

# **World Navigation**

**Motasem Aldiab**

**Atypon**

Mohamad Saad Eddin

## Contents

Summary: .....	1
Introduction: .....	2
World Navigation- The Map: .....	3
The hard level (hard map): .....	3
The easy level (easy map): .....	10
The commands of the game:.....	11
Clean Code Part: .....	20
Avoiding code smell: .....	21
Comments: .....	21
Functions: .....	21
GENERAL: .....	23
NAMES:.....	29
Conclusion: .....	30
Design Patterns: .....	31
Used Data Structures:.....	34
Conclusion:.....	35
References: .....	36

## Summary:

World Navigation is a console game where the user issues commands to navigate and act on a map.

The map is made of a graph of Rooms.

Each room can be thought of as a logical square and has four walls, a wall can have a painting, a chest, a mirror, a door, a seller or is just a plain wall.

The player starts at the <Start> room and is facing one of the four walls and if the map specifies, the player can start with an initial amount of gold.

I need to develop this game using JAVA language.

And apply some important principles and technics with the code to make it clean and readable as much as possible.

I will try to make my code satisfies the clean code principles according to **Robert Martin's "Clean Code" book**, And the effective Java code principles according to **Joshua Bloch's "Effective Java" book**.

# Introduction:

There are two things- Programming and Good Programming.

Programming is what we have all been doing. Now is the time to do good programming. We all know that even the bad code works. But it takes time and resources to make a program good. Moreover, other developers mock you when they are trying to find what all is happening in your code. But it's never too late to care for your programs.

To make your code clean and readable, it's not easy, but it deserves the suffering.

Always try to let your code as you want to find others code.

Make your code is readable is not enough. Also, you need to make it more effective as much as you can.

effective code means the code can live a long time with the ability to change and extend this code in a good and smooth way.

I will discuss with you the project and the code and will mention the principles and the data structures which I have used within the project.

## World Navigation- The Map:

let's start discussing the maps of the game.

For now there are tow maps ( 2 levels ), the first one is the hard map (9 rooms),

and the other one is an easy map (4 rooms)

The default one is the hard map, So we will display it now and describe how it works and the initial state of it.

### The hard level (hard map):

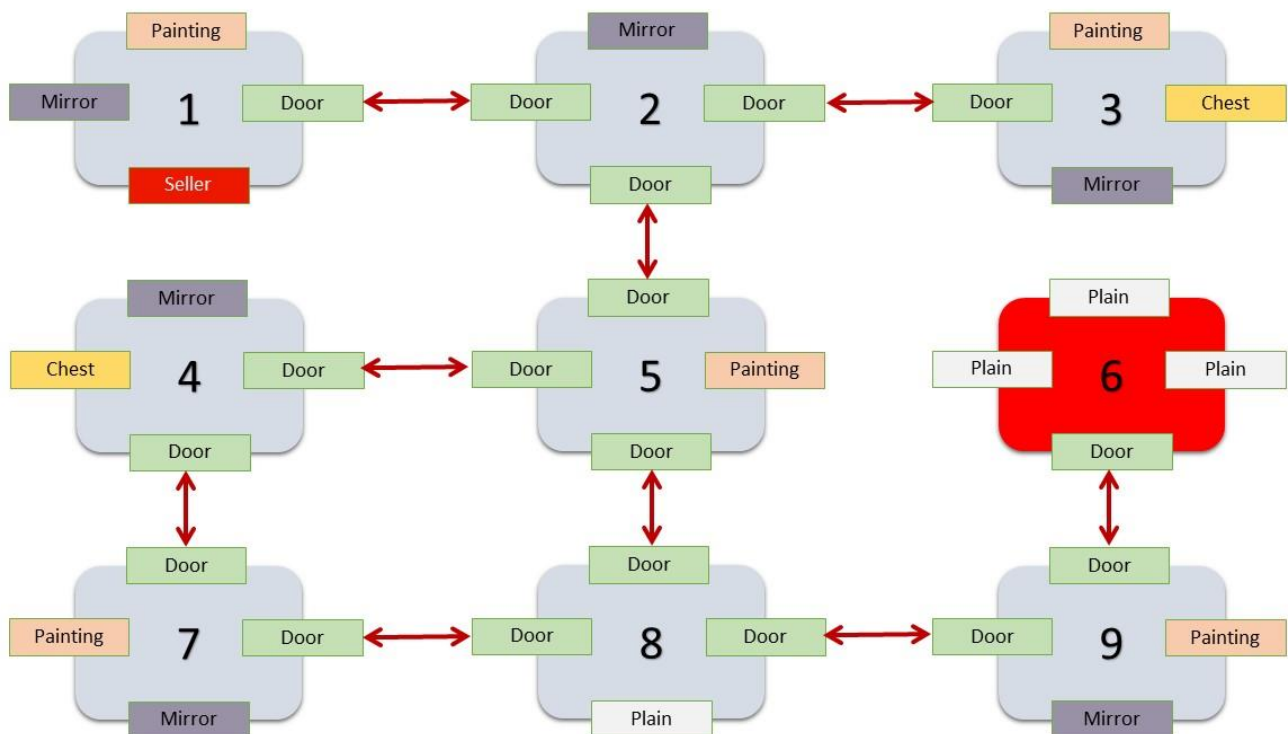
this map consists of 9 rooms some of them are adjacent to some others, and each room has 4 walls, each wall may has an item on it like (Painting, Mirror, Door, Chest, Seller, Or Plain (empty wall)).

