When I was first introduced to video games, the Nintendo Game Console had just entered the market and with it, an amazing little game called Luigi's Mansion. Since its release I have completed the game no less than a dozen times, and it is also the game that I used as rather heavy inspiration when adapting the World Of Zuul text adventure. The games main protagonist starts in the main lobby of the house and must navigate the building in search of a key that is able to unlock the main entrance located in the lobby. Another mechanic used is locked doors, with the keys to these doors being left in other rooms.

The lobby will act as a tutorial. All the exits out of this room are locked by default, this will force the player to look for ways of unlocking the doors by finding keys within the room. This will also allow players to experiment with moving items to and from their inventory. The game ends when the player finds a master key that opens the buildings front door to safety.

Base Tasks Completed:

- The game has several locations/rooms [completed]
    - This task was the easiest of which to complete. A room within the game is a single instance of the class *Room.* This class contains the name of the room, and three HashMaps that holds the information for the rooms exists, the possible locked doors in the room and the items that may exist within the room. In order to create the room, a constructor was crafted that takes one parameter, the name of the room, and creates a new rom. This constructor then initialises the three Hashmaps that are able to store information. Once I have created the room object, I then create the items separately, as well as the exists and locked doors, and add them separately to the room object that was created.
- The player can walk through the locations.
    - This task was already completed for me in the source code I was given. The code I was given implemented a "go" command that first checked to see if the player first entered a location for the character to move to. Then checks to see if that room has the exit explicitly stated by the user. If this exit does exist, the game calls "getExit" from the class Room and passes to it as a parameter the second part of the command. This method returns an object of type Room that is then stored in the field "currentRoom". To finalise the transition, the method "getLongDescription" from the Room class which returns a string. Then System.out.println is used to display the text to the console.
- The player can carry some items with him. Every item has a weight. The player can carry items only up to a certain total weight.
    - This was the first task I had to create from scratch. In order to complete this task I. had to create two separate classes, the first Item and the second Inventory. The class Item is used to store all the information about any items that may appear within the videogame. This includes fields such as name, type and weight. The class Inventory is used to represent the inventory of not only the player. But by other interactable objects within the game too. The Inventory class contains a Hashmap field that is used to keep track of all the items within the inventory. Other fields used in the inventory are "maximumWeight" and "currentInventoryWeight". In order to create the players inventory, I created a constructor within the Inventory class that takes

one parameter, the maximum weight of that inventory. The constructor also initialises the Hashmap, sets the maximum weight of the inventory and sets how many items are currently in the inventory to zero. After I created the constructor, I also created three methods. The first is called "addItemToInventory". This method takes one parameter, of type Item. It then checks to see if the items weight combined with the total weight inside the inventory is greter that the maximum weight. If it is, than the item cannot be added to the inventory. If this returns false, than the item is added into the HashMap and the items weight is added to the current weight of the inventory. The next method is called "displayInventory". This method has no return types or parameters and simply prints each entry of the HashMap to the console. The final method is called "findInventoryItem". This method is used to check if an item exists within the inventory. If it does, it returns true, if not , it returns false. The item class is used to define the item including its type, its weight, and whether or not the item can be held by the player. Each Item also has an internal inventory. This is used to store items that would be inside cabinets and other such decoration. In order for the player to move items between the objects inventory and the player inventory, I implemented command that take two parameters excluding the command word. The command is "take [itemName] [locationName]". The method used to implement this first checks to see if the item name is there, then the location name and once both parts have been found, the game then checks the room inventory to see if the object is there, if it is it then checks the object inventory to see if the item is in that location. If it is, it calls the method addItemToInventory which adds the items to the players inventory. Then the item is removed from the objects inventory.

- The player can win:
    - The end object of the game was to find a final key that would unlock the door to escaping the house. Once this key was found the player had to navigate back to the lobby of the building where the game would check to see if the player had collected the key. If the player had, then the game prints a notification to the screen telling the player they had won.
- Back Command:
    - The back command was relatively simple to implement. I added another field into the Game class called "previousRoom". Before the "currentRoom" variable is changed when the player changes room, it is stored in the variable "previousRoom". When the back command is used, a method is. Used to set the currentRoom to previousRoom and prints out its name.
- Four New Commands:
    - The four new commands I added were inventory, used to list all the items the player is currently holding, inspect, which is used to view the items stored inside an object within the room, back, which takes the player to the previous room, take, which takes an item from an objects inventory and puts it into the players and describe, which is used to list all the items within the current room.

