World Navigator Game

This Document was submitted in Partial Fulfillment of the Requirements for the Software Engineering and DevOps Bootcamp.

BY:

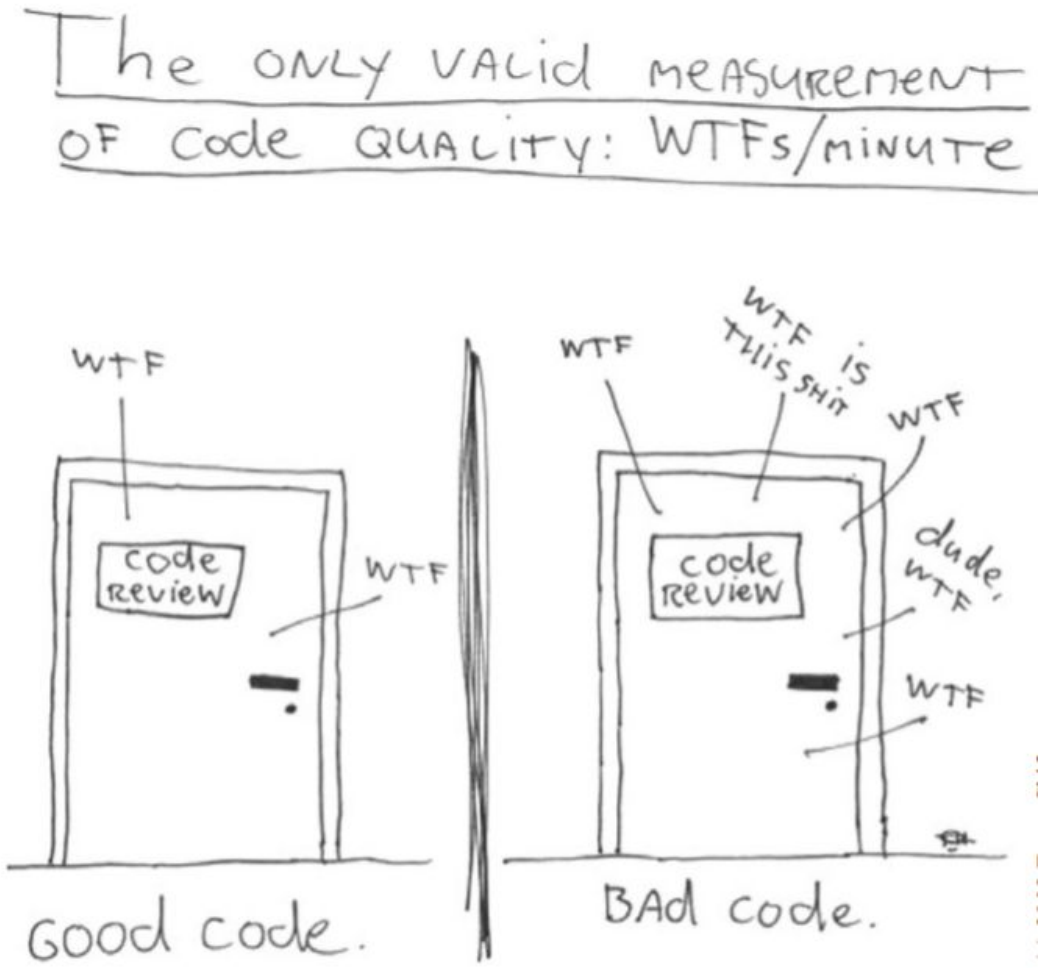
M u n t a s e r A b u k h a d i j a h

TABLE OF CONTENT

Just a few points,

- I tried as much to make this explanation about how the code applies the rules not explain the rules.
- Most of the points will show the changes in the code from its first version, to version after some rules have been applied to it.
- I did not write about this point “Concurrency issues”, i did not use a separate thread for timing, i used a simple timer that will check the time after each command, but will not terminate the thread if the time ended before the next check.
- And I think i used “I think” a lot :)

Clean code principles



Which door represents your code? After reading Clean code book and Effective java book, I realized why there is no third door without any “WTF”.

MEANINGFUL NAMES

Let's say we have this "check" method,

```
public ItemsCart check() {
// Do something
}
```

Is it possible for us to know what this method checks from its name? What the return type ItemCart represents? I think there are a lot of the possibilities, but what if the name became "**checkFurniture**", it is easier to answer the previous questions now, Yes we use checkFurniture to check furnitures(Door, chest...), and by saying "checkFurniture to check furniture", then we could be confident to say we applies the "**Use Intention-Revealing Names**" rule on this method name, What about the second question? By knowing that using this method we check the furniture, we could now see that the returned ItemsCart represents all the items that the checked furniture contains.

Let's say i want to talk with someone about my code, I don't think that i will face any problem to have a conversation about one of the classes or methods in the code, there will not be any sentence sounds like "hey omar, why **get Ne Di** method returns wrong results, that is Weird!", but instead the sentence would sound like "hey omar, why **get next direction** method returns wrong results, that is weird!" and by having good understandable sentences when we talk about our code, that means we use "**Pronounceable Names"** in our code.

What would happen if i wanted to search about the variable that represents the position of the player on the X-axis, Note that the variable name is 'x', in Player class that will give 35 matches, half of them related with the 'x' as variable, it is a lot, isn't it? That is because 'x' is not "**Searchable Name"**, also it violates "**Use Intention-Revealing Names**", so let's find another name, how about "rowPosition", I think it sounds good.

In world navigation assignment there are two lockable furnitures, Chest and Door, so by sense we could expect their code should have one of these words lock or padlock, but what if the door has lock, and the chest has padlock word, both of them refer to the same concept, so why? And what is the difference? I think one of them should be used, and I used lock word, so it is better to use "**One Word per Concept".**

We are talking about meaningful names, and the names should be clear and telling about their responsibility, but is naming class as Chest the right thing to do? Does that follow those rules? I

am wondering because chest could have totally different meanings about the context it is used in. but yes it is the right choice to name the class by chest, because it follows the "**Problem Domain Names**" rule, we are talking about maze and furniture, so it is expected to find a chest.

The world navigator requirements describe the room and its content, also it tells that some rooms don't have light switches, so now let's say we have in the Room class field called "state", what does this field represent? the state of whether there is a light switch or not, or the state of the light, is it on or off? To solve this problem we have to "**Add Meaningful Context",** how about "lightState" sounds good and meaningful I think.

FUNCTIONS

Let's see what operations are needed to complete a trade operation, buying operation for example, first we need to check that the sellers item cart is not empty, then we need to take the user choice of which item wants to buy, then to check if the input is valid, And whether or not the item is in the seller's item cart, till this point, if everything is okay, then we have to check the logic, can the player buy this item, Does he have enough money? If yes we want to add the item to the player item cart, subtract the price from player money, and remove the item from the seller's item cart.