The player will start from the room number 1.

The target is the room number 6.

when the player reaches the target room, s/he will win the game.

The next picture is a virtual map to imagine it as a visual maze.



I generated this Map using JSON file, by listing all the necessary data inside the file as json format.

then, when we start the game: the program will read the data from the file and create the map as we will see later.

the JSON file for this map contains this information:

```
1  {
2  >   "northWalls": [...
46  ],
47
48   "southWalls": [
49     {
50       "type": "seller",
51       "items": [
52         {"itemName": "chest3", "price": 10},
53         {"itemName": "flashlight", "price": 5}
54       ]
55     },
56     {
57       "type": "door",
58       "keyName": "door2.5",
59       "isOpen": false
60     },
61     {
62       "type": "mirror",
63       "isThereKey": false,
64       "keyName": ""
65     },
66     {
67       "type": "door",
68       "keyName": "door4.7",
69       "isOpen": false
70     },
71     {
72       "type": "door",
73       "keyName": "door5.8",
74       "isOpen": false
75     },
76     {
77       "type": "door",
78       "keyName": "door9.6",
79       "isOpen": true
80     },
81     {
82       "type": "mirror",
83       "isThereKey": false,
84       "keyName": ""
85     },
86     {
87       "type": "plain"
88     },
89     {
90       "type": "mirror",
91       "isThereKey": false,
92       "keyName": ""
93     }
94   ],
95
96   "eastWalls": [...
143 ],
```

```

95
96 > "eastWalls": [ ...
143 ],
144
145 > "westWalls": [ ...
192 ],
193
194 "roomsLightStatus": [
195     true,
196     false,
197     false,
198     false,
199     true,
200     false,
201     false,
202     true,
203     false
204 ],
205
206 "northAdjacentRooms": [0,0,0,0,2,0,4,5,6],
207 "southAdjacentRooms": [0,5,0,7,8,9,0,0,0],
208 "eastAdjacentRooms" : [2,3,0,5,0,0,8,9,0],
209 "westAdjacentRooms" : [0,1,2,0,4,0,0,7,8],
210 "playerInfo": {
211     "position": 1,
212     "goldsWithPlayer": 15,
213     "direction": "east",
214     "keysWithPlayer": [],
215     "hasFlashLight": false
216 }
217 }
218
219

```

You can notice the information of each room from this file.

For Example, we can read from the JSON file that the room number 1 is only connected with room number 2 from the East side of room 1 (the first index in the “*eastAdjacentRooms* = 2 “

And the player status is:

- The player in the position 1 (room number 1) which is the initial state for the player in this map
- He has 15 golds and no flash light with him neither keys, And he faced the east wall of the room 1



## **MAVEN => GSON:**

I created maven project to add GSON library as a dependency to my project, in purpose to dealing with JSON files (read / write).

## **Create The Rooms:**

After the reading of JSON file step.

we will create the rooms as the following:

- class Room is created, each room will be an instance of this class and has these properties:
  - number of the room
  - item on north wall
  - item on south wall
  - item on east wall
  - item on west wall
  - north adjacent room
  - south adjacent room
  - east adjacent room
  - west adjacent room
  - is the room lit? (light status)

The rooms are created as a graph ( List of rooms ) each room has a pointers to the rooms which are connecting with it.

Steps of creating the rooms:

- declare a list of rooms.
- add a null as the first index ( to point to it from the rooms which doesn't adjacent any room in some sides (room number 0) ).
- loop over the number of rooms ( the size of any array of which come from the JSON file like, northWallsItems.
- create a room and set the items on the forth walls as they come from the JSON file, each wall can get its item from the corresponding array (EX: northWallsItems.get(currentIndex) ).

