UNIVERSITY OF SUSSEX

PROTECTOR OF VALORIA

Candidate Number: 276270

1. Introduction

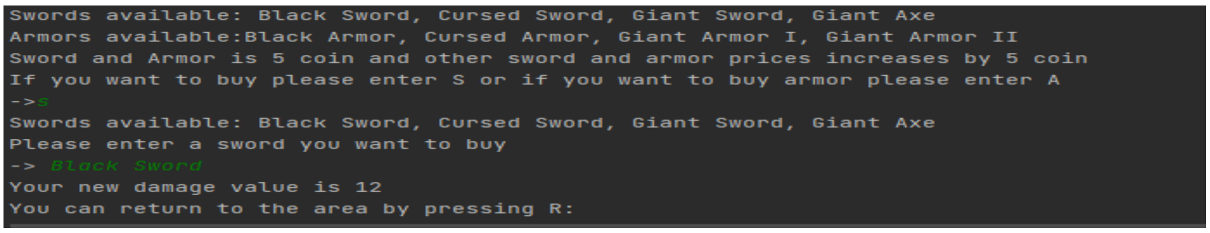
1.1Brief overview of the Protector of Valoria game

Protector of Valoria is an adventure game where users will be able to fight with monsters and at the same time utilize their character by buying some items via the market. In the game, users can move around rooms. These rooms are represented with direction names. Each room is filled with monsters. after the user kills these monsters, the person playing the game will be able to collect coins and potions. It is your consumable option in order to collect the loot. The potion is used in order to refill our hero's health. At the same time, coins are going to be used in order to increase our hero's health and attack permanently after we buy those items shown on the market. There are 10 rooms which are south, west, north, east, northeast, northwest, southeast, southwest, market, and safe zone. Users can suppose that the base is surrounded by those rooms and the market is sub room of the safe zone, which makes 10 rooms eventually. Southwest is a room with the final boss. Once the user has a sufficient level can face the final boss. The main condition to meet the level is to kill every monster in the room so that you will be able to have an adequate level to challenge the boss and unlock this room too. Even if you kill the monster in the room, you can revisit here again and again but you are going to find it empty, which will be up to the person playing the game how he is going to move. Killing Monster is going to provide XP as well as it provides money and coins which are optional choices. When the user manages to kill the boss in the southwest room, you are going to meet the win condition, as well.

2.Gameplay Instructions

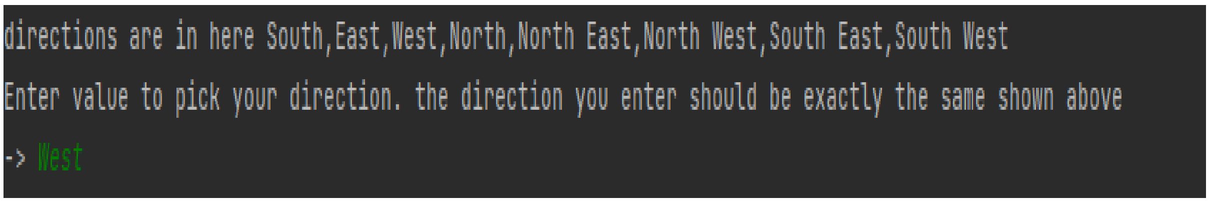
ProtectorOfValoria UnitTest.py   ProtectorOfValoria.py

Two different files are provided. "ProtectorOfValoria.py" can directly work via Windows shell. In addition to that, you need to move both files to PyCharm so that you will be able to access unit test codes, as well. When you run the code "ProtectorOfValoria.py" file, the game log file is going to occur for PyCharm and Windows shell too. You will have an opportunity to track your loggings. The most important advice on the game is that when the program asks where you will go and buy something from the market, the name of the direction and armor or sword name should be exactly matched because it is case sensitive. There are two different figures shown below for the sake of the user.



```
Swords available: Black Sword, Cursed Sword, Giant Sword, Giant Axe
Armors available:Black Armor, Cursed Armor, Giant Armor I, Giant Armor II
Sword and Armor is 5 coin and other sword and armor prices increases by 5 coin
If you want to buy please enter S or if you want to buy armor please enter A
->S
Swords available: Black Sword, Cursed Sword, Giant Sword, Giant Axe
Please enter a sword you want to buy
-> Black Sword
Your new damage value is 12
You can return to the area by pressing R:
```

Figure 1.



```
directions are in here South,East,West,North,North East,North West,South East,South West
Enter value to pick your direction. the direction you enter should be exactly the same shown above
-> West
```

Figure 2

Otherwise, the user will not be able to go to the place or get the items he wants, which is only valid for these circumstances. In other pressing button options are not case-sensitive, at all.

