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.

```
1  import java.util.HashMap;
2  /**
3   *  This class is the main class of the "World of Zuul" application.
4   *  "World of Zuul" is a very simple, text based adventure game.  Users
5   *  can walk around some scenery. That's all. It should really be extended
6   *  to make it more interesting!
7   *
8   *  To play this game, create an instance of this class and call the "play"
9   *  method.
10  *
11  *  This main class creates and initialises all the others: it creates all
12  *  rooms, creates the parser and starts the game.  It also evaluates and
13  *  executes the commands that the parser returns.
14  *
15  * @author  Michael Kölling and David J. Barnes
16  * @version 2016.02.29
17  *
18  * modified by Charlie Madigan (K19019003)
19  */
20
21 public class Game
22 {
23     private Parser parser;
24     private Room currentRoom, previousRoom;
25     private Inventory playerInventory;
26     /**
27      * Create the game and initialise its internal map.
28      */
29     public Game()
30     {
31         parser =  new Parser();
32         playerInventory = new Inventory (20);
33         createRooms();
34     }
35
36     /**
37      * Create all the rooms and link their exits together.
38      * This method also generates the items that  can be found inside that room and
    associates them  to the room.
39      */
40     private void createRooms()
41     {
42         Room lobby, secondFloorHallway, library, masterBedroom,
    masterBedroomBathroom, loft, firstFloorHallway, livingRoom,
43         porch, office, officeBathroom, diningRoom, basementHallway, kitchen;
44
45         // create the rooms
46         lobby = new Room("Lobby");
47         secondFloorHallway = new Room("Second Floor Hallway");
48         library = new Room("Library");
```

```
        masterBedroom = new Room("Master Bedroom");
        masterBedroomBathroom = new Room("Master Bedroom Bathroom");
        loft = new Room("Loft");
        firstFloorHallway = new Room("First Floor Hallway");
        livingRoom = new Room ("Living Room");
        porch = new Room("Proch");
        office = new Room("Office");
        officeBathroom = new Room("Office Bathroom");
        diningRoom = new Room ("Dining Room");
        basementHallway =  new Room("Basement Hallway");
        kitchen = new Room ("Kitchen");

        lobby.setExit("east", diningRoom);
        lobby.setLockedDoors("east", 1);
        lobby.setExit("west", livingRoom);
        lobby.setLockedDoors("west", 1);
        lobby.setExit("upstairs", secondFloorHallway);
        lobby.setLockedDoors("upstairs", 1);
        Item lobbyCabinate =  new Item ("cabinate", "Decoartion", false, true, 30);
        Item lobbyCabinateGoldCoins = new Item("coin", "Score", true, false, 0);
        Item lobbyEastDoorKey  = new Item("eastkeylobby", "Key", true, false, 5);
        lobbyCabinate.addItemToInventory(lobbyCabinateGoldCoins);
        lobbyCabinate.addItemToInventory(lobbyEastDoorKey);
        Item lobbySmallTable = new Item("smalltable", "Decoration", false, true,
20);
        Item lobbyGoldBar =  new Item("goldbard", "Score", true, false, 0);
        Item lobbyFood = new Item("food", "nurishment", true, false, 2);
        lobbySmallTable.addItemToInventory(lobbyGoldBar);
        lobbySmallTable.addItemToInventory(lobbyCabinateGoldCoins);
        lobbySmallTable.addItemToInventory(lobbyFood);
        lobby.addItemToRoom(lobbyCabinate);

        livingRoom.setExit("north", porch);
        livingRoom.setLockedDoors("north", 0);
        livingRoom.setExit("east", lobby);
        livingRoom.setLockedDoors("east", 0);
        Item livingRoomTable = new Item("table", "Decoration", false, true, 50);
        Item mainExitKey = new Item ("mainexitkey", "key", true, false, 0);
        livingRoomTable.addItemToInventory(mainExitKey);
        livingRoom.addItemToRoom(livingRoomTable);

        porch.setExit("east", office);
        porch.setLockedDoors("east", 0);
        porch.setExit("south", livingRoom);
        porch.setLockedDoors("south", 0);

        office.setExit("south", officeBathroom);