- after the first loop finished, loop again on the rooms to specify each side of each room to which room is connected.
- done.

```
1  public void createRooms(){
2      setRoomsWithItems();
3      setAdjacentRooms();
4  }
5
6  private void setRoomsWithItems() {
7      int numberOfRooms = itemsOnNorthWalls.size();
8      rooms.add(null);
9      for (int counter = 0 ; counter < numberOfRooms; counter++){
10         int roomNumber = counter +1;
11         Room room = new Room(
12             roomNumber,
13             itemsOnNorthWalls.get(counter),
14             itemsOnSouthWalls.get(counter),
15             itemsOnEastWalls.get(counter),
16             itemsOnWestWalls.get(counter),
17             roomsLightStatus.get(counter)
18         );
19         rooms.add(room);
20     }
21 }
22
23
24 private void setAdjacentRooms() {
25     int numberOfRooms = itemsOnNorthWalls.size();
26     for (int counter = 0; counter < numberOfRooms; counter++) {
27         int currentRoomNumber = counter+1;
28         int indexOfNorthAdjacentRoom = northAdjacentRooms.get(counter);
29         int indexOfSouthAdjacentRoom = southAdjacentRooms.get(counter);
30         int indexOfEastAdjacentRoom = eastAdjacentRooms.get(counter);
31         int indexOfWestAdjacentRoom = westAdjacentRooms.get(counter);
32
33         Room currentRoom = rooms.get(currentRoomNumber);
34         currentRoom.setNorthAdjacentRoom(rooms.get(indexOfNorthAdjacentRoom));
35         currentRoom.setSouthAdjacentRoom(rooms.get(indexOfSouthAdjacentRoom));
36         currentRoom.setEastAdjacentRoom(rooms.get(indexOfEastAdjacentRoom));
37         currentRoom.setWestAdjacentRoom(rooms.get(indexOfWestAdjacentRoom));
38     }
39 }
```

## Create The Player:

each player is represented as an instance from the Player class which has these properties:

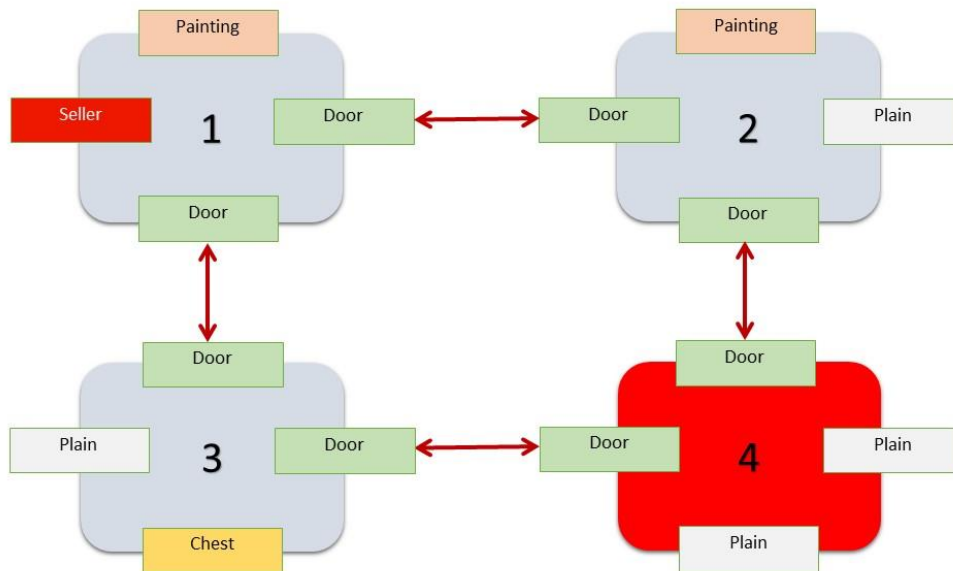
- amount of golds with player
- the current position of player
- the direction or orientation of the player
- the current room which the player is inside it.
- (Set) of keys which are with the player
- is the player has a flashlight?

To create a player just declare an instance of Player class and specify the initial status of the player ( current room, position, direction, golds, flashlight)

## The Items On The Walls:

All of the items are representing as instances from their classes (Seller, Painting, Mirror, Plain, Chest, Door) and All of them implement the **Item interface**.

## The easy level (easy map):




Nothing new about generating this map, All steps are the same as the previous map.

## The commands of the game:

During the game.. you can interactive with the player through some commands like (move forward/backward, turn right ,turn left ...etc.)

Let's take a look how these commands work.

- **turnLeft() / turnRight():**
  - will change the orientation of the player depending on his current direction, EX: (if the player faced the **West** and ask him to turn **right** → he will face the **North**.)



```
1  public void turnRight(){
2
3      if(isFacingNorth())
4          faceEast();
5
6      else if(isFacingEast())
7          faceSouth();
8
9      else if(isFacingSouth())
10         faceWest();
11
12     else if(isFacingWest())
13         faceNorth();
14 }
```