3.Game Map

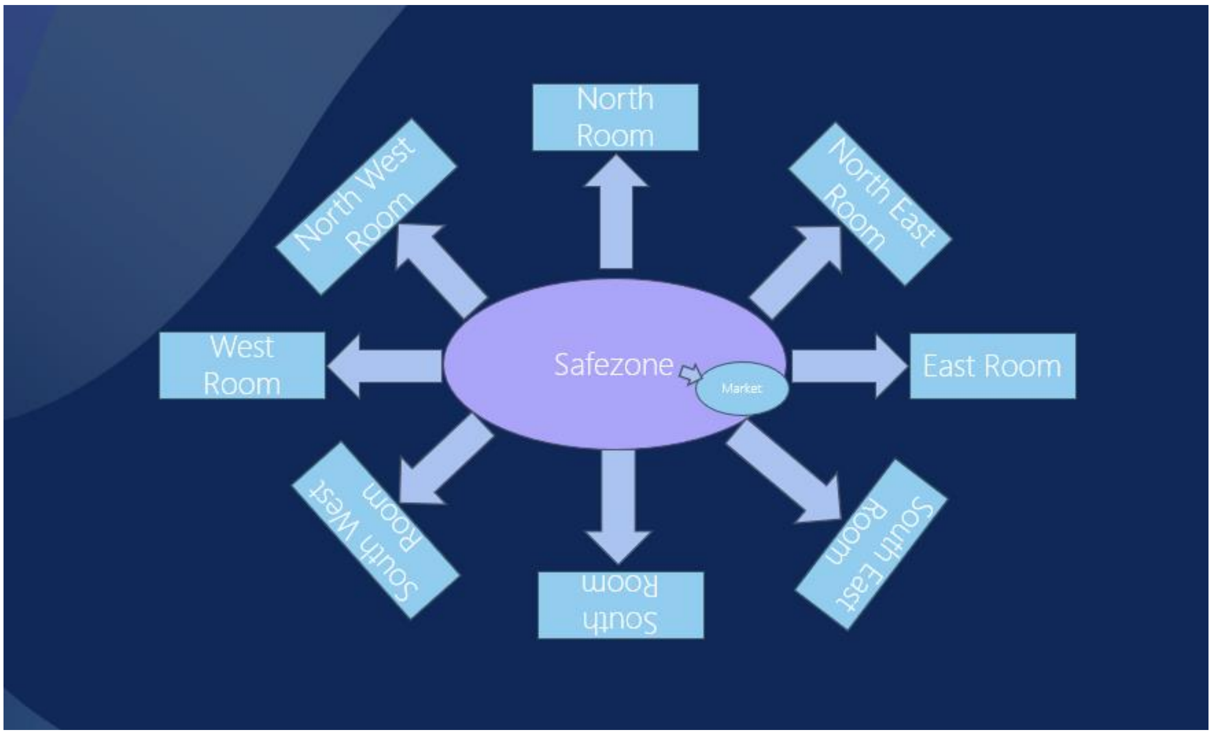


Figure 3

In Figure 3, you can see the map. So, the southwest room is a place where the boss is located and requires a certain level so as to unlock it. The market looks like a sub-room of the safe zone. User needs to go back and forth to the base in order to wander around map, which is a place connecting every room to each other as well as the market is connected to here.