        office.setLockedDoors("south", 0);
        office.setExit("west", porch);
        office.setLockedDoors("south", 0);
```

```
98
99          officeBathroom.setExit("north", office);
100         officeBathroom.setLockedDoors("north", 0);
101
102         diningRoom.setExit("north", kitchen);
103         diningRoom.setLockedDoors("north", 0);
104         diningRoom.setExit("west", lobby);
105         diningRoom.setLockedDoors("west", 0);
106         Item diningRoomTable = new Item("table", "Decoration", false, true, 30);
107         Item diningRoomPlate = new Item("plate", "Decoration", true, false, 4);
108         Item diningRoomKnife = new Item("knife", "Decoration", true, false, 4);
109         Item diningRoomFork = new Item("fork", "Decoration", true, false, 4);
110         Item diningRoomGlass = new Item("glass", "Decoration", true, false, 4);
111         Item diningRoomCoin = new Item("coin", "Score", true, false, 0);
112         Item diningRoomGoldBar = new Item("goldbar", "Score", true, false, 0);
113         Item diningRoomFood = new Item("food", "Nurishment", true, false, 2);
114         diningRoomTable.addItemToInventory(diningRoomPlate);
115         diningRoomTable.addItemToInventory(diningRoomKnife);
116         diningRoomTable.addItemToInventory(diningRoomFork);
117         diningRoomTable.addItemToInventory(diningRoomGlass);
118         diningRoomTable.addItemToInventory(diningRoomCoin);
119         diningRoomTable.addItemToInventory(diningRoomGoldBar);
120         diningRoomTable.addItemToInventory(diningRoomFood);
121         diningRoom.addItemToRoom(diningRoomTable);
122
123         kitchen.setExit("south", diningRoom);
124         kitchen.setLockedDoors("south", 0);
125         Item kitchenOven = new Item("oven", "Decoration", false , true, 50);
126         Item kitchenSink = new Item("sink", "Decoration", false , true, 50);
127         Item kitchenWashingMachine = new Item("washingmachine", "Decoration", false
    , true, 50);
128         Item kitchenTable = new Item("table", "Decoration", false , true, 50);
129         Item kitchenPlate = new Item("plate", "Decoration", true, false, 4);
130         Item kitchenKnife = new Item("knife", "Decoration", true, false, 4);
131         Item kitchenFork = new Item("fork", "Decoration", true, false, 4);
132         Item kitchenGlass = new Item("glass", "Decoration", true, false, 4);
133         Item kitchenCoin = new Item("coin", "Score", true, false, 0);
134         Item kitchenGoldBar = new Item("goldbar", "Score", true, false, 0);
135         Item kitchenFood = new Item("food", "Nurishment", true, false, 2);
136         Item livingRoomKey = new Item("westkeylobby", "key", true, false, 0);
137         Item kitchenPot = new Item ("pot","Decoration", true, false, 0);
138         kitchenTable.addItemToInventory(diningRoomPlate);
139         kitchenTable.addItemToInventory(diningRoomKnife);
140         kitchenTable.addItemToInventory(diningRoomFork);
141         kitchenTable.addItemToInventory(diningRoomGlass);
142         kitchenTable.addItemToInventory(diningRoomCoin);
143         kitchenTable.addItemToInventory(diningRoomGoldBar);
144         kitchenTable.addItemToInventory(diningRoomFood);
145         kitchenTable.addItemToInventory(livingRoomKey);
146         kitchenWashingMachine.addItemToInventory(diningRoomCoin);
```

```
147         kitchenWashingMachine.addItemToInventory(diningRoomGoldBar);
148         kitchenWashingMachine.addItemToInventory(livingRoomKey);
149         kitchenSink.addItemToInventory(diningRoomPlate);
150         kitchenSink.addItemToInventory(diningRoomKnife);
151         kitchenSink.addItemToInventory(diningRoomGlass);
152         kitchenSink.addItemToInventory(diningRoomFork);
153         kitchen.addItemToRoom(kitchenOven);
154         kitchen.addItemToRoom(kitchenTable);
155         kitchen.addItemToRoom(kitchenWashingMachine);
156         kitchen.addItemToRoom(kitchenSink);
157 
158         previousRoom = currentRoom = lobby;  // Sets the starting room of the  game
159     }
160 
161     /**
162      *  Main play routine.  Loops until end of play.