Doing that in one function will violate the **"Do one thing"** rule because these at least four operations to be done, and of course, one function to do all these things, it will not be small, and that violates the rule that says a function should be "**small"** and less than "**20 lines".** One possible solution is dividing the problem into the following functions:

```
isItemCartContainsItems(ItemsCart itemsCart){}
isItemIdInItemCart(ItemsCart itemsCart, int itemId){}
getItemId(ItemsCart itemsCart, String operationName){}
checkItemPrice(Item item){}
performTrade(){}
```

By doing this separation, each functions seems to do just one thing, of course the functions are smaller, but we can also see another rule applied here, "**Use Descriptive Names**", and we could expect what is the responsibility of each function, and that is what Ward's principle says "You know you are working on clean code when each routine turns out to be pretty much what you expected."

We may say that checking the validation of the input could be in "getItemId" method for example, but actually that is not the best way, because **"Error Handling Is One Thing"** so there is need to "isItemIdInItemCart" method here,

If we look at the previous methods and their parameters, we will see that two of them have two parameters, the other two have one parameter, and the last one has zero parameter, so these methods apply the “**Function Arguments**” rule.

And if we look at the “move” method in Player class, we can see that this method groups the return values that represent the new (x,y) position, and that is “**Argument Objects”** rule, also this is not cheating because “they are likely part of a concept that deserves a name of its own”.

But also I can't say that this “Function Arguments” rule did not be violated in the code, in some places in the code I could not figure out a way to separate the function more to have less arguments, so the maximum number of arguments in the code is three argument.

”Structured Programming” rule says that “ Following these rules means that there should only be one return statement in a function, no break or continue statements in a loop, and never, ever, any goto statements.”, The only place I used "break" is inside the loop shown below:

```
private String getNextDirection() {
    String[] directions = {"north", "east", "south", "west"};
    int index = 0;
    for (int i = 0; i < directions.length; i++) {
        if(currentOrientation.equalsIgnoreCase(directions[i])) {
            index = i;
            break;
        }
    }
    if (rotationDirection.equals("left")) {
        index = (((index - 1) % 4) + 4) % 4;
    } else {
        index = (((index + 1) % 4) + 4) % 4;
    }
    return directions[index];
}
```

But actually I don't see the point here to not use break, even if the loop is just iterates four times, but still a good place to use break, what is the point to continue in the iteration, while we have already obtained the result!

What about the function that never used, I think the main reason of the “**dead functions”** is the single responsibility principle, when we find that class has more than one reason to change, we

do separate it into classes with single responsibilities, so let's say we have Player Class that do represent the basic information about the player and also perform another action like moving from room to another, so here we have one class with two responsibilities, it seems that we are going to create two classes, one for each responsibility, but that is not the case, usually we don't create two classes, but one for the extra responsibility, and keep the first one, and then we move the methods and the fields that are needed to perform the second responsibility to the new created class, and that most likely will cause us some dead methods in the first/origin class.I removed more than seven methods from the Player class, after applying the SRP on it.

COMMENTS

If we stand in one of the maze rooms and want to use the "look" command, to know what furniture we are facing, the room should have light and the light is on, or the player should have a flashlight and it is on, to express thies conditions in code we can write this:

```
    if (lightState||player.isFlashLightOn()) {
      for (int i = 0; i < walls.size(); i++) {
       // omitted ……..
}
```

Here we have three possibilities that the programmer who wrote this code can think of:

- The condition is clear and no need to add any comments.
- The condition is clear but let's add some comments to illustrate it more.
- The condition is clear but it could be clearer, let's modify it, and it will look like this:

```
if (isRoomLit(player)) {
      for (int i = 0; i < walls.size(); i++) {
       // omitted ……..
}
```

The programmer who will choose the first possibility, has to learn more about clean code.
The programmer who will choose the second possibility, has to know that "**Comments Do Not Make Up for Bad Code".**
The programmer who will choose the third possibility, most likely knows that **he/she has to explain him/herself in code**. Not using comments.

And why do all possibilities say that "The condition is clear..."? That is because most of the time this is what programmers think about what they write.

We are still in the same room that we used “look” command in, and the result of the command was “Empty wall”, so we decided to move backward, to do that, there are three conditions must be met, the furniture on the wall behind us should be door, unlocked, and open, and let’s keep in mind that the we want to move backward, so many approaches could be used here, for example we could have method that handel the backward movements, or another approaches is to fix the player orientation virtually to be facing the back wall as it is front of him, so the point here, this is decision the programmer will take, and in this case if we added a comment to **“explain our Intent”**, it may consider good comment, code below shows example of that:

```
// In case of a backward move, the orientation of the player will be flipped
//virtually.
    this.playerOrientation= fixOrientation(inputDirection);
```

Another reason that writing commit could be helpful thing, when we have obscure argument, and the statement could be misunderstood, or it can be explained in more than one way, for example when we override compareTo method in player class we had this line

```
result = itemsCart.compareTo(o.itemsCart);
```

It is possible to give a lot of expectations about the results of this line, so we may add comments to **“Clarify”** what is the result of the line.

```
    // result == 1  if this.itemCart.size() > o.itemsCart.size()
    // result == -1  if this.itemCart.size() < o.itemsCart.size()
    // result == 0  if this.itemCart.size() == o.itemsCart.size()
    result = itemsCart.compareTo(o.itemsCart);
```

"Journal Comments", when i started read this point, i thought that it is a great idea, why would it be under bad comments section, then I asked myself, didn't I do this every time I start editing and refactoring the code, didn't i write what i fixed and which items i applied on the code in this day?, but also I did not use comments to do that. I used commits, and this points in the book emphasized that there is softwares can track the changes for us. I used this feature that “git” provides us with, this will save us from **"Journal Comments"**, below figure shows the usage of “git” instead of **"Journal Comments"**.

```
f60f2c15e822dee4a283a5c9c70227313474fef1 override compairTo method
c77030d05f44b9416463d4b825e0381f2324588e Apply items 10, 11 and 12 on the code
2b5b9c5ed4b5b2df549a44a6cfa05fc24aae71b9 KeyUsage, FlashLightSwitch, OppositeFur
nitureChecking class added
28e35a0d1cb582fe4d84c6bf55c94154987316a4 SRP applied on Player class
17ba63e2dc9ed67a9ca51e5537977d4bf167e7c1 copyrights added
```

FORMATTING

“How big are most Java source files? It turns out that there is a huge range of sizes and some remarkable differences in style.“
What i want to show here is some simple statitscase of the source files from the assignment solution:

- The largest file in World navigator is about 203 lines.
- The smallest file in World navigator is about 6 lines.
- The average number of lines per file is about 58 lines

Now we know some numbers about the code, but is it well formatted? Can we read it almost as we read the newspaper?Let's see a Class player for example.
That is the order of the methods in the class :

1. getRowPosition	2. getColumnPosition	3. getDirection	4.getAmountOfGold
5. orientationChange	6. Move	7.setDirection	8.printInfo
9.setRowPosition	10.setColumnPosition	11.earnGold	12.payGold
13. Look	14. useFlashLight	15. isFlashLightOn	16.getItemCart

The first thing that could cross our minds, after looking at the order of the methods, is that there are a lot of responsibilities here!Actually I don’t think so, most of these methods are not responsible for doing the actual job, for example the move method uses the PlayerMovement class to do the actual needed steps to move from room to another. But that is not the only thing should cross our minds, first we get information then we change some information about the player state then again print information and the same pattern repeated again, that is not the best way to order the class, we could order them in a better way, we can use this order:

- First we have the method that represents the class fields.(get methods and set methods)
- Then the methods that could change fields based on the players actions (payGold, earnGold)
- Then we could have the methods that represent the actions that the player can do(look,...).