4.UML Class Diagram

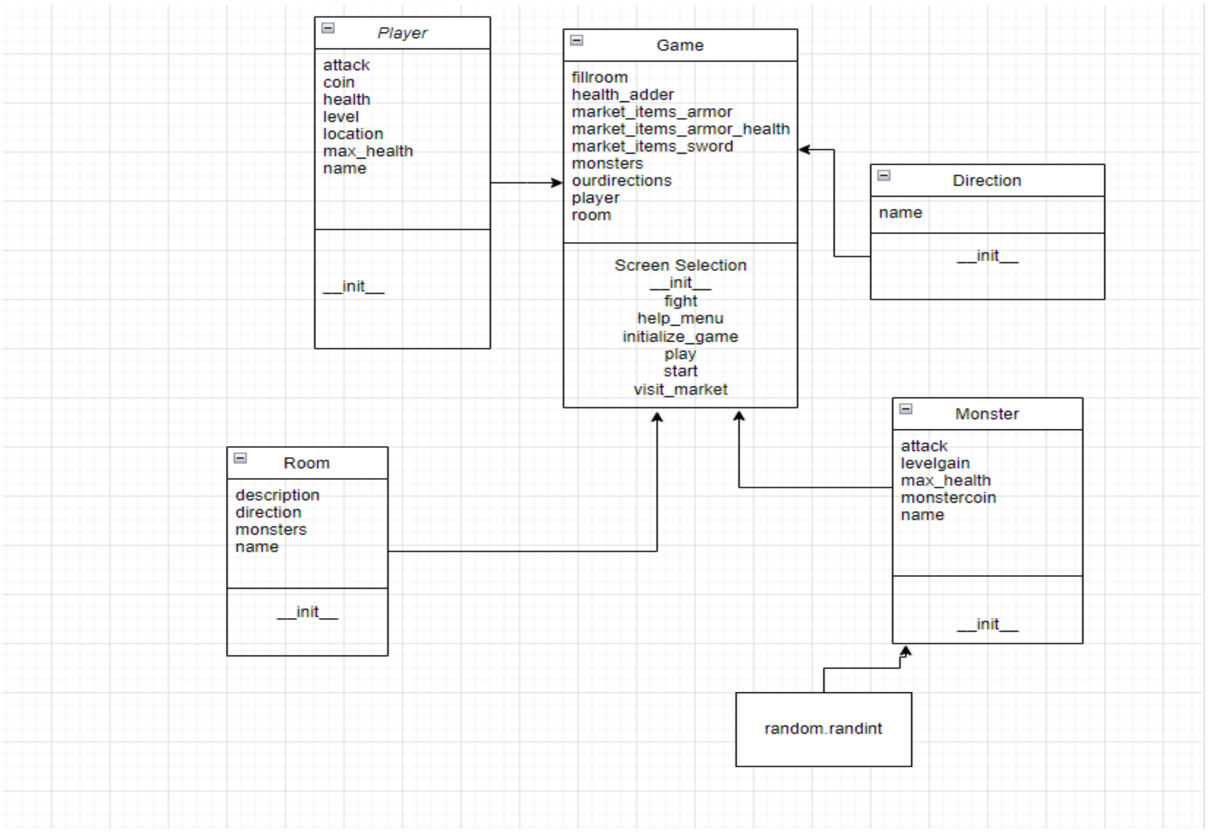


Figure 4

5. Informations about Classes in the Code

The code was built from scratch and then each function and line of code was carefully reviewed. The code's class and its functionality will be shared.

5.1. Player Class

Represents the game's player traits. Its attributes involve attack, coin, location, level, max_health, and health. These variables are set up with default values via the constructor.

5.2 Monster Class

In the game, it represents a monster. Its features consist of monstercoin, attack, max_health, and level gain.

5.3 Room Class

Represents a room in the game. It has attributes for name, description, a list of monsters in the room, and directions to other rooms.

5.4 Direction Class

Represents a direction in the game (e.g., South, East). These features are set by the constructor with the given parameters.

5.5 Game Class

The primary class that is in control of managing the game's flow is this one. It includes attributes like market item dictionaries, ourdirections, health_adder, and monster lists. It includes operations to set up and launch the game, control player actions (such as moving and fighting), entry the market, and show a help menu. The start method creates the game, while the initialize_game method sets the starting state for the game. The fight method simulates player-versus-monster combat, while the play method controls the main gameplay loop. The player may buy items from the market by using the visit_market method. Screen selection method is also vital because It offers choices for launching the game, going to the help menu, and quitting it. Def start and def play methods have tasks providing the player with an introduction and gameplay guidelines so they can choose how to proceed and make decisions in the game. Def help menu provide a detailed guideline about the game, which is reccomended to user to read.