- **moveForward():**

- At the first will check if the faced wall is a door.
- Then check if the door is open.
- After that depending on the players' direction do the following:
  - declare a room and give it the pointer to the adjacent room of the current room from the side which is faced from the player.
  - make the current room is the adjacent which we got it in the previous step.
  - make the player position is the number of new current room.
- EX: (If the player facing the North.. 1- get the North Adjacent Room (let's say Room2). 2- make the current room is Room2. 3- set the position of the player to (2).

```
1 public void moveForward(){
2     Item theFacedItem = getTheFacedItem();
3     if(theFacedItemIsDoor(theFacedItem)){
4         Door theFacedDoor = (Door) theFacedItem;
5         if(theFacedDoor.isOpen()){
6             if(isFacingNorth()){
7                 Room northAdjacentRoom = this.currentRoom.getNorthAdjacentRoom();
8                 this.currentRoom = northAdjacentRoom;
9             }
10 >         else if(isFacingEast()){ ...
14 >         }
15 >         else if(isFacingSouth()){ ...
19 >         }
20 >         else if(isFacingWest()){ ...
24 >         }
25         this.position= currentRoom.getRoomNumber();
26     }
27 >     else{ ...
29 >     }
30 }
31 > else{ ...
33 > }
34 }
```

- **moveBackward():** the same as forward but at first will turn the player around, then moveForward.
- **look():**
  - This method is declared in the **Item** interface, so each item must have this method.
  - First of all.. this method will check if the room is lit =>
  - Then, will print to the user the faced item name on the faced wall
  - If the room wasn't lit => then it will print ("Dark").

```
1  public String look(){
2      if(currentRoom.isTheRoomLit()){
3          Item theFacedItem= getTheFacedItem();
4          return theFacedItem.look();
5      }
6      return "Dark";
7  }
```

- **check():**
  - This method declared in the **Checkable** Interface.
  - These are the items which are implement that interface (Door, Chest, Mirror, Painting)
  - First of All, will check if the faced item is checkable?
  - then apply this method using this item as following:
    - The Door: will show you if it open or closed and the required key name to open the door

- The Mirror: will give you the key which is behind it if was there a key, or will print (nothing behind this Mirror)
- The Painting: same as Mirror
- 
- The Chest: If it is open, the items inside the chest will move to the player items and show you all the items acquired from the chest, OR if the chest was closed, will print that the chest is close and the required key name to unlock this chest.

```
public void check(){
    Item theFacedItem = getTheFacedItem();
    if(isTheItemCheckable(theFacedItem)){
        Checkable checkableFacedItem = (Checkable) theFacedItem;
        checkableFacedItem.check(this);
    }
    else{
        System.out.println("You can't check this item.\n " +
            "you can only check the (Mirror, Painting, Door, And Chest).");
    }
}
```

```
@Override
public void check(Player player) {
    if(isThereKey){
        player.keysWithPlayer.add(this.keyName);
        isThereKey=false;
        System.out.println( "The (" +keyName+ ") key was acquired");
    }
    else {
        System.out.println( "Nothing behind this Mirror.");
    }
}
```



- **open():**
  - This method is applicable only by (Door AND Chest)
  - The first step is checking if the item is Door or Chest.
  - Then, check if the key name of this item is existing with the player.
  - If the player has the key, make the door/chest open.

```
public void open(){
    Item theFacedItem = getTheFacedItem();
    if(isTheFacedItemDoor(theFacedItem))
        openTheDoor(theFacedItem);
    else if (isTheFacedItemChest(theFacedItem))
        openTheChest(theFacedItem);
    else
        System.out.println("You can apply this command only on doors or Chest");
}

public void openTheDoor(Item theFacedItem){
    Door theFacingDoor = (Door) theFacedItem;
    String requiredKey = theFacingDoor.getKeyName();

    if(theFacingDoor.isOpen()){
        System.out.println("The door has already opened");
    }
    else{
        if(isTheKeyExistInPlayerKeys(requiredKey)){
            theFacingDoor.isOpen(true);
        }
        else{
            System.out.println("* ( " + requiredKey + " ) key required to unlock");
        }
    }
}
```

- **close():**
  - The opposite of open() => same stapes but opposite output.

- **trade();**

- This command can be executed only if the player is facing a seller.
- For simplify the operations, will consider the price of each key is 10 golds , And for flashlight is 5 golds.( sell OR buy ).
- The first step is checking if the faced item is a seller.
- Then, list the all items with the seller.
- After that print out the commands which the user can execute to interact with the trade mode, And wait the response from the user input.
- Then, separate the command to two parts first one is the order( buy, sell, list), the second one is the item name (flashlight, key name)
- If the command was: buy flash OR buy "key Name":
  - check if the player has enough golds.
  - then check if the seller has the item for sale.
  - then add the item to the user and remove it from the seller.
  - decrease the amount of golds for the player as much as the price of the item.
- If the command was: sell flash OR sell "key Name"
  - check if the player has the item.
  - remove the item from the player and add it to the seller.
  - increase the amount of golds for the player as much as the price of the item.
- If the command was: list, list the items with the seller again.
- If the command was: finish, exit the trade mode.

```

1  public void trade(){
2      Item theFacedItem = getTheFacedItem();
3      if(isTheFacedItemSeller(theFacedItem)){
4          Seller theFacedSeller = (Seller) theFacedItem;
5          theFacedSeller.listItems();
6          printTheTradeCommands();
7          Scanner inputScanner = new Scanner(System.in);
8          String inputCommand= inputScanner.nextLine();
9
10         while(theCommandIsNotFinish(inputCommand)){
11             executeTheCommand(inputCommand, theFacedSeller);
12             System.out.println("You are in the trade mode, Enter your next command...");
13             inputCommand=inputScanner.nextLine();
14         }
15     }
16     else{
17         System.out.println("You can execute this command only when you faced a seller.");
18     }
19 }