So the order can be like this:

1.getRowPosition	2. setRowPosition	3.getColumnPosition	4. setColumnPosition
5.getDirection	6. setDirection	7.getAmountOfGold	8. earnGold
9.payGold	10. isFlashLightOn	11.useFlashLight	12. printInfo
13.getItemCart	14. orientationChange	15. Move	16. Look
17.getItemCart			

By doing that, we may say that we applied **"The Newspaper Metaphor"**

Why we should apple **"Vertical Openness Between Concepts",** by now we have two reasons, first one, it makes the the code easier to read, if you don't care about that, but you follow "Google Java Style Guide", you have to apply whether you care about the readability or not,

If we go back to the previous page, we can see that we have added three comments to explain the value of "result" variable, actually this way of formatting could affect the **Vertical Density**, the reader could lose his thoughts because of them, it is better to add them on the side of the lines.

In "**Vertical Distance"** point, there is the following sentence:
"closely related concepts should not be separated into different files **unless you have a very good reason**. Indeed, this is one of the reasons that protected variables should be avoided."

Here comes the problem, how we can measure the reasons!, in the solution, i have superclass called Trade, in this class there are four protected variables, of course having them in the superclass will affect on achieving the **vertical distance**, but the reason i have them as protected is that i will need to use them in the subclasses, to avoid the duplication i wrote them in the superclass, is this "a very good reason"? I think so.

Under the "Vertical Distance" point we have subpoint says " local variables should appear a the top of each function", But this is the opposite of what effective java book and Google Java Style Guide say, if we look at effective java book we will find that item57 says "minimize the scope of local variables" and in google guide we will find it says "Declared when needed", and it talks about declaring the local variable, so in this case i decided to go with effective java book and google style, because i see there way is better, and it is better because it avoids the reader from guessing the usage of this variable, and give us more flexibility to use the same variable name in the same method if they were in different scopes.

In keyUsage class, the class that responsible of open doors or chests using a key, we could see that there order follows the **"Dependent Functions"** rule, for example "useKey" method call "isLockableFurniture" method, and "isLockableFurniture" method is right after it, and the same for "getKeyNames" method that call "getKeysList" and it is right after it. And in another class, in

Room class, this class represent the room object, below code shows two method from this class:

```
private void lightSwitchOn() {
 Omitted ….
}
private void lightSwitchOff() {
  Omitted ….
}
```

Here these two methods don't have direct dependence, but they are still near each other, that is because there are other reasons to order methods near each other, one of these reasons is **"Conceptual Affinity"** and as we see in this case, the two methods perform similar operations.

"Horizontal Alignment", this point will be discussed later on, in "Google Java Style Guide" section, But we can note here that under "Uncle Bob's Formatting Rules", we can see that these rules have a lot in common with google style guide, the only difference i noticed that in case the "if" statement or the loop has one line, Robert Martin doesn't use the curly brackets unlike what google java style guide says.

Error Handling

I think a lot of errors happen because of bad inputs, especially when the input should match some patterns, so I decided that the user will deal with a list of options, and will choose his/her choice using the option id. And of course the inputs are not the only thing responsible for errors, but let's start with that.

To make the code more organize and with less duplication, i created boundary object, PlayerConsole class, and it is responsible to talk with the end users, to display the available options to the end users, and to take the inputs, and here are some errors could happen though this process, what if the user input was "null" or the input was from different data type, or even input from the same data type but still not valid input, these different possibility could cause different errors, so i used **try-catch-finally** to handle checking the inputs, as shown below***

```
public static int readMenuChoice(String prompt, String[] menu) {
  // Omitted ……
  try{
   choice = scanner.nextInt();
 }catch (Exception e){
    System.out.println(e);
```

```
        scanner.next();
    }finally{
        System.out.println("\n-------------------------------------\n");
        return choice;
     }
  }
```

But here comes two questions, the exception message for the end users or for the programmers? When there is an exception, should the program terminate or we should handle the error without any termination?

These equations are important to be answered, for example if there is an exception message for the end user, it should not have technical information, and for the second question, If the program will terminate, is it a good reason to terminate the program because of wrong input?!

Let's answer the first question, short answer, exception messages aren't for end users, but why? I could menthind there reasons here:

- Exception messages should be helpful to fellow programmers, to help them know what is wrong, and that include that exception messages should be clear, honest, and describe the reasons of the exception.
- Security, because exception messages may leak information about how the system works.
- Are end uses programmes? End users don't speak Java for example. And even if it's not about programming language, what if the user doesn't speak english, are we going two write the messages in their languages?!

Well, we have finished the first question, should we terminate the program or not in case of exception error. I think it is not reasonable to terminate the program if the inputs are wrong, I chose to tolerate, and give the user another chance to enter the option Id.

Another reason that could cause errors, is returning null, the rule says **"Don't Return Null"** and we are going to talk about that in ITEM 54 from effective java book, but i would like to show here some lines of code about how the code would look like if we did return null, in Chest class we have method called checkFurniture, it will return the items list that chest has in case the chest not locked, but what if it is? We are not going to return null, just an empty items list, that says we don't have your request, but how is that affecting the code? Let's take a look at one of the methods that call checkFurniture method,

```
public void checkFurniture() {
   if (isItTheRightFurniture()) {
     ItemsCart itemsCart = ((CheckableFurniture) furniture).checkFurniture();
       if( itemCart == null)   // because we don't return null
```

```
            // Do sth                there is no need for this check
    if (itemsCart.numberOfItems() > 0) {
     // Omitted.....

  }
```

Classes

We payed attention even to the number of indentation in the format section, and then how the comments should be written, the names, number of lines of each method and class, number of parameters, now it’s time for us to step back a little bit, and see the the bigger picture, let's talk about the clean classes.