5.Interesting Design Features

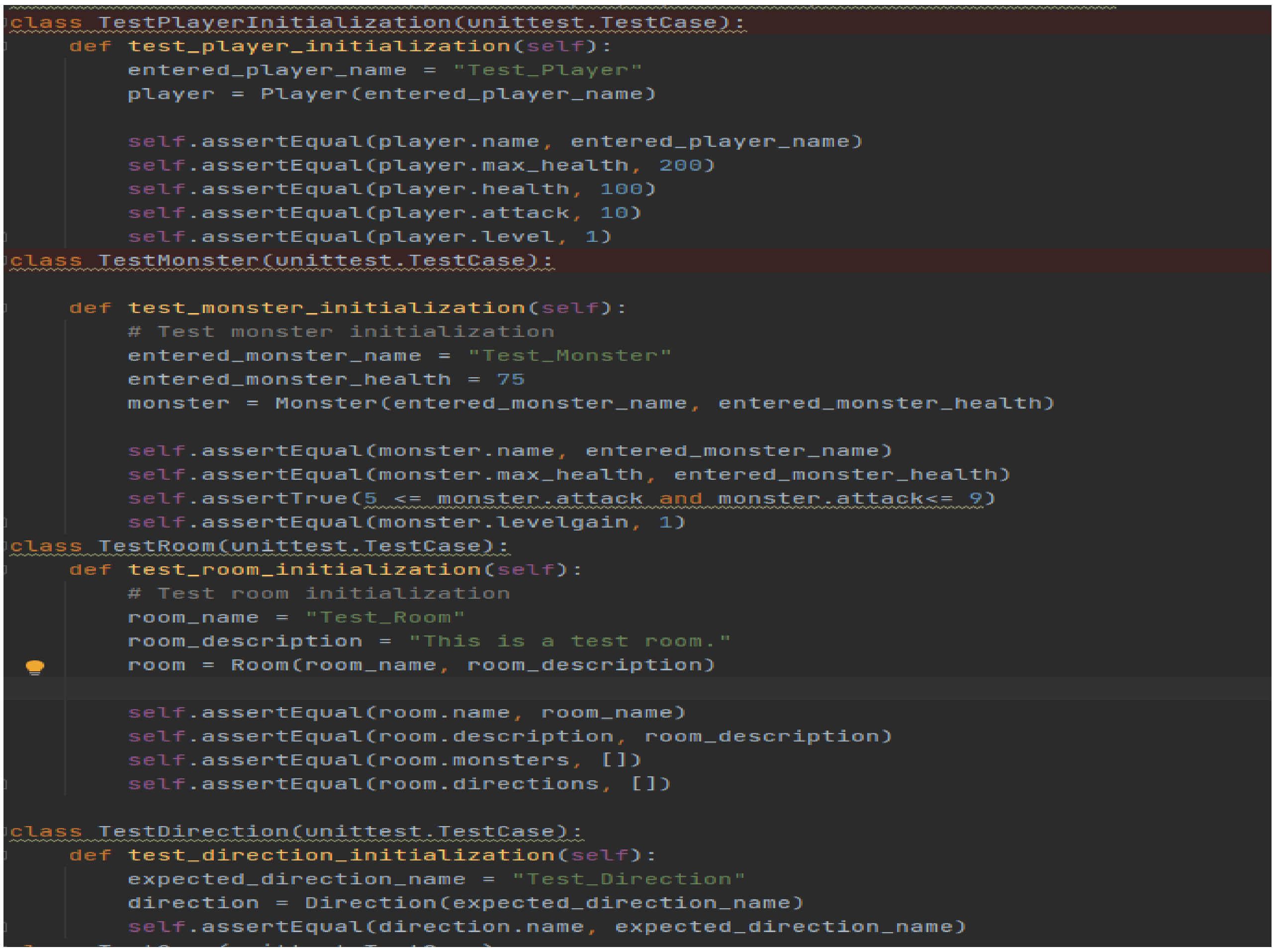
One of the interesting design features is the "fight" function. It allows users to fight manually with monsters. Every hit is felt so that sense of fight increases thanks to that. Also, the user can refuse to attack and push "p", which results in getting damage from the monster rather than attacking. Another interesting thing is after you defeat the monster, you can loot potions and money to be able to spend those coins on the market. "visit_market" is also helpful when users buy something from the market. So, the item bought is going to be removed so that the user will be able to follow easily what is left in the store. In addition to that, if the user boss fight room has a lock. The only way to open the door is to have a sufficient level, which is essential to reach the winning condition in the game.