```

```

1  public void executeTheCommand(String inputCommand,Seller theFacedSeller){
2
3      String order = getTheOrderPartFromTheCommand(inputCommand);
4      String item = getTheItemPartFromTheCommand(inputCommand);
5
6      if(isTheOrderBuy(order)){
7          if(item.equalsIgnoreCase("flash")){
8              buyFlashlight(theFacedSeller);
9          }
10         else{
11             String keyName = item.toLowerCase();
12             buyKey(keyName,theFacedSeller);
13         }
14     }
15     else if(isTheOrderSell(order)){
16         if(item.equalsIgnoreCase("flash")){
17             sellFlash(theFacedSeller);
18         }
19         else{
20             String keyName = item.toLowerCase();
21             sellKey(keyName, theFacedSeller);
22         }
23     }
24     else if(order.equalsIgnoreCase("list")){
25         theFacedSeller.listItems();
26     }
27 }

```

```

1 private void buyKey(String keyName, Seller theFacedSeller) {
2     int priceOfKey=10;
3
4     if(hasTheKey(keyName)){
5         System.out.println("You already have this key");
6     }
7     else{
8         if(hasEnoughGolds(priceOfKey)){
9             if(theFacedSeller.hasItemForSale(keyName)){
10                 goldsWithPlayer -= priceOfKey;
11                 keysWithPlayer.add(keyName);
12                 theFacedSeller.removeItemFromTheList(keyName);
13             }
14             else {
15                 System.out.println("The seller doesn't have this key for sale.");
16             }
17         }
18         else{
19             System.out.println("you don't have enough Golds to buy this key ( "+keyName+" ).");
20         }
21     }
22 }
23
24 private void sellKey(String keyName, Seller theFacedSeller) {
25     int priceOfKey=10;
26     if(hasTheKey(keyName)){
27         keysWithPlayer.remove(keyName);
28         goldsWithPlayer += priceOfKey;
29         theFacedSeller.addItemToTheList(keyName,priceOfKey);
30     }
31     else {
32         System.out.println("You can't sell this key, because you don't have it.");
33     }
34 }
35
36

```

- **useFlashlight():**

- For turning the light of the room ON/OFF.
- First, check if the player has a flashlight.
- then turn the light on if the room is not lit.
- 
- turn the light off if the room is lit.



```
1  public void useFlashlight(){
2      if(hasFlashlight()){
3          if(currentRoom.isTheRoomLit()){
4              turTheLightOff();
5          }
6          else {
7              turnTheLightOn();
8          }
9      }
10     else{
11         System.out.println("You don't have a flashlight.");
12     }
13 }
```

- **restart:**

- will reset the map to the initial state and start the game again.

- **save:**

- will save your status (checkpoint) and exit the game.

- **quit:**

- will close the game and the player will consider as a loser.

## Clean Code Part:

During the development process of this project (coding),  
I tried to make my code as clean as possible.  
Clean, mean readable, changeable, easy to understand,  
And well named (variables, classes, methods).

This is my first attempt to write a clean code with a full project, so it may be not perfect one, or not 100% clean, but at least it's much better than before trying that.

The code may be seeming longer when you try to make it clean, but that doesn't matter, the important thing is to make your code clean and readable.

I read a great book about clean code which is (Robert Martin's "Clean Code" book.), And I try to apply what I learned from this book on my code.

In this section, I will display some points or list of clean code concepts to avoid the code smell, which I satisfy it in my code, and will display the snippet code of this concept here if it was need to show.

## Avoiding code smell:

### Comments:

comments almost bad things in the code, because the comments may lie, but the code never lie.

your code should be readable without needing comments to describe it.

some types of bad comments: (Inappropriate information, Obsolete comment, Redundant comment, Poorly written comment, And Commented-out code).

I never used comments in my code.

### Functions:

- **Too many arguments:**

“Functions should have a small number of arguments. No argument is best, followed by one, two, and three. More than three is very questionable and should be avoided with prejudice.”

- **Output Arguments:**

“Output arguments are counterintuitive. Readers expect arguments to be inputs, not outputs.”

- **Flag Arguments:**

“Boolean arguments loudly declare that the function does more than one thing. They are confusing and should be eliminated.”