Is following class, **organized class** or not?

```
public class Player implements Comparable<Player> {

  private int rowPosition;
  private int columnPosition;
  private String direction;
  private int amountOfGold;
  private ItemsCart itemsCart;

  public Player() { /*Omitted …*/ }
  public int getRowPosition() {/*Omitted …*/  }
  public int getColumnPosition() {/*Omitted …*/ }
  public String getDirection() {/*Omitted …*/ }
  public int getAmountOfGold() {/*Omitted …*/ }
  public void orientationChange(String rotationDirection, Room room) {
    PlayerDirectionRotate playerDirectionRotate =
        new PlayerDirectionRotate(rotationDirection, getDirection());
    setDirection(playerDirectionRotate.getOrientation());
  }

  private void setDirection(String orientation) {
    this.direction = orientation;
  }

  public void move(String movementDirection, Room room) {
```

```
    PlayerMovement playerMovement = new PlayerMovement(getDirection(), room,
movementDirection);
    Pair pair = playerMovement.move();
    if (pair.getY() + pair.getY() != -2) {
      setRowPosition(pair.getX());
      setColumnPosition(pair.getY());
    }
  }

  private void setRowPosition(int x) {
    rowPosition += x;
  }

  private void setColumnPosition(int y) {
    columnPosition += y;
  }
  public void printInfo() { /*Omitted …*/ }
/*Omitted ..............
```

I think it is, because it is follow these rules:

- a class should begin with a list of variables.
- Public functions should follow the list of variables.
- The private utilities that are called by a public function comes right after the public function itself. (As the highlight colors shows)

More about classes will be discussed later on, in the SOLD section.

JAVA CODE PRINCIPLES

Methods Common to All Objects

ITEM 10: OBEY THE GENERAL CONTRACT WHEN OVERRIDING EQUALS

It was confusing how to determine if this class should override the "equal" method or not, but if we follow the below rule, the process will be easier.
"Is it possible that objects of this class could end up in collections?"
If the answer is "Yes" then override the "equal" method, otherwise most likely there is no need to do that. So classes that represents some data like : Chest, Door, Player, Seller, Room, Mirror and Key, They should override “equal” method, but objects that not represent data, and these objects could be classified as control objects or boundary objects most likely should not override it, and these objects like :Buy(Control object), Sale(Control object) and PlayerConsole(Boundary object).
Below code shows the override of the “equal” method :

```
@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null || getClass() != o.getClass()) return false;
  Key key = (Key) o;
  return price == key.price &&
```

```
            keyName.equals(key.keyName);
  }
```

ITEM 11: ALWAYS OVERRIDE HASHCODE WHEN YOU OVERRIDE EQUAL

We are not going to explain why we MUST override hashcode when we override equals but this quotation from the book is a good reason : “You must override hashCode in every class that overrides equals.”

And below code shows the override of the “hashcode” method :

```
@Override
  public int hashCode() {
    return Objects.hash(keyName, price);
  }
```

ITEM 12: ALWAYS OVERRIDE toString

The effective java book says “Whether or not you decide to specify the format, you should clearly document your intentions.” and the book provide us two types/ways to document “toString” method one of them should specify the format and the other not, i decided to use the specified format with the “entity object”, the objects that represent data and real world objects, I decided that, because I think the “entity object” is the more needed to be printed for debugging purposes than other types of objects.

Following code shows the “toString” method in the Player class with its documentation that follow the specified format,

```
/**
   * Returns the string representation of this Player. The string
   * consists whose format is "Player{rowPosition=XXX',
```

```
 * columnPosition=YYY',
 * direction=ZZZ, amountOfGold=TTT, itemsCart=NNN }"
 * where XXX is the X position of the player inside the map,
 * YYY is Y position, ZZZ is the players orientation,
 * TTT is the amount of gold that a player has.
 * NNN is the description of each item the player has.
 */

@Override
public String toString() {
  return "Player{" +
          "rowPosition=" + rowPosition +
          ", columnPosition=" + columnPosition +
          ", direction='" + direction + '\'' +
          ", amountOfGold=" + amountOfGold +
          ", itemsCart=" + itemsCart +
          '}';
}
```

Following code shows the “toString” method in the KeyUsage class with its documentation that follow the not specified format,

```
/**
 * Returns a the player who want to use key, with list of keys and names
 * and the room that player stands in.
 */

@Override
public String toString() {
  return "KeyUsage{" +
          "player=" + player +
          ", room=" + room +
          ", keyList=" + keyList +
          ", keyNames=" + Arrays.toString(keyNames) +
          '}';
}
```

And the returned string could be parsed to extract data from it, but this process is not the best practice, because it could cause:

- Reducing the performance.
- Making unnecessary work for programmers.

- Error-prone and results in fragile systems that break if any change happens in the format.

To avoid the need to parse the returned string i Provided programmatic access to all of the information contained in the value returned by toString().

```
  public int getRowPosition() {
    return rowPosition;
  }

  public int getColumnPosition() {
    return columnPosition;
  }

  public String getDirection() {
    return direction;
  }

  public int getAmountOfGold() {
    return amountOfGold;
}
```

ITEM 14: CONSIDER IMPLEMENTING COMPARABLE

Do I have classes that need to implement comparable interface? I think each value class should implement comparable, because at some point during the development of the program maybe we will need to have a collection of this object, and we need it to be ordered by one or more of its fields. Let's say for some reason we want ArrayList of Chest class, and we want to order it by its lock state first then by its number of content, in this case we should have already implemented comparable.

But as in "equal" methods not all classes should implement comparable, there is no need for FlashLightSwitch class to implement comparable, because it is not representing a real object, it's controlling the process of switching room light.

Following code shows the override of compareTo method in Chest class, first we check if the parameter is null or not, then sort by the lock state of the chest, then by the number of content.

```
@Override
  public int compareTo(Chest o) {
    if(o==null){
      throw new NullPointerException();
    }
    int result = Boolean.compare(isLooked,o.isLooked);
    if(result!=0){
      return result;
    }
   return Integer.compare(itemsCart.numberOfItems(),o.itemsCart.numberOfItems());
  }
```

ITEM 15: MINIMIZE THE ACCESSIBILITY OF CLASSES AND MEMBERS

In Trad class we have a number of protected fields and methods, they are protected to allow the subclasses like Buy and Sale to access these fields and methods, but they are not public to prevent access to them from outside the package. Following code shows protected methods and it's doc comments, we may ask, why protected method have doc comment?

```
  /**
   * To check if the item cart is empty or not.
   *
   * @param itemsCart the items cart to be checked
   * @return the true in case the itemCart contain items,otherwise false.
   */
  protected boolean isItemCartContainsItems(ItemsCart itemsCart) {
    return itemsCart.numberOfItems() > 0;
  }
```

"Item 56: Write doc comments for all exposed API elements" says all exposed API elements, and the protected method here is exposed for the subclasses of the Trade class. By doing this, we have applied a part of the suggestions this item talks about.

ITEM 16: IN PUBLIC CLASSES, USE ACCESSOR METHODS, NOT PUBLIC FIELDS

I created a Pair class as a data structure that couples together a pair of values, first time I implemented it with an exposed data field, but as this item says “public classes should never expose mutable fields.” the fields became private fields, and they are accessible using the getter and setter.

ITEM 17: MINIMIZE MUTABILITY

This item is full of shocks, **“Make all fields final”**, I think this is like someone says “Make all the websites static”, but after that the book says, **“unless there’s a good reason to do otherwise”**, the good reason here is that the player needs to move, so the fields that represent his/her position needs to change, we have seller and player and each one of them has ItemCart, and this item cart needs to change when any trade operation happen between them, and we may have other good reasons to say, but in contrast i found that there are a lot of fields that have to be final, and that is one of the shocks this item caused.
Each chest and door has a key to unlock it, i think this key field should be final, flashlight price, key name, key price, and wall have direction, all of these should be final.