6.Encountered Setbacks

The biggest deal of the project was to fill rooms with monsters and to check their existence in the room, which was the most challenging part of the code. The data coming from the monster class must be processed. The fight function also should be integrated properly. That's why, room and other function relations were the most challenging part. after understating the relationship, everything started looking easier than before. In addition to that, certain things could be better of course. In this part, it will be mentioned which things should be changed and so on. The looting system could be better by adding a backpack but it would make the algorithm much more complicated because all of them should be stored in a certain dictionary with its object name and amount, which would mean comprehensive development. Sword and Armor prices should be shown rather than saying every sword and Armor price increase by 5.

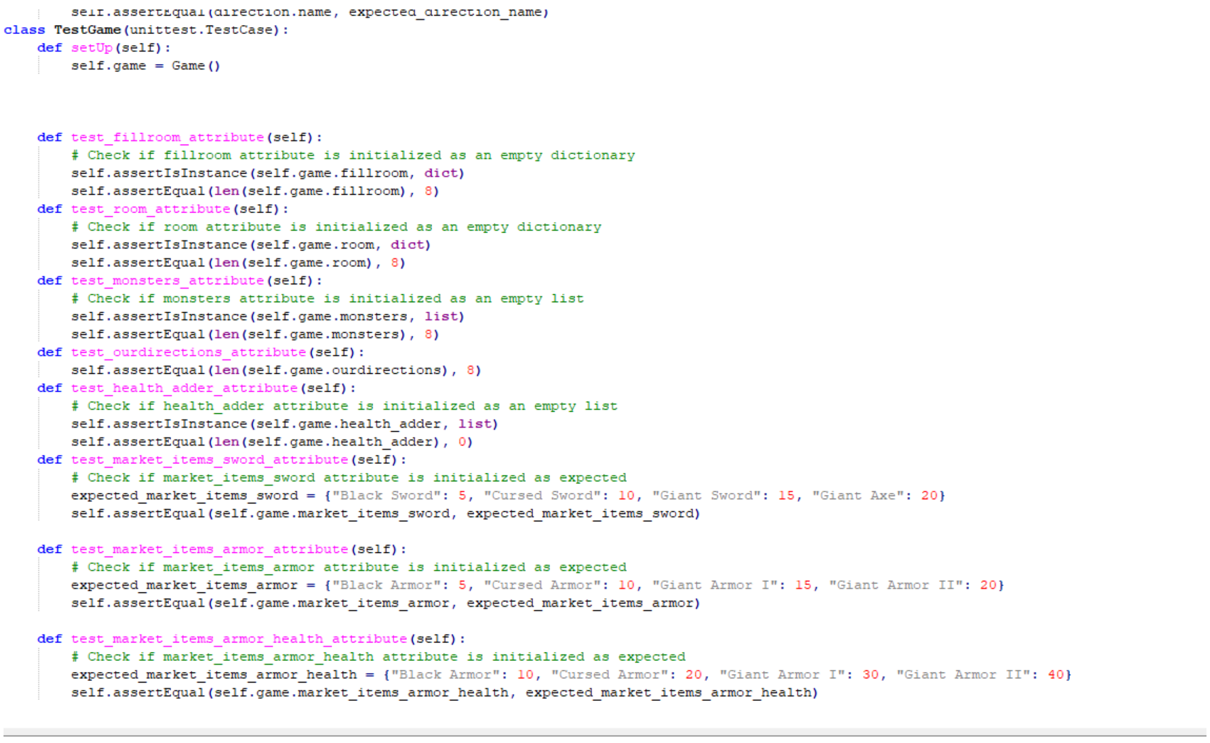
7.Testing Process

every class constructor element except for the game class is tested, which is necessary to verify constructor variables work properly. You can also examine unit test functions in Figure 5.



```python
class TestPlayerInitialization(unittest.TestCase):
    def test_player_initialization(self):
        entered_player_name = "Test_Player"
        player = Player(entered_player_name)

        self.assertEqual(player.name, entered_player_name)
        self.assertEqual(player.max_health, 200)
        self.assertEqual(player.health, 100)
        self.assertEqual(player.attack, 10)
        self.assertEqual(player.level, 1)
class TestMonster(unittest.TestCase):

    def test_monster_initialization(self):
        # Test monster initialization
        entered_monster_name = "Test_Monster"
        entered_monster_health = 75
        monster = Monster(entered_monster_name, entered_monster_health)

        self.assertEqual(monster.name, entered_monster_name)
        self.assertEqual(monster.max_health, entered_monster_health)
        self.assertTrue(5 <= monster.attack and monster.attack<= 9)
        self.assertEqual(monster.levelgain, 1)
class TestRoom(unittest.TestCase):
    def test_room_initialization(self):
        # Test room initialization
        room_name = "Test_Room"
        room_description = "This is a test room."
        room = Room(room_name, room_description)

        self.assertEqual(room.name, room_name)
        self.assertEqual(room.description, room_description)
        self.assertEqual(room.monsters, [])