- **Dead Function:**

“Methods that are never called should be discarded. Keeping dead code around is wasteful.”

```

private boolean isTheFacedItemChest(Item theFacedItem){...}

public void moveBackward(){...}

public String look(){...}

public void playerStatus() {...}

public void check(){...}

private boolean isTheItemCheckable(Item item) { return isTheItemNotWall(item) && isTheItemNotSeller(item); }

private boolean isTheItemNotWall(Item item) { return ! typeOfItem(item).equalsIgnoreCase( anotherString: "Wall"); }

private boolean isTheItemNotSeller(Item item) { return ! typeOfItem(item).equalsIgnoreCase( anotherString: "Seller"); }

public void open(){...}

public void openTheDoor(Item theFacedItem){...}

public void openTheChest(Item theFacedItem){...}

private boolean isTheKeyExistInPlayerKeys(String requiredKey ){...}

public void close(){...}

private void closeTheDoor(Item theFacedItem) {...}

private void closeTheChest(Item theFacedItem) {...}

public void trade(){...}

public void printTheTradeCommands(){...}

```



## GENERAL:

- **Obvious Behaviour Is Unimplemented:**

“Following “The Principle of Least Surprise,”<sup>2</sup> any function or class should implement the behaviours that another programmer could reasonably expect.”

```
public boolean isFacingWest () { return player.getDirection().equalsIgnoreCase( anotherString: "west"); }

public void turnRight(){...}

public void turnLeft(){...}

public void turnAround(){...}

public void moveForward(){...}

public Item getTheFacedItem(){...}
```

- **Duplication**

“Every time you see duplication in the code, it represents a missed opportunity for abstraction. That duplication could probably become a subroutine or perhaps another class outright.”

```
public void turnRight(){

    if(isFacingNorth())
        faceEast();

    else if(isFacingEast())
        faceSouth();

    else if(isFacingSouth())
        faceWest();

    else if(isFacingWest())
        faceNorth();

}

public void turnLeft(){...}

public void turnAround(){
    turnRight();
    turnRight();
}
```

- **Dead Code:**

Dead code is code that isn't executed. You find it in the body of an if statement that checks for a condition that can't happen. You find it in the catch block of a try that never throws. You find it in little utility methods that are never called or switch/case conditions that never occur.

EX:

```
private boolean isTheFacedItemDoor(Item theFacedItem) { return typeOfItem(theFacedItem) == "Door"; }

private String typeOfItem(Item obj){...}

private boolean isTheFacedItemPainting(Item theFacedItem) { return typeOfItem(theFacedItem) == "Painting"; }

private boolean isTheFacedItemSeller(Item theFacedItem) { return typeOfItem(theFacedItem) == "Seller"; }

private boolean isTheFacedItemMirror(Item theFacedItem){...}

private boolean isTheFacedItemChest(Item theFacedItem){...}

public void moveBackward(){...}
```

I never used these methods, so at the end I got rid of them.

- **Vertical Separation:**

Variables and function should be defined close to where they are used. Local variables should be declared just above their first usage and should have a small vertical scope. We don't want local variables declared hundreds of lines distant from their usages.

Private functions should be defined just below their first usage. Private functions belong to the scope of the whole class, but we'd still like to limit the vertical distance between the invocations and definitions. Finding a private function should just be a matter of scanning downward from the first usage.

```

public void useFlashlight(){
    if(player.isHasFlashlight()){
        if(player.getCurrentRoom().isTheRoomLit()){
            turTheLight0ff();
        }
        else {
            turnTheLightOn();
        }
    }
    else{
        System.out.println("You don't have a flashlight.");
    }
}

private void turTheLight0ff() { player.getCurrentRoom().makeTheRoomDark(); }

private void turnTheLightOn() { player.getCurrentRoom().makeTheRoomLit(); }

```

- **Inconsistency:**

“If you do something a certain way, do all similar things in the same way. This goes back to the principle of least surprise. Be careful with the conventions you choose, and once chosen, be careful to continue to follow them.”

```

private boolean isTheFacedItemDoor(Item theFacedItem) { return typeOfItem(theFacedItem) == "Door"; }

private String typeOfItem(Item obj){...}

private boolean isTheFacedItemSeller(Item theFacedItem) { return typeOfItem(theFacedItem) == "Seller"; }

private boolean isTheFacedItemChest(Item theFacedItem){...}

```

- **Clutter:**

Of what use is a default constructor with no implementation?

All it serves to do is clutter up the code with meaningless artifacts.

Variables that aren't used, functions that are never called, comments that add no information, and so forth. All these things are clutter and should be removed. Keep your source files clean, well-organized, and free of clutter.