“Make all fields private”, this one maybe the easiest to follow, actually that is what i thought in beginning, but that doesn’t mean adding private access modifier is enough, let’s look at this scenario, we have the following line of code:

```
private ItemsCart<Item> itemsCart;
```

Seems good, it is private, but in the same class we have following method:

```
public ItemsCart getItemCart() {
    return itemsCart;  }
```

That means, private almost means nothing, because we are giving the client reference to this object and that was the second shock, we should **“Ensure exclusive access to any mutable components.”**, all what we have to do is make defensive copies as item 50 says.

ITEM 26: DON’T USE RAW TYPES

After I finished reading this item, I took a moment to think, did I use raw type or not in my code? I found that in three places I used the raw type and in other places I did not use it, without knowing why I used either of them, following code shows one of the example of using parameterized type:

```
private List<Item> itemCart = new ArrayList<Item>();
```

Let's say we used raw type in this code instead of parameterized, and somehow we did pass String to be added to the list, everything will be okay until we try to fetch the element, we will get error, and this error will be on the run time, and most likely will be distant from the code cause the error, and that will make finding it harder.

ITEM 28: PREFER LISTS TO ARRAYS

Does that mean "array" is evil, and we should always use "List"? I agree that most of the time we need to use a list instead of an array, because it is better in adding and removing since it is resizable, but what if we want a fixed array of elements, just to print a list of options as in the assignment solution, is it better to use a list? I don't think so, first because we will not benefit of list features, second all we want to do is fetch the elements to print them for the user , so it is pretty obvious that array[0] is faster than array.get(0), as the later internally does the same call, but adds the overhead for the function call plus additional checks.
For example, here I don't see the need to replace the array with List.

```
String[] directions = {"north", "east", "south", "west"};
String[] navigationCommand = {"Left", "Right", "Forward", "Backward",
"Player status"};
```

ITEM 29: FAVOR GENERIC TYPES

Do I have classes that could be generify, I mean should be generated, in my solution I have a class called “ItemCart”, it works as the shopping carts, it stores the items, like flashlights and keys. Following code represents the class:

```
public class ItemsCart implements Comparable<ItemsCart>, Iterable<Item> {

  private List<Item> itemCart = new ArrayList<Item>();

 // Omitted ……..
}
```

So how can I determine if this class should be generated or not? Till now our cart can store only objects of subclasses that implement an Item interface, but what if we added a new thing to the game, that player can acquire, or if we had a new type of furniture that contains elements that can’t implement Item interface? Are we going to create new classes similar to "ItemCart" class but with different parameters? No, we can make ItemCart generic class.
So the class will be as follow:

```
public class ItemsCart<E> implements Comparable<ItemsCart>, Iterable<E> {

  private List<E> itemCart = new ArrayList<E>();
  public ItemsCart() {}
  public ItemsCart(List<E> itemCart) {
    this.itemCart = itemCart;
  }

  public List getList() {
    return itemCart;
  }

  public void addItem(E item) {
    itemCart.add(item);
  }
// Omitted -----
```

ITEM 49: CHECK PARAMETERS FOR VALIDITY

In readMenuChoice method in PlayerConsole class, i am just checking if the input is not integer, but what if it is null, should we check for null exception too, actually i think no need for that, let's treat "null" as string, and then it will be under mismatch input exception, but what about other methods in the solution, shouldn't we check for null exception for every method and every parameter, i don't think so, if so , then why do we follow the item54 that says don't return null, that is one of the reasons why i think it is not necessary in this assignment solution to check for null everywhere, and the other reason is that we check all the users input if it is null or not suitable it will be handled before using it, so here we save us some clutter in the code, because that is what comments do most of the time.
That is good, now we know that the input user doesn't mismatch the required data type, but what if the input out of the required ranger, what if we have ten choices, and the user input was 12, i tell three scenarios here, either we ignore this mistake, by doing nothing, without user message to tell him that his input is wrong, but i think this is not good choice, leaving the user without any response, or the second scenario is to throw IndexOutOfBoundsException, but is it worth it to terminate the program because this mistake, the third scenario is to tell the user that his input is wrong, and giving him another choice.

And after all of that, we should document what we did, using javadoc.

ITEM 50: MAKE DEFENSIVE COPIES WHEN NEEDED

It was hard to find out if there is need to apply this item on the code or not, and to make the decision easier, I found that if the answer of following question is "Yes" that means there is high possibility that this item needs to be applied, "Is your code returns or receive reference type of private field?".
And my answers was "Yes" because i found this method on Player Class:

```
public ItemsCart getItemCart() {
    return itemsCart;
  }
```

So we are returning an object, and it's private. And in other classes we receive objects then we initialize private fields by them.

```
  public Mirror(ItemsCart itemsCart) {
    this.itemsCart = itemsCart;  }
```

To solve this problem, simply we have to make defensive copies, like the item says. For example:

```
public Chest(Key key, boolean isLooked, ItemsCart itemsCart) {
    this.key = new Key(key.getItemName(),key.getPrice());
   // omitted ...
  }
```

A point worth mentioning, We don't have to do that with string, because string is immutable, it will create a new object on it's own.

Below code illustrate that:

```
public EmptyWall(String color) {
  this.color = color;  // Equivalent to "this.color = new String(color);"
}
```

ITEM 51: DESIGN METHOD SIGNATURES CAREFULLY

This item cover all of part of design method:

- **Choose method names carefully**, but what carefully means?
  Let's just look at this method name : public boolean isFlashLightOn()
  Is it well named? How can we tell? If the name of the method can tell us what its responsibility, that means most likely the name is good.
- **Avoid long parameter lists,** here we can see the differences about what Joshua Bloch and Robert Martin, Joshua Bloch says that four parameters or less is acceptable, bue Robert Martin says that more than two parameters or less is acceptable, three arguments are significantly harder to understand than two. but more than that we should find a way to reduce the number of the parameters, like "Argument Objects", anyway, in my code the maximum number of parameters is three, and that number is almost acceptable by both of them.

- **For parameter types, favor interfaces over classes.**(It will be discussed in item 64, later on).
- **Prefer two-element enum types to boolean parameters,** boolean types are not used as parameters in the solution.

ITEM 54: RETURN EMPTY COLLECTIONS OR ARRAYS, NOT NULLS

To allow the user to use key to open door, i needed to get List that contains all the keys that player acquired and ask the user to choice one of them to use, but the problem was, what if the player doesn't have any key, first idea came to my mind is to return "null", even though that doesn't seem right, so i added comment to reminded me to handle this problem better later, and the solution was in this item.

```
public ItemsCart getKeysList() {
    ItemsCart itemsCart= new ItemsCart();
    for (int i = 0; i < numberOfItems(); i++) {
      if (getItem(i) instanceof Key) {
        itemsCart.addItem(getItem(i));
      }
    }
    return itemsCart;
  }
```

But this item talks about methods that its return type is Collection or array, but what if the return type is an object? almost the same, we should return object represent "null", it could be something like this:

```
return new Key("Null key", 0);
```

ITEM 56: WRITE DOC COMMENTS FOR ALL EXPOSED API ELEMENTS

This item brought a lot of questions, Are all public methods considered public APIs?, isn't that going to make the code clutter and less readable? What about the protected members of base classes that have subclasses, aren't they API?