163      */
164     public void play()
165     {
166         printWelcome();
167 
168         // Enter the main command loop.  Here we repeatedly read commands and
169         // execute them until the game is over.
170 
171         boolean finished = false;
172         while (! finished) {
173             Command command = parser.getCommand();
174             finished = processCommand(command);
175         }
176         System.out.println("Thank you for playing.  Good bye.");
177     }
178 
179     /**
180      * Print out the opening message for the player.
181      */
182     private void printWelcome()
183     {
184         System.out.println();
185         System.out.println("Welcome to the World of Zuul!");
186         System.out.println("World of Zuul is a new, incredibly boring adventure
    game.");
187         System.out.println("Type 'help' if you need help.");
188         System.out.println();
189         System.out.println(currentRoom.getName());
190         System.out.println(currentRoom.getExitString());
191     }
192 
193     /**
194      * Given a command, process (that is: execute) the command.
195      * @param command The command to be processed.
```

```
196      * @return true If the command ends the game, false otherwise.
197      */
198     private boolean processCommand(Command command)
199     {
200         boolean wantToQuit = false;
201 
202         if(command.isUnknown()) {
203             System.out.println("I don't know what you mean...");
204             return false;
205         }
206 
207         String commandWord = command.getCommandWord();
208         if (commandWord.equals("help")) {
209             printHelp();
210         }
211         else if (commandWord.equals("go")) {
212             goRoom(command);
213         }
214         else if (commandWord.equals("quit")) {
215             wantToQuit = quit(command);
216         } else if (commandWord.equals("back")) {
217             backCMD();
218         } else if (commandWord.equals("inventory")) {
219             showInventory();
220         } else  if (commandWord.equals("take")) {
221             takeItem(command);
222         } else if (commandWord.equals("inspect")) {
223             inspect(command);
224         } else if (commandWord.equals("describe")) {
225             describe();
226         }
227         // else command not recognised.
228         return wantToQuit;
229     }
230 
231     // implementations of user commands:
232 
233     /**
234      * Print out some help information.
235      * Here we print some stupid, cryptic message and a list of the
236      * command words.
237      */
238     private void printHelp()
239     {
240         System.out.println("You are lost. You are alone. You wander");
241         System.out.println("the house in a panic.");
242         System.out.println();
243         System.out.println("Your command words are:");
244         parser.showCommands();
245     }
```

```
246
247     /**
248      * Try to in to one direction. If there is an exit, enter the new
249      * room, otherwise print an error message.
250      */
251     private void goRoom(Command command)
252     {
253         if(!command.hasSecondWord()) {
254             // if there is no second word, we don't know where to go...
255             System.out.println("Go where?");
256             return;
257         }
258         if (currentRoom.isDoorLocked(command.getSecondWord()) == true) {
259             System.out.println("This door is locked, maybe theres a key to open it
somewhere around here?");
260             String itemName = command.getSecondWord() + "key" +
currentRoom.getName().toLowerCase();
261             if (playerInventory.findInventoryItem(itemName)) {
262                 System.out.println("Oh, you;ve found the key! You can now enter this
room.");
263                 currentRoom.unlockDoor(command.getSecondWord());
264             } else {
265                 return;
266             }
267         }
268         String direction = command.getSecondWord();
269
270         // Try to leave current room.