- **Obscured Intent**

We want code to be as expressive as possible. Run-on

expressions, Hungarian notation, and magic numbers all obscure the author's intent.

```
private void sellKey(String keyName, Seller theFacedSeller) {  
    int priceOfKey=10;  
  
    if(hasTheKey(keyName)){  
        player.keysWithPlayer.remove(keyName);  
        int goldsBeforeTheSale = player.getGoldsWithPlayer();  
        player.setGoldsWithPlayer(goldsBeforeTheSale + priceOfKey);  
        theFacedSeller.addItemToTheList(keyName,priceOfKey);  
    }  
    else {  
        System.out.println("You can't sell this key, because you don't have it.");  
    }  
}
```

- **Use Explanatory Variables:**

One of the more powerful ways to make a program readable is to break the calculations up into intermediate values that are held in variables with meaningful names.

- **Function Names Should Say What They Do:**

If you have to look at the implementation (or documentation) of the function to know what it does, then you should work to find a better name or rearrange the functionality so that it can be placed in functions with better names.

- **Replace Magic Numbers with Named Constants:**

In general it is a bad idea to have raw numbers in your code. You should hide them behind well-named constants.

- **Encapsulate Conditionals:**

“Boolean logic is hard enough to understand without having to see it in the context of an if or while statement. Extract functions that explain the intent of the conditional.”

```
public void check(){
    Item theFacedItem = getTheFacedItem();

    if(isTheItemCheckable(theFacedItem)){
        Checkable checkableFacedItem = (Checkable) theFacedItem;
        checkableFacedItem.check(player);
    }
    else{
        System.out.println("You can't check this item.\n " +
            "you can only check the (Mirror, Painting, Door, And Chest).");
    }
}

private boolean isTheItemCheckable(Item item){ return isTheItemNotWall(item) && isTheItemNotSeller(item); }
private boolean isTheItemNotWall(Item item){ return ! typeOfItem(item).equalsIgnoreCase( anotherString: "Wall"); }
private boolean isTheItemNotSeller(Item item){ return ! typeOfItem(item).equalsIgnoreCase( anotherString: "Seller"); }
```

- **Avoid Negative Conditionals:**

Negatives are just a bit harder to understand than positives. So, when possible, conditionals should be expressed as positives. For example:

```
private boolean isTheItemNotWall(Item item){
    return ! typeOfItem(item).equalsIgnoreCase( anotherString: "Wall");
}

private boolean isTheItemNotSeller(Item item) {
    return ! typeOfItem(item).equalsIgnoreCase( anotherString: "Seller");
}
```

- **Functions Should Do One Thing:**

It is often tempting to create functions that have multiple sections that perform a series of operations. Functions of this kind do more than one thing, and should be converted into many smaller functions, each of which does one thing.

```
public void open(){
    Item theFacedItem = getTheFacedItem();

    if(isTheFacedItemDoor(theFacedItem))
        openTheDoor(theFacedItem);

    else if (isTheFacedItemChest(theFacedItem))
        openTheChest(theFacedItem);

    else
        System.out.println("You can apply this command only on doors or Chest");
}

private void openTheDoor(Item theFacedItem){...}

private void openTheChest(Item theFacedItem){...}
```

- **Avoid Long Import Lists by Using Wildcards:**

If you use two or more classes from a package, then import the whole package.

```
import com.google.gson.JsonArray;
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.JsonParser;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;
```



```
import com.google.gson.*;
import java.io.*;
import java.util.*;
```

## NAMES:

- **Choose Descriptive Names:**

“Don’t be too quick to choose a name. Make sure the name is descriptive.

This is not just a “feel-good” recommendation. Names in software are 90 percent of what make software readable. You need to take the time to choose them wisely and keep them relevant. Names are too important to treat carelessly.”

```
public class Chest implements Item, Checkable {
    String itemName="Chest";
    String keyName;
    boolean isOpen;
    List<String> keysInTheChest = new ArrayList<>();
    boolean isThereFlashLight;
    int amountOfGoldsInsideTheChest;
```

- **Use Standard Nomenclature Where Possible:**

“Names are easier to understand if they are based on existing convention or usage.

Names are easier to understand if they are based on existing convention or usage.”

```
@Override
public String toString() {
    String itemsFound = "* These items are acquired:{\n";

    for(String str : keysInTheChest){
        itemsFound+="\t* (" + str + ") Key\n";
    }

    if(isThereFlashLight){
        itemsFound+="\t* Flash Light.\n";
    }
    itemsFound+= "\t* " + amountOfGoldsInsideTheChest + " GoLds.\n" +'}';
    return itemsFound;
}
```

## Conclusion:

This is a list of some smells which I tried to avoid them in my code, these not all of them, but these are the simplest ones which I can easily describe them and spot the light into them in my code.

Again, it may be not perfect clean code, but I hope that you can read and understand my code easily.



# Design Patterns:

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

There is the design pattern which I used in my project...