There was no clear answer to the first question, both of the answers "YES" and "NO" were present. I think "No" is the neary to the right answer, sometimes we could have public methods but they are not for world to use, they are for internal use, and we have total control on the clients that use it, and that leave us with the API methods, that is announced and published to the world to use.

For the second equation, it doesn't matter, we have to use doc comments for APIs. and yes if we provide base class for client code to subclass, then its protected members are also "exposed API elements", and i think we have to use doc comments with them.
Below code shows an example of using doc comment with an API.

```
/**
  * Returns true if the player has enough money to buy the item otherwise returns false
  *
  * @param item, the item to buy
  * @return boolean depends on the player amount of money
  */
 private boolean checkItemPrice(Item item) {
   if (player.getAmountOfGold() >= item.getPrice()) {
     return true;
   } else {
     return false;
   }
 }
```

*Note: I decided to not document all the APIs manually right now, because that will take a lot of time, and i need that time to cover as much as i can of the items, and i will document them manually later, and i hope "later" here doesn't mean never as clean code book says.

ITEM 57: MINIMIZE THE SCOPE OF LOCAL VARIABLES

This item have two points to be applied:

- Minimizing the scope of a local variable is to declare it where it is first used.
- Nearly every local variable declaration should contain an initializer.

If we look at any one of the overridden compareTo method, we will see this item applied. We could have done as the code below(Declare result in wrong time):

```
@Override
 public int compareTo(Key o) {
  int result;
  if(o==null){
     throw new NullPointerException();
   }
   result = Integer.compare(price,o.price);  // Line 7
   if(result!=0){
     return result;
   }
   return keyName.compareToIgnoreCase(o.keyName);
 }}
```

But that will cause us:

- Clutter in the code.
- Distract the reader.
- Not initialized local variable.

The solution is just postpone the declaration until we need the result variable to be initialized, at line number 7.

ITEM 58: PREFER FOR-EACH LOOPS TO TRADITIONAL FOR LOOPS

Even though I didn't need to use nested loops, and the most significant advantage of using foreach is that it could avoid us some tricky bugs when using nested loops, I used it because it saves the code some clutter.

ITEM 60: AVOID FLOAT AND DOUBLE IF EXACT ANSWERS ARE REQUIRED

The item says explicitly, that double and float are ill-suited to monetary calculations, but let's say that point does not exist, do we really need double and float in our case! we have the amount of gold that player has, the prices of each item, and the seller's price list. I don't see why any of these would be float numbers, actually I don't remember any game that I played that had a float price for anything inside it.

ITEM 61: PREFER PRIMITIVE TYPES TO BOXED PRIMITIVES

If we look at each class, we will see that all the fields that could be represented by primitive types, are represented by primitive types, because that save us:

- Wrong comparator results.
- From throw NullPointerException,when the boxed primitives are not initialized.
- performance problems.

ITEM 62: AVOID STRINGS WHERE OTHER TYPES ARE MORE APPROPRIATE

Three fields in Player class are primitive integer type,two of them represent the x,y-position and the third one is the amount of gold, these could be represented by Strings, and each time we want to modify them, we parse them into primitive integer type and modify the values then store them again into string, this approach could be reasonable in first, but we know that string is expensive, and this item gives us other reasons to not use string when other types are more appropriate.

ITEM 63: BEWARE THE PERFORMANCE OF STRING CONCATENATION

We have a lot of string in the assignment, item names like flashlight, bag of money, and other items, the same for the room furniture like chest, mirror and other furnitures, but that doesn't mean we should use StringBuilder in place of a String to avoid performance issues, because we don’t concatenate these string, we don’t perform any operation on them except comparison, so in this case i think we should go with String class.

ITEM 64: REFER TO OBJECTS BY THEIR INTERFACES

It makes sense for us to ask, why? Let’s see this line of code from the world navigator assignment,

```
private List<Item> itemCart = new ArrayList<Item>();
```

When i wrote this line, i did that before i know about this item, the reason i wrote it like that, because i wasn’t sure if using ArrayList is the better choose, i thought i may change the container to LinkedList, so i used List interface as data type, because it will give me more flexibility to change it later if it turns out that ArrayList is not the best choose, and that is exactly what this item talk about.

Another great usage of this item, when used with parameters, it will give us the flexibility to change the argument that we sent to any other type from the interface’s child.

Reason of why some items are not here:

- Not applicable here, in this assignment, for example
  Item 66: Use native methods judiciously,i don’t see a need for native methods here .
- Some items talk about things mentioned in the clean code book.
- Some items I didn't have time to apply them.
- And others I couldn’t fight out how to use them.

DESIGN PATTERNS

In furniture interface we have six subclasses that implement the Furniture interface, they are not exactly the same, some furnitures have three fields, and others have two fields, the point here is

that we may have complex object to be constructed, so first i thought the best way to simplify the construction operation of these classes is to use Factory Method Pattern, but the problem with this patten was that the concrete objects take different constructor parameters, so it seems this pattern is not an option, so i decided to use Builder pattern, because it solve the different constructor parameters problem.

Below code represents part of the builder class.

```
public class FurnitureBuilder {

  private boolean isLocked;
  private boolean isOpen;
  private String wallColor;
  private ItemsCart<Item> itemsCart = new ItemsCart();
  private ItemsCart<Item> itemPrices = new ItemsCart();
  private Key key;

  public FurnitureBuilder key(Key key) {
    this.key = key;
    return this;
  }

  public FurnitureBuilder itemsPrices(ItemsCart itemsCart) {
    this.itemPrices = itemsCart;
    return this;
  }

  public FurnitureBuilder itemsCart(ItemsCart itemsCart) {
    this.itemsCart = itemsCart;
    return this;
  }

  public FurnitureBuilder open(boolean isOpen) {
    this.isOpen = isOpen;
    return this;
  }
 // Omitted ..........................
```

DESIGN PRINCIPLES

SRP: THE SINGLE RESPONSIBILITY PRINCIPLE

I think the best way to illustrate how SRP is applied to the assignment solution is to walk through the process of changing a class that violates SRP to a class that follows SRP.
Let's take Player Class from the beginning, if we asked ourself, how many reasons could change this class?
1- Player class was responsible for navigation, but what if we wanted to add a new navigate command, like "Move Right", that means we should change the navigation method, so in this case it is better to create a navigation class responsible for the navigation commands.

2- And to not make the same mistake again and creating class with more than one responsibility, let's see what navigation would have as responsibilities, it has to change the orientation and to move from room to another, and this seems more than one responsibility for me, so let's divide the problem more, what about two classes, "PlayerMovement" Class to handle the movement from room to another, and "PlayerDirectionRotate" to handle the orientation rotation, in this case each one of these classes has single responsibility, and by doing that we moved two resposibilites from the "Player" class.