        self.assertEqual(room.directions, [])

class TestDirection(unittest.TestCase):
    def test_direction_initialization(self):
        expected_direction_name = "Test_Direction"
        direction = Direction(expected_direction_name)
        self.assertEqual(direction.name, expected_direction_name)
```

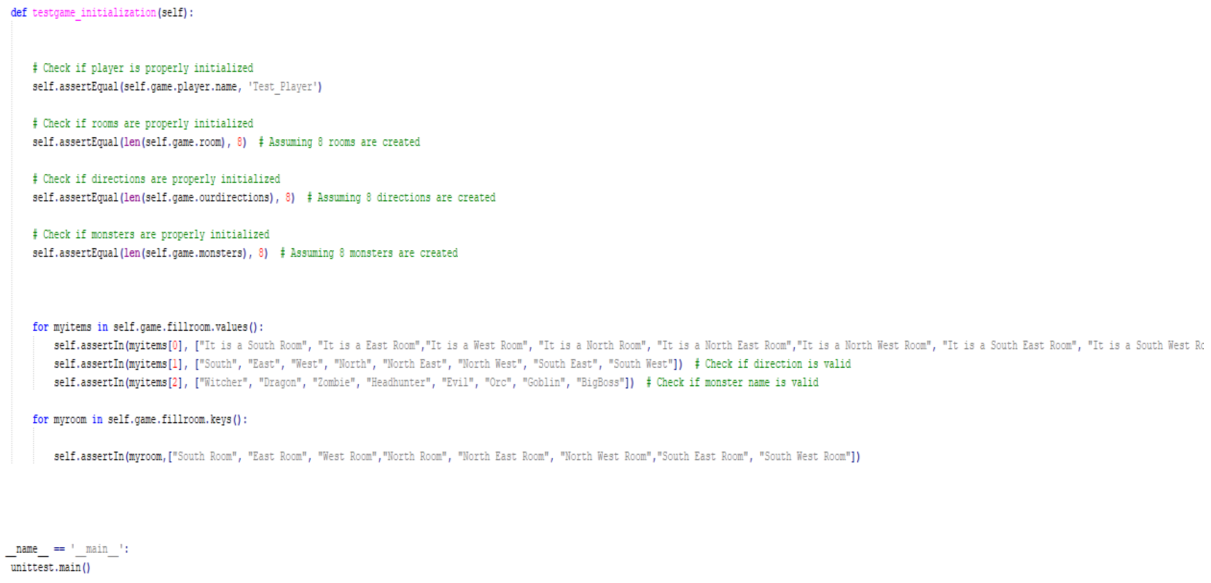
Figure 5.



```python
        self.assertEqual(direction.name, expected_direction_name)
class TestGame(unittest.TestCase):
    def setUp(self):
        self.game = Game()



    def test_fillroom_attribute(self):
        # Check if fillroom attribute is initialized as an empty dictionary
        self.assertIsInstance(self.game.fillroom, dict)
        self.assertEqual(len(self.game.fillroom), 8)
    def test_room_attribute(self):
        # Check if room attribute is initialized as an empty dictionary
        self.assertIsInstance(self.game.room, dict)
        self.assertEqual(len(self.game.room), 8)
    def test_monsters_attribute(self):
        # Check if monsters attribute is initialized as an empty list
        self.assertIsInstance(self.game.monsters, list)
        self.assertEqual(len(self.game.monsters), 8)
    def test_ourdirections_attribute(self):
        self.assertEqual(len(self.game.ourdirections), 8)
    def test_health_adder_attribute(self):
        # Check if health_adder attribute is initialized as an empty list
        self.assertIsInstance(self.game.health_adder, list)
        self.assertEqual(len(self.game.health_adder), 0)
    def test_market_items_sword_attribute(self):
        # Check if market_items_sword attribute is initialized as expected
        expected_market_items_sword = {"Black Sword": 5, "Cursed Sword": 10, "Giant Sword": 15, "Giant Axe": 20}
        self.assertEqual(self.game.market_items_sword, expected_market_items_sword)

    def test_market_items_armor_attribute(self):
        # Check if market_items_armor attribute is initialized as expected
        expected_market_items_armor = {"Black Armor": 5, "Cursed Armor": 10, "Giant Armor I": 15, "Giant Armor II": 20}
        self.assertEqual(self.game.market_items_armor, expected_market_items_armor)

    def test_market_items_armor_health_attribute(self):
        # Check if market_items_armor_health attribute is initialized as expected
        expected_market_items_armor_health = {"Black Armor": 10, "Cursed Armor": 20, "Giant Armor I": 30, "Giant Armor II": 40}
        self.assertEqual(self.game.market_items_armor_health, expected_market_items_armor_health)
```

