PPA-Assignment

ARTEMIS

game report

Date 24-11-2017

# Yeshvanth Prabakar K1763461



# Game name: Artemis

-Note for reader. Lines of code taken from assignment and user given input, output to illustrate certain points are written between single quotes ‘’;

-Classes, Variables and Keywords from code are written in *italics*.

# Description of game:

The game revolves around the city of Artemis an underwater Utopia that was designed as the future for mankind. However, a pandemic spread killing the inhabitants and now it is an abandoned city. The game starts with the player finding himself in the midst of the sea right after his plane crashes. He must enter the building like structure that is Artemis and find his way home. The user must find the key to open the submarine door and 3 fusecomponents to turn on the power to win the game.

# Walkthrough of game (winning commands):

‘go north’, ‘select Key’,’ go up’, ‘go west’, ‘go south’, ‘select SecondFuseComponent’, ‘back’, ‘go east’, ‘go south’, ‘select ThirdFuseComponent’, ‘go north’, ‘go down’, ‘go down’, ‘select FirstFuseComponent’, ‘go north’.

# Base tasks:

* **Game has several rooms/player can walk through these:**

This was already implemented in the basic version of the zuul game given to us. Additional new rooms were created using the *createRooms()* method.

* **How the player knows they won:**

To enter the submarine (room) the player must find the key and have it in his inventory. An if statement with the use of the *hasKey()* method from the *Character* class checks if the key is present and whether the next room is submarine. An if statement was implemented in the *goRoom()* method (of Game class) to check for this condition, if the user has not collected the Key he is prompted to do so. The user can enter the submarine with the key but must collect all 3 fusecomponents to turn on the power of the submarine and win the game. So once he enters the submarine the game prompts the user to pick up all fuse components.

The *hasEscaped()* method in the *Character* class checks whether the player has met all conditions of the game (player has key and all 3 fusecomponents in his inventory) and returns true otherwise returns false. An if statement in the *win()* method (Game class) checks if the player is in the submarine with ‘currentRoom==submarine’ and whether *hasEscaped()* returns true; then only will the *win()* method return a true value. The *win()* method was then called in the *processCommand()* method of Game class; then using an if statement to check whether the winning condition is true, if true the command was given to quit the game: ‘wantToQuit = quit(command);’.

The game also ends automatically (program stops executing) when the user enters quit (as the commandWord) or if the player is killed by the monster. The *monsterAppearance()* returns true when the condition is satisfied (monster spawns and kills player) which was checked using if statement in *processCommand()* method. In either case the command to quit the game was given using ‘wantToQuit = quit(command);’.

* **Items in the room:**

Items that the player encounters in the room were stored as an *items* HashMap in the class Room. These objects (items) have the name of the item as the key (String) and the weight of the item (Integer) in arbitrary units (as the value). The setItem() method in game class was used to add items to the HashMap.

* **Items carried by player, some items cannot be carried and weight limit:**

Items carried by the player in their inventory were added to the itemsCarried LinkedList. A LinkedList was used instead of an ArrayList as items have to be frequently picked up and dropped by the player, in other words elements have to be added to and removed through the course of the game, thus a LinkedList was used as it is more appropriate. One may argue however that the difference in performance is negligible since this not a huge application.

The maximum weight that can be carried by the player is 150 units and to simulate this multiple nested if statements were used in class Game *selectItem*() method. The first if statement checks whether the item entered by the user is a valid item. If it is a valid item (that is, the item is an item in the room which is returned by the *getItems()* method). To stimulate the idea that some items can’t be picked up by the user, certain items were given an *Integer* value greater than 150units. That is what the if statement checks for; ‘if((currentRoom.returnWeight(secondWord))>150)’. If the item is a valid item (its weight is less than 150units). The *character.carryItem(String item)* method is called to store the object in the private *itemsCarried* LinkedList. The else if statement that follows the if statement checks whether your inventory exceeds 150units or if it would exceeds 150units were you to carry that item as that is the maximum weight the player can carry. If both these conditions aren’t satisfied only then is the item stored by entering the else statement. If the item is not a valid item (weight of item >150units) or the players inventory is full (weight of inventory+weight of item>150 units) the appropriate error-response is outputted to the user.

If the player picks up the item from the room, the item still remains in the room when he returns to the same room later. This concept of ‘unlimited items’ was opted for as it fits in better with this fictional storyline.

* **Back command:**

To implement the *back*, *back* was first added as a keyword in the string array *validCommands* in *CommandWords* class. Then a class *directionBack()* was created which sets the object variable *currentRoom* to *previousRoom*. When the player enters ‘go direction” currentRoom is set to nextRoom but before that line of code is executed, the previousRoom stores the currentRoom: ‘previousRoom = currentRoom;’ This is done so that when the *‘currentRoom previousRoom’* is executed in the *directionBack()* method, the player will go back to the room he was previously in. The *fountainOfHeritageRoom* teleports the player to a random room. But if the user enters ‘back’ right after, they remain in the same room since the previousRoom variable now points to the same room as currentRoom that the player has been teleported to so when the ‘back’ commandWord is entered: ‘currentRoom = previousRoom;’ is executed and the player stays in the same room. So a limitation of the ‘back’ command is that it only works once, after the player enters ‘go direction’.

|  |  |
| --- | --- |
| **Commandword** | **Effect** |
| select | Picks up the item in the room and adds it to the inventory |
| drop | Drops the item that was picked up |
| give | Give the mysteriousMan an ancestralCoin in exchange for a cloak |
| back | Takes player to last room |
| use | Use an item, only the cigarette can be ‘used’ |

* **Four new commands:**

A *drop* keyword was added to drop the item that the player is carrying in their inventory. An if statement was added to the *processCommand()* method in the Game class to check whether the commandWord was ‘drop’. If that condition is satisfied and the item is a valid item that is being carried by the player, the respective methods are called to remove the item from the inventory and subtract the weight.

