**Introduction:**

The classes vital to the programs operation have been suitably tested, these are typically manipulated within the main.py Python file. For each of the relevant important classes, including classes from files:

1. ai\_agent.py
2. bank.py
3. game.py
4. player.py
5. timer.py

A suitable unittest test class has been created, in relevant order from top-to-bottom, this includes the unittest classes:

1. testAI.py
2. testBank.py
3. testGame.py
4. testPlayer.py
5. testTimer.py

In order to run these unittest classes, first I navigate to the location where these tests have been stored (aka. “tests/unit”, located from the root directory of the project) and using the following command:

python –m unittest testPlayer testGame testTimer testBank testAI

This will run each test in the order specified in the command above; during running these unittest classes, if any test within any class should fail, the terminal output would respond accordingly and halt progressing to the end.

As of right now, running these tests completely succeeds without halting or error; additional information can be seen, including printed output information where relevant, primarily seen from the testAI.py class giving information such as “Out of valid houses”, etc. Information shown inside the console is not relevant to the test being successful; the text “Out of valid houses” does not indicate failure, rather that the test is progressing where certain methods used/tested return such output seen in the console.

If the unit tests were to fail, we would have to expect undefined behaviour and results from the program overall; given all the tests succeed without problem, problematic code is unlikely to be found within any of the tested classes, and more likely with how these classes are being used in the main program itself (such as with main.py).

**ai\_agent.py -> testAI.py**

The first unit test revolves around testing the ai\_agent.py class; this class controls the non-playable characters found within the game. Due to not being directly controlled by any player, decisions such as trading, placing roads, houses, etc. have to be decided using methods, accepting variables and determining the proper response.

The test begins by initialising the AI class itself (found within ai\_agent.py), where we first confirm weather the newly created player is considered to be an “AI” user. The actual method used to determine this has been re-created, and the expected output is measured.

The main method within the ai\_agent.py class involves the decision making. There are 5 different game\_states defined within the class, for each game\_state specific decisions for the AI would be performed; the initial house and road placement, further house and road placements and the default method. Other methods within the class include, such as trading with other players (50% chance of being successful when called), confirming the requested house position is not considered “too close” to other player houses, trading with the bank, returning a list of the AI’s resources and removing a list of resources from the player, are primarily used as complimentary methods to the main makeDecision() method.

**bank.py -> testBank.py**

The bank class is an extremely simple class, involving storing, returning, adding, removing resources dedicated to a non-playable “Bank”, along with saving these resources and the use of dev cards.

After the bank had been initialised within out unit test class, we first will confirm the resources inside the Bank with what would should expect. A single resource is added to the total, for which the change in frequency is measured, then 10 resources are added using a slightly different method that includes an amount parameter. After, a single resource is removed, where the change in frequency gets measured for the expected outcome. We then simulate and measure the change in resources from placement of infrastructure; it is important to determine that each infrastructure placement corresponds with the expected change in the resources amount. We then specify a specific set of dev\_cards and confirm that after being shuffled, the original combination is no longer being used.

The final method being tested is with regard to saving the Bank information; the expected save information would include a combination of dev\_cards and resources that the Bank currently owns.

**game.py -> testGame.py**

The game class gets initialised using information such as a lit of the player’s names and a list of available colours, and combines each player name with one of the available colors.

The class includes a method for checking weather the game is considered “Over” from the amount of victory points in circulation, for one player to reach “10” victory\_points, the method would return True, otherwise False.

**player.py -> testPlayer.py**

The player class defines each player currently inside of the game, storing important information for each player, such as resource, amount of victory points, dev cards and even the dice functionality. The player class is intialised using a provided name and colour.

At this point, the player has just been created, therefore lacks the victory points required to win; even after giving the player 9 additional victory points, the has\_won() method will only return true given the player currently has 10 victory points.

The amount of knights played should be incremented from the add\_knight() method; starting from the initialised 0, we would expect a return of “1” after we call the add\_knight() method only once. The next method involves testing the trade ratios for the player; we can expect exactly what values should be present after initialisation, meaning further manipulation (such as updating a trade ratio) can also be expected.

The resources available to the player should be “0” of each after first being created; at this point, we have permission to create any infrastructure (where the permission is lost after a threshold is reached), but lack the resources to do so. After manually adding the required resources for each infrastructure, the method should conversely return True, we then attempt to remove a resource that would (as expected) invalidate the has\_enough\_resources() method, making it return False instead.

To test the dev cards being used by the player, we attempt to manually add a specific list of different development cards, then determine weather the player has properly received such development cards, from either the total list of cards, of by frequency of each type of dev card.

The final method within the unit test checks the dice rolling mechanism, where we first affirm that we have been manually able to change the roll of each die, confirm that it has changed from before and the total value of each die added up (from the manually provided values).

These provided values are considered invalid, therefore should not ever get repeated when using the proper roll\_dice() method.

**timer.py -> testTimer.py**

The final unit test class involves the timer.py class. This very simple class involves methods for pausing and unpausing the timer along with saving the current timer information inside a .txt file.

We are expecting the timer to get initialised to 15 minutes in milliseconds. We then attempt to use the pause and unpause methods and confirm the current status of the timer after each method.

We then save the timer using specified values (including setting a specific time, pausing the timer and setting the time remaining). After saving these values, they should be stored within the file timer\_data.txt, to confirm that these values have actually been saved, first we attempt to change the current timer values before attempting to load the timer\_data.txt file and confirming the specified saved values get properly loaded into the timer.