- **Singleton Design Pattern:**

A class of which only a single instance can exist.

```
public class MapSaver {  
    public static MapSaver obj = new MapSaver();  
  
    private MapSaver() {  
    }  
  
    public static MapSaver getInstanceOfMapSaver() {  
        return obj;  
    }  
}
```

```
else if(inputCommand.equals("save")){  
    MapSaver mapSaver = MapSaver.getInstanceOfMapSaver();  
    mapSaver.saveData(map);  
    System.out.println("Your status is saved successfully.");  
    break;  
}
```

- **Builder Design Pattern:**

Separates object construction from its representation

The main advantages of Builder Pattern are as follows:

- It provides clear separation between the construction and representation of an object.
  - It provides better control over construction process.
  - It supports to change the internal representation of objects.
  - You don't have to remember the sequence of the constructor's arguments when you need to declare an instance.
- I applied this design pattern on chest class, because it has a lot of arguments to be send in the constructor (this is one reason to use this pattern).

*Chest class:*

```
public class Chest implements Item, Checkable {
    String itemName="Chest";
    String keyName;
    boolean isOpen;
    List<String> keysInTheChest = new ArrayList<>();
    boolean isThereFlashLight;
    int amountOfGoldsInsideTheChest;

    public Chest(String keyName, boolean isOpen, List<String> keysInTheChest, boolean isThereFlashLight, int amountofGoldsInsideTheChest) {
        this.keyName = keyName;
        this.isOpen = isOpen;
        this.keysInTheChest = keysInTheChest;
        this.isThereFlashLight = isThereFlashLight;
        this.amountOfGoldsInsideTheChest = amountofGoldsInsideTheChest;
    }
}
```

## ChestBuilder Class

```
import java.util.*;

public class ChestBuilder {

    private String keyName;
    private boolean isOpen;
    private List<String> keysInTheChest = new ArrayList<>();
    private boolean isThereFlashLight;
    private int amountOfGoldsInsideTheChest;

    public ChestBuilder setKeyName(String keyName) {...}

    public ChestBuilder setIsOpen(boolean open) {...}

    public ChestBuilder setKeysInTheChest(List<String> keysInTheChest) {...}

    public ChestBuilder setIsThereFlashLight(boolean thereFlashLight) {...}

    public ChestBuilder setAmountOfGoldsInsideTheChest(int amountOfGoldsInsideTheChest) {...}

    public Chest getChest(){
        return new Chest( keyName, isOpen, keysInTheChest, isThereFlashLight, amountOfGoldsInsideTheChest);
    }
}
```

## Create the instances of the Chest

```
private Item createChest(JsonObject jsonObject) {

    String keyName = jsonObject.get("keyName").getAsString();
    boolean isOpen = jsonObject.get("isOpen").getAsBoolean();
    boolean isThereFlashlight = jsonObject.get("isThereFlashLight").getAsBoolean();
    int golds = jsonObject.get("golds").getAsInt();
    List<String> keysInTheChest = new ArrayList<>();
    JsonArray keysJson = jsonObject.getAsJsonArray( memberName: "keys");

    for( JsonElement element : keysJson){
        String key = element.getAsString();
        keysInTheChest.add(key);
    }

    ChestBuilder chestBuilder = new ChestBuilder();
    chestBuilder.setKeyName(keyName);
    chestBuilder.setAmountOfGoldsInsideTheChest(golds);
    chestBuilder.setIsThereFlashLight(isThereFlashlight);
    chestBuilder.setIsOpen(isOpen);
    chestBuilder.setKeysInTheChest(keysInTheChest);

    Item chest = chestBuilder.getChest();
    return chest;
}
```

## Used Data Structures:

In this project, I used many types of data structure to complete the project and to work consistently.

I will mention most of them with the reason of choosing each one.

- **Array List:**  
I used this for creating a list of items which I read them from the JSON file and for used them in the creating room process.
- **Graph:**  
Graph was very important data structure in my project, I needed it for connect the rooms together each one with its adjacent rooms.
- **Hash Map:**  
This is used with the Seller class, for storing each item with its price, EX: ("door1.2 key", 10).
- **Hash set:**  
used it in the Player class, for storing the keys with the player, And the keys should not be redundant.
- And default arrays and strings...etc.

## Conclusion:

I tried to apply in this project most of things which I learned recently about writing clean code and design patterns and styling rules..etc.

The result is not perfect, or correct 100%, but I hope that it will be clean and readable and stable in a high percent.

I learned a lot during this project and every time I checked the code, I detect or think about something might be better in a way.

Before this project, all what matter to me, was the final status of the code (is it works or not), I wasn't care about readability of the code and the cleaning up of it.

Now, I changed my mind and this is a great experience, and I hope to improve this skills much better, to end up with clean and perfect code.

## References:

- Robert Martin's "Clean Code" book.
- Joshua Bloch's "Effective Java" book.
- <https://google.github.io/styleguide/javaguide.html>