>, and you will see this top-up table in PyCharm, <Left-Click> the three vertical dots highlighted in red near the entry called **Python tests in tests**



Secondly, after clicking the three vertical dots, you will see a pop-up table, and <Left-Click> **Cover** to show the line coverage result.



Execution Recommendation: As mentioned before, **method 1** is more reliable than method 2, as in method 2, sometimes on a new computer, the **Python tests in tests** entry may not show up in the pop-up table.

IV. Unit Tests Example Showcase

Section Overview

This section focuses on showcasing two examples on how the Unit Tests are written for automatically testing the expected outcomes and correct behavior of the program code. Before proceeding to showcasing the test code, one important condition to declare is that all of the tests are done using the python standard library **unittest**, in which I have **TestCase**, and I have also used **patch** and **MagicMock** in **unittest.mock**

```
from unittest import TestCase
```

```
from unittest.mock import patch, MagicMock
```

Example I

The figure below shows the first example of the Python Unit Test code written

```
#Test raises error when a json file is corrupted when loading a board
def test_invalid_board_json_read(self):
    test_controller = GameController(TestGameController.gui)
    invalid_json_string = "{'key': 'value'"
    directory = '../saves/gameboard_setups'
    file_name = "invalid_json"
    file_path = os.path.join(directory, f"{file_name}.json")
    os.makedirs(directory, exist_ok=True)
    with open(file_path, 'w') as json_file:
        json_file.write(invalid_json_string)

    test_controller.load_gameboard(file_name)
    self.assertRaises(json.JSONDecodeError)

    os.remove(file_path)
```

Comment stating what the test is about

Assertion for checking expected behavior

IV. Unit Tests Example Showcase

Example 2

The figure below shows the second example 2 of the Python Unit Test code written

```
# test different landings in sequence (buy property(but not enough money) -> rent -> not buy property -> go
@patch('tkinter.Tk.wait_variable')
def test_land_and_complete_round(self, mock_wait_variable):
    test_controller = GameController(TestGameController.gui)
    player_this_turn = self.game_set_up(test_controller,  seed: 4)

    # Buy but not enough Money
    mock_wait_variable.side_effect = lambda var: var.set("buy")
    player_this_turn.set_current_money(300)
    player_this_turn.set_square(1)
    test_controller.land_and_complete_round(test_controller.board.tiles[1], player_this_turn)
    self.assertEqual(player_this_turn.get_current_money(),  second: 300)

    # Rent
    test_controller.board.tiles[2].set_owner(player_this_turn.get_name())
    player_this_turn.set_square(2)
    test_controller.land_and_complete_round(test_controller.board.tiles[2], player_this_turn)
    self.assertEqual(player_this_turn.get_current_money(),  second: 300)

    # No Buy
    player_this_turn.set_current_money(1500)
    mock_wait_variable.side_effect = lambda var: var.set("not_buy")
    test_controller.land_and_complete_round(test_controller.board.tiles[4], player_this_turn)
    self.assertEqual(player_this_turn.get_current_money(),  second: 1500)

    # Go
    player_this_turn.set_square(18)
    test_controller.game_logic.player_move(2, player_this_turn, test_controller.board)
    test_controller.land_and_complete_round(test_controller.board.tiles[0], player_this_turn)
    self.assertEqual(player_this_turn.get_current_money(),  second: 1650)

    # Income Tax
    player_this_turn.set_square(3)
    player_this_turn.set_current_money(1000)
    test_controller.land_and_complete_round(test_controller.board.tiles[3], player_this_turn)
    self.assertLess(player_this_turn.get_current_money(),  b: 1000)
```

NOTE: The ‘second’ parameter name is auto generated by PyCharm, which is incorrect.

When there are multiple test cases in a single test function (in this case it is called ‘test_land_and_complete_round’) because they all need the same initialization:

Red the comments that indicates what the test is about, some separate the test cases for better readability.

IV. Unit Tests Example Showcase

Orange are the assertions specifically for that testcase which check whether the test meets the expected results, which consequently means the source code is behaving in a way that is logically correct according to the pre-defined requirements.