Figure 6.

Figure 6 checks game function constructor attributes and their data types so on. It is important if the used data is in the correct format or not. Furthermore, their initial values are also checked.



```
def testgame_initialization(self):


    # Check if player is properly initialized
    self.assertEqual(self.game.player.name, 'Test_Player')

    # Check if rooms are properly initialized
    self.assertEqual(len(self.game.room), 8)  # Assuming 8 rooms are created

    # Check if directions are properly initialized
    self.assertEqual(len(self.game.ourdirections), 8)  # Assuming 8 directions are created

    # Check if monsters are properly initialized
    self.assertEqual(len(self.game.monsters), 8)  # Assuming 8 monsters are created



    for myitems in self.game.fillroom.values():
        self.assertIn(myitems[0], ["It is a South Room", "It is a East Room","It is a West Room", "It is a North Room", "It is a North East Room","It is a North West Room", "It is a South East Room", "It is a South West Ro
        self.assertIn(myitems[1], ["South", "East", "West", "North", "North East", "North West", "South East", "South West"])  # Check if direction is valid
        self.assertIn(myitems[2], ["Witcher", "Dragon", "Zombie", "Headhunter", "Evil", "Orc", "Goblin", "BigBoss"])  # Check if monster name is valid

    for myroom in self.game.fillroom.keys():

        self.assertIn(myroom,["South Room", "East Room", "West Room","North Room", "North East Room", "North West Room","South East Room", "South West Room"])



__name__ == '__main__':
 unittest.main()
```

Figure 7.

Also, check list elements' values are keys are created correctly. It is done with dictionary element controls in that part. It is important that when code asks in unit tests to enter a name, you need to enter "Test_Player" because the player input name is defined as Test_Player. Otherwise, tests in-game class is going to be failed.

7.Conclusion

All in all, the game is completed by using an object-oriented approach. At the same time, unit tests have been done in order to control constructor attributes to work correctly, and also checked fillroom dictionary values and keys, as been checked. This dictionary control has vital importance because it fills all room, which is significant for the maintability of the Project. Furthermore, logging points are also created to compute the player's actions.