3- Player class was responsible for look command, but what if we changed the process of looking, it means Player class should change and that is the second reason to change Player class, so Look command should have its own class and any changes would be on that class.

4 - Player Class was responsible for the Check command, but we could change the why we want to check the elements, as if we want to decide which element to acquire, so the check command should be moved to a separated class with a single responsible, and its responsibility is to check the opposite furniture, and till now we got rid of three responsibilities there were in the Player class.

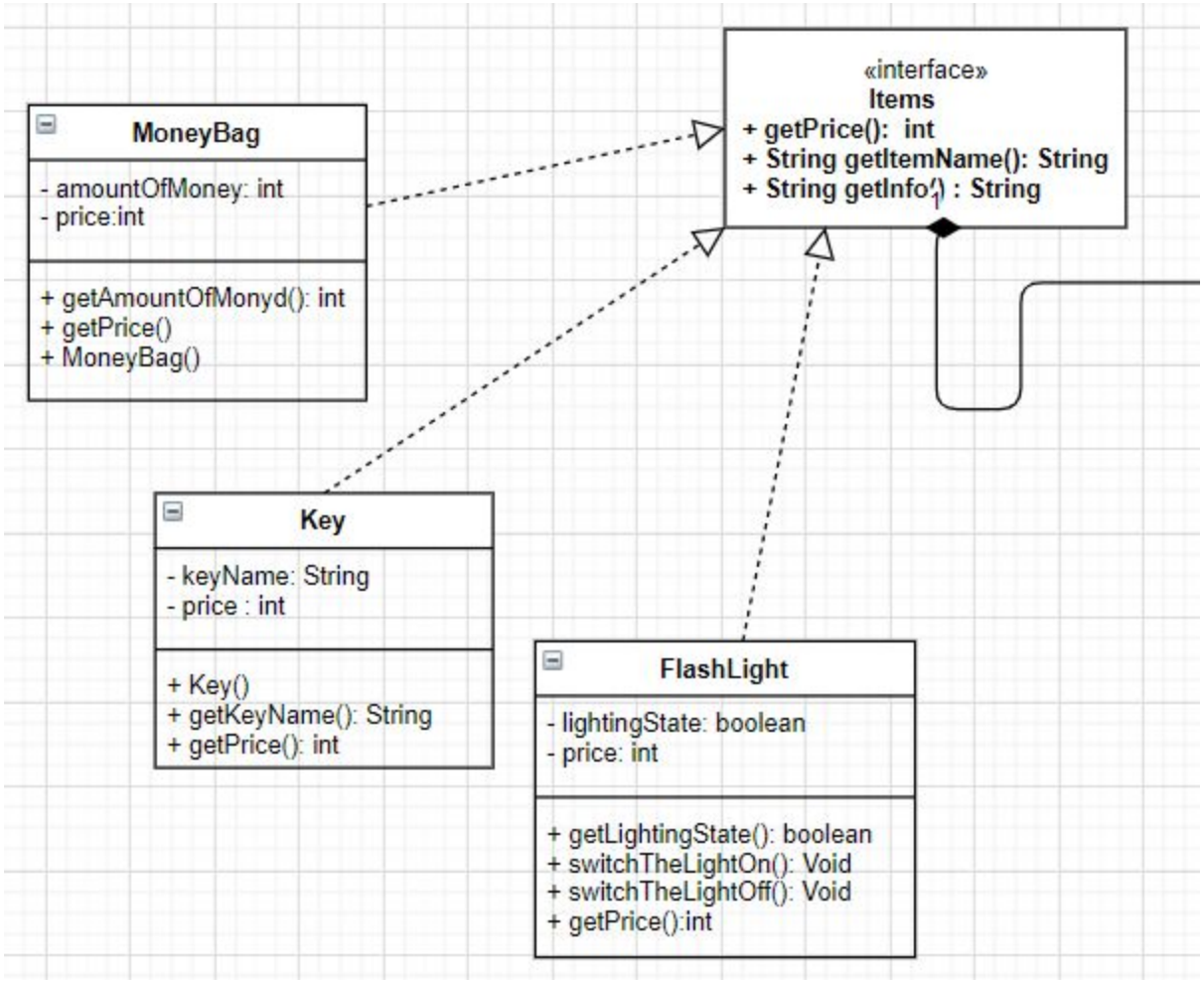
5 - I think if we said **"Open doors"** is a player responsibility that would make sense, but also move from room to another, sell, buy, and check. But that is not enough to put them all in the same class, because the SRP is not about what you do, or what you can do, but also should apple "One reason to change" rule, and it makes no harm to move all the previous commands to separated classes, actually it make the code reusable and mentinalbe so again **OPEN** command should be in separate class.

OCP: THE OPEN-CLOSED PRINCIPLE

Let’s think that we are in one of the rooms from the maze, this room could have four “thing” let’s say Mirror, Chest, Door and Seller, and we want to use the look command to know what is the opposite furniture, that means we should have a class that consist of five methods with different parameters(overloaded methods) to return the name of the opposite furniture, first problem here is the redundancy, we wrote the same code four times but with different parameters type. but that is not the only problem, what if we want to add a new type of furniture?!, let’s say now we have a drawer, that means we should open the Look class, modify it and add new method to handle the new furniture, and this violates the OCP, so what i did is create a Furniture interface and each type of furniture implement this interface, and the Look class will deal with the furniture using the polymorphism, and by doing that we can add new furniture as much as we want and there is no need to modify the Look class.

LSP: THE LISKOV SUBSTITUTION PRINCIPLE

This principe shown clearly when we deal the items we acquire, either by buying them or by looteing them from the chest, but these item not spertated, the are grouped using data structure, and to do that we should follow the LSP, below figure shows the the interface the items implement.



And using “List” of the “Item” interface we are able to use and substitute the items.

```
private List<Item> itemCart = new ArrayList<Item>();
```

ISP: THE INTERFACE SEGREGATION PRINCIPLE

Again, I think the best way to illustrate how ISP is applied in the world navigator assignment is to walk through the evaluation of the Furniture hierarchy, if we looked at Chest, Door, Painting, Mirror, empty wall and even seller they are all furniture construct rooms in the maze and share common behavior, that behavior is to tell the player what he see. Cool, let's rush now and write the code, and after writing the code we will realize that Chest, Door, Painting and Mirror are checkable furniture, also Chest and Door are lookable furniture, okay eazy, let's add these behaviors to the interface and force the subclasses to implement them(Figure 4-2) and now we are violating the ISP by enforcing some classes to implement methods they don't need.