To prevent the compiler from throwing a null pointer exception if an item is dropped in a different room than the one it was picked up in: an if and if-else statement was implemented inside the *dropItem()*method under the block of code where the ‘drop’ commandWord was implemented. The compiler threw null pointer exceptions when the secondWord or thirdWord of a user given command was null, in other words one more blanks (\_) was entered by the user after the commandWord (‘select\_‘ or ‘give \_ \_’), give is a 3command word:

‘java.lang.NullPointerException

at java.lang.String.contains(String.java:2133)

at Game.processCommand(Game.java:176)

at Game.play(Game.java:123)’

This was accounted for by using branched if, else-if statements in the *getCloakOfShadows()* and an if statement at the top of the *selectItem()* methods of *Game* class to check whether the words inputted by the user were null. There is also not an issue when the user enters a wrong word or misspells it as this was accounted for by using *if* statements in the “Game” class and *getCloakOfShadows()*, *selectItem()* methods that prints the appropriate negative responser to the user (refer code).

# . Challenge tasks:

* **Magical Transporter:**

The player is teleported when he enters a room; the *fountainOfHeritage*. To carry this out an ArrayList *randomRoom* was made which stores the room objects. The *addRooms()* method adds each room to the ArrayList and the random rooms are generated by using the *Random* class to generate an int, which is passed as a parameter to the ArrayList *getmethod():* ‘nextRoom=randomRoom.get(randomNumber);’. The room at a random index in the ArrayList is accessed and using an if statement in the *goRoom()* method the currentRoom is set to a random Room: ‘currentRoom=nextRoom;’

* **Characters (Monster and MysteriousMan);**

A monster class was created to add monsters to the game which can kill a player. A random number between 0-5 (inclusive end points) was generated using the random class in both the *probabilityKilled() and*  *spawn()* methods (of class Monster). These methods were then called in the *monsterAppearance()* method of Game class. The monster can spawn and or kill the player in every new room that the character enters. If the number equals 1, then both method returns true and the game ends. However if the monster is just spawned (*spawn()* methods returns true) and *probabilityKilled()* returs false then the player escapes the monster. So there is a probability that the monster appears and a further probability the player is killed, which yields a chance that the player dies every time he enters a new room. This was done to mimic the effect of a monster chasing the player throughout the whole game as the player tries to escape. The mysteriousMan is a stationary character who appears in the *mainHall*. He is created using the *createCharacter()* method in *Game* Class.

* **Extending the parser (3 worded commands):**

The parser was extended to recognize three words by modifying the Parser class. This was done by creating a new String word3 and letting ‘word3 = tokenizer.next();’. Then using an if and else if conditions in the *getCloakShadows()* method (of game Class) check whether the player has the ancestralCoin in his inventory which is generated in a random room (refer bottom of *createItems()* method):

If he does and the player can use the three worded command ‘give ancestralCoin mysteriousMan’ to exchange the coin (coin is removed from inventory) for the *cloakOfShadows* item(cloak is added to inventory). This feature allows the player to disguise; reducing the chance that he is killed by the monster. This was implemented in the *getCloakOfShadows()* method (Game class) which checks if the cloakOfShadows item is in the players inventory: ‘if(itemsCarried.contains("cloakOfShadows"))’. If yes, it passes 10 as a parameter to the p*robabilitykilled(10)* method. This ensures that the randomNum now can have possible values between 0-9 (randomNum must be equal to 1 in order for the player to be killed) hence reducing the probability that *probabilityKilled()* method in class *Monster* returns true in other words reducing the chance the player is killed.

# Code Quality:

Careful attention was given to the design of the program in terms of coherence and coupling. A glance of the class-diagram in BlueJ which has only one arrow (of one direction) between any two classes gives us a rough idea that best efforts were put through to limit coupling between classes. Proper encapsulation was carried out to reduce coupling between classes

by keeping all instance variables private. Public Getter(accessor) and Setter(mutator) methods were used to access these private variables and collections from other classes. The *setItem()* and *getItem()* methods in the Room class illustrate this well which were used to add and display the items stored in the items ArrayList.

To carry out coherence each class was attempted to represent one entity while each method performed only one single task. For example, all concepts related to the room and its design were incorporated in the *Room* class. This includes the items (objects in the program) that were added to the room; they were stored in a *items* HashMap. Additionally, each method was responsible for one task only; a *getItems()* method for example was only responsible for returning the item in each room.

More attention could have been paid to responsibility driven design in this program. The base game came with the objects of each room storing the exits of the room, making the room responsible for providing this information. The same was done for the items, where each room was responsible for storing its own items using the *setItem()* method.

Maintenance of code was taken into account when the program was constructed. Proper encapsulation and coherence makes sure that the code does not break should modifications be made. Every time an item is set in a room using a *setItem()* method, the appropriate method *getItems()* automatically calls and displays the newly added item: so if a new room were to be added this would be done automatically. Another example would be the teleporter which transports the player to a random room. Once the new room is created, all the programmer has to do is add the room to the *randomRoom* ArrayList and the player can be teleported to this new room because the *size()* method (which automatically resizes depending on the number of elements) is passed as a parameter when the get method wants to retrieve the value at the specific index.

# Minor add-ons:

Using the random class to generate a random number and an ArrayList to store all the rooms, the *setItem()* method was called to add items to the different rooms randomly.   
The default response when a user tried to enter in a direction where there was no exit was “ There is no door”. However an ArrayList named *doorOptions* (in Game class) was used to store a list of responses and the Random class was used to generate these responses randomly to the user.