271         Room nextRoom = currentRoom.getExit(direction);
272
273         if (nextRoom == null) {
274             System.out.println("There is no door!");
275         }
276         else {
277             previousRoom = currentRoom;
278             currentRoom = nextRoom;
279             System.out.println(currentRoom.getName());
280             System.out.println(currentRoom.getExitString());
281             if (currentRoom.getName().equals("Lobby")) {
282                 checkEndCondition();
283             }
284         }
285     }
286
287     /**
288      * This method checks to see if the  end  condition for the game has been met
289      * if it has, then the game will move to the game over text.
290      */
291
292     private void checkEndCondition () {
```

```
293             if (playerInventory.findInventoryItem("mainexitkey")) {
294                 endGame();
295             }
296         }
297 
298         /**
299          * this method tells the player they have finished the game but instead of
    quitting, the player has the option to continue exploring.
300          */
301 
302         private void  endGame () {
303             System.out.println("Congratulations!\nYou found the key and was able to
    escape from the house!\nThank you for playing!");
304             System.out.println("You can continue to explore if you like but when your
    finish type 'quit' to leave.");
305         }
306 
307         /**
308          * This command allows the player to go backwards to the previous room
309          */
310 
311         private void backCMD () {
312             currentRoom = previousRoom;
313             System.out.println(currentRoom.getLongDescription());
314         }
315 
316         /**
317          * this method calls a method from the player  inventory which outputs the
    current player inventory to the screen.
318          */
319 
320         private void showInventory () {
321             playerInventory.displayInventory();
322         }
323 
324         /**
325          * This method takes an item from one inventory and places it into  another
326          * It first  checks to see if the command has the item [second token], if not
    ask the player what item they are looking for
327          * Then it checks to see if the command has the location of the item [third
    token] if not ask the player where they think the item is
328          * Then it checks to see if the current room has the location of the item, if
    the location  isnt in this room, tell the player
329          * Then it checks if the item exists in that object. if the item doesnt exist in
    that location tell the player the item isnt there
330          * then it moves the object from one inventory to the other.
331          */
332         private void  takeItem (Command command) {
333             if (!command.hasSecondWord()) {                         //Check if we
    have the second word in the command
```

```
334             System.out.println("What item do you want to take?");
335         } else {
336             if (command.hasThirdWord() == false) {                  //Check if we
have the third word in the command
337                 System.out.println("Where do you want to take the item from?");
338             } else {
339                 if (currentRoom.getItemList().isEmpty()) {
340                     System.out.println(command.getThirdWord() + " doesn't exist
within this room.");
341                 } else {
342                     for (int i = 0; i < currentRoom.getItemList().size();
i++){             //This block checks to see if the item is in the room
343                         Item tmp = currentRoom.getInventoryItem(i);
344                         if (tmp.getItemName().equals(command.getThirdWord()))
{        //If the item is in this room, get the item inventory to find the item we
want
345                             Item itemMoved =
tmp.removeItem(command.getSecondWord());
346                             if (itemMoved == null) {
347                                 System.out.println("Sorry, I couldn't find that item
in there");
348                             } else {
349                                 playerInventory.addItemToInventory(itemMoved);
350                             }
351                             return;
352                         }
353                     }
354                     System.out.println("Sorry I couldnt find that item.");
355                 }
356             }
357         }
358     }
359
360     /**
361      * This method allows the player to see the items in an objects inventory
362      * It first checks to see if the location is specified [second token], if not
ask the player what they would like to  inspect
363      * Then it prints the full list of items out.
364      */
365
366     private void inspect(Command command) {
367         if (!command.hasSecondWord()) {
368             System.out.println("What would you like to inspect?");
369         } else {
370             for(int i = 0; i < currentRoom.getItemList().size(); i++){
371                 Item tmp = currentRoom.getInventoryItem(i);
372                 if (tmp.getItemName().equals(command.getSecondWord())) {
373                     tmp.displayInventory();
374                 }
375             }
```

Gam