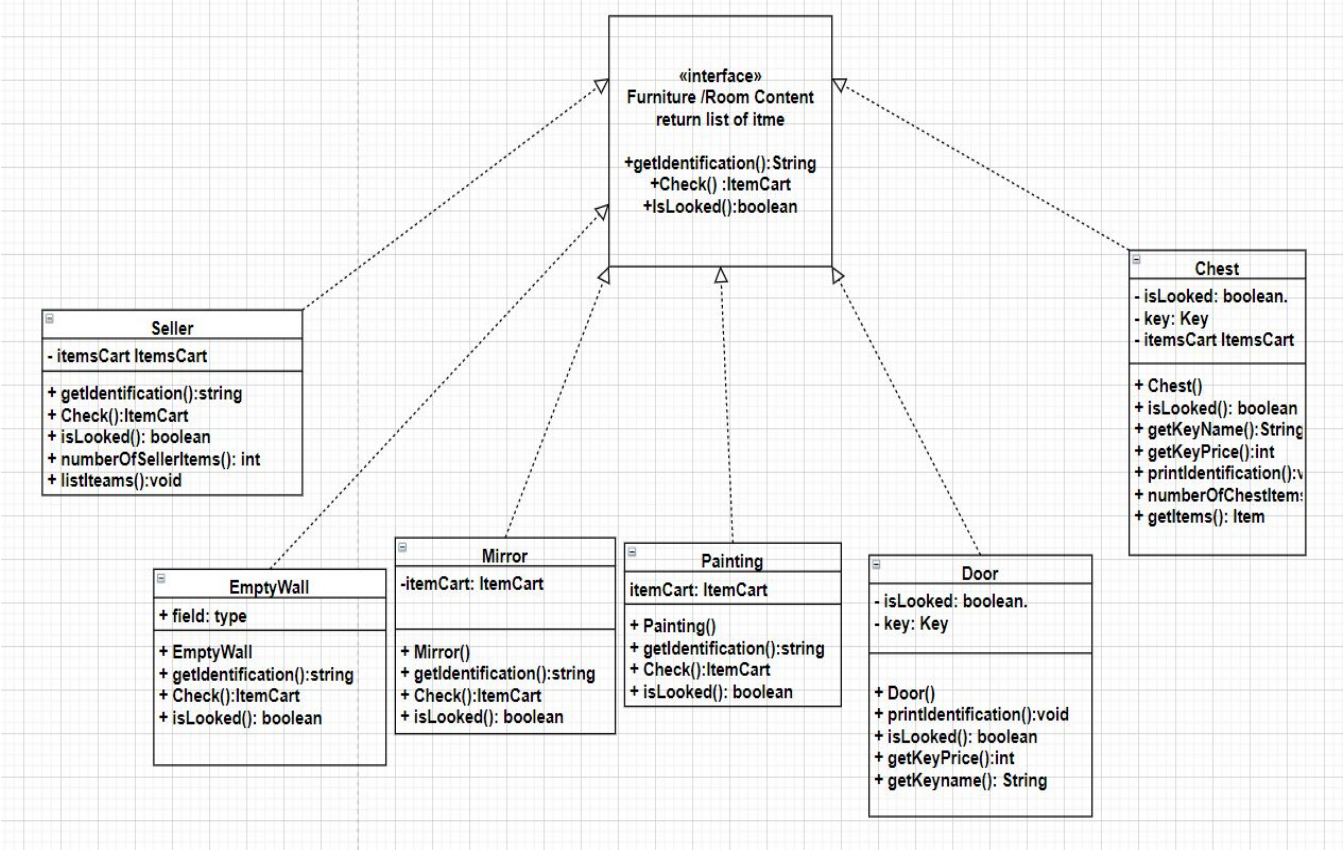
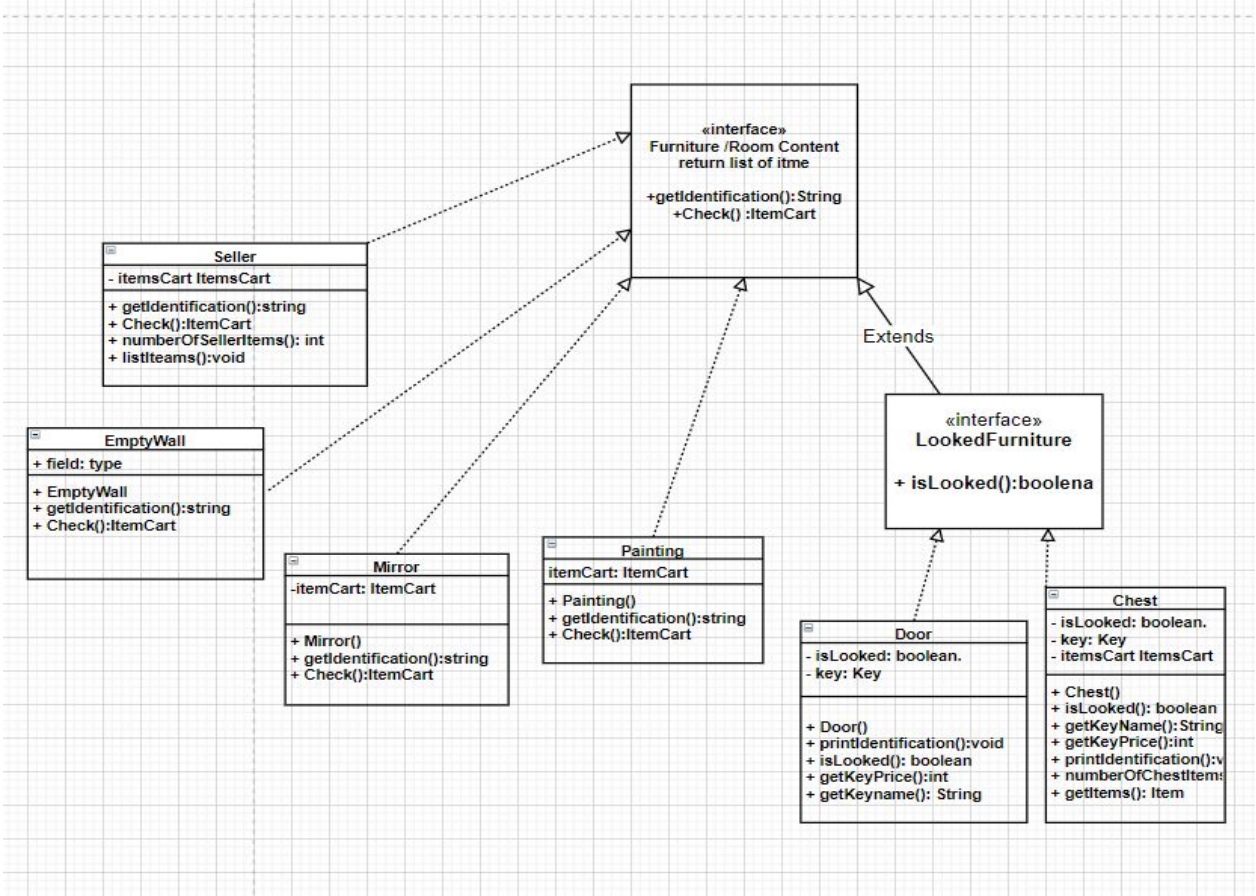