Challenge Tasks Complete:

- The first challenge task I finished was the implementation of a three token command. This was relatively simple to implement, I added another conditional loop to check if the string contains a third token, if so, it separates it into a third word. I also added to the constructor of the command to take a third parameter to add the third word in the command.
- The next challenge I added was the use of locked doors. This would limits the players exploration, which allows a sense of progression through the game. I did this by adding a new HashMap to the Room class. This Hashmap stores the possible exists as Strings and an Integer value which represents either true or false. A one for true and a zero for false. Whilst in the "go" command, the game references this HashMap with the direction used in the "go" command. If the value returns a one then the door is locked and if a zero. Is returned the door I unlocked. If the door is locked, the game will check to see if the key is available within the players inventory.

Brief Game Walkthrough:

When starting a game, the first room you'll be placed into is the Lobby. This rooms starts off with all doors locked and two objects in the room. This was intentionally done to teach the player that they can inspect objects, takes items from the objects and use keys to unlock doors. The first command that players would use is the go command, this would let the player know the doors are locked. This would then make the player use the help command to see all the possible commands. The command describe should then be used to list all the items within the room so they can use the inspect command on them to find if. They have any. Items within the objects. If they do, the player can then use the take command to take the item from the object and put it into their inventory. Keys are used to unlock doors. The name of the key is designed such that "eastkeylobby" represents that this. Key unlocks the east side door in the lobby room. The main purpose of this game is to get the "mainkeylobby" which resides in the house somewhere.

Known Game Bugs:

Go Command:

When using the go command in the game, the game checks to see if that door is locked before the game checks if the game exists. As such, if exit does exist in the locked doors HashMap, then the game returns a null value exception.

Inventory:

When removing items from the inventory, the items have to be removed in the opposite order in which they were added. If they are not taken out in this order, than the inventory may encounter a null reference exception when trying to find an item in the Hashmap as the key may not exist.

Code Quality Considerations:

Coupling: Coupling refers to how tightly interwoven a class is with other classes within the project. This coupling is described as either tight coupling or loose coupling. Ideally, we want to have as loose a coupling as possible, thus reducing how much one class depends on the other. At the time of writing this, there is a very tight coupling between the main Game class with other classes, especially the command and parser classes. A lot of the command interpretation is handled by the main Game classes, but the command needs to be parsed which is done inside the Parser. In order to weaken this coupling, I want to take the command interpretation out of the main Game class and into the Parser.

Cohesion: This refers to how many individual logical tasks this class undertakes. High cohesion means only one class is used to undertake one logical task whilst low cohesion means this class undertake multiple different logical task. The goal is to increase the cohesion of the classes so each one is undertaking one logical task. Once again, I refer to the game class. The game class has a very low cohesion. It undertakes several tasks from command interpretation to locating items inside different objects. The way to increase the cohesion of a class is by taking each logical task and isolating it into its own separate class. An example of this is done is the inventory class. This class is responsible solely for the creation and modification of the players inventory. If another class needs to access the inventory, it has to be done through the inventory class through methods.

Responsibility Driven Design (RDD): This design aspect concerns itself with how the classes manipulate information. RDD means each class has access to its own set of variables and should be the only class that has the ability to modify these values. An example of this in my code is once again the inventory class. This class only manipulates the information that it has stored within the class. This manipulation is done through the use of methods built into the class with a public access modifier. Not only does this increase cohesion, it also increases maintainability and expandability. It allows for one class to be modified internally without other classes being affect by the change. The inventory class can modify the internal workings of the variables and even change and rename them and won't affect other classes as long as the

Maintainability: Maintainability not only measures how hard it is for a piece of code to be kept up to date, it also measure how much the code can be modified without afflicting the other classes in the project. The easier it is to add another feature to a class, the more maintainable it is. This can be seen in the Item class. This class has only one purpose, the controlling and manipulation of item data. All the fields in this class are private and as such, they cannot be directly accessed by another class. This is. The first step in approving maintainability. These variables can be freely changed internally without affecting the other classes. The next aspect if the actual modification of this data. All of the modification is handled by the class through the use of publicly accessible methods. This means other classes can call the methods and get information, or modify the information within the class. This adds a layer of abstraction to the implementation of the code allowing it to be changed without affecting other classes as long as the signature of the method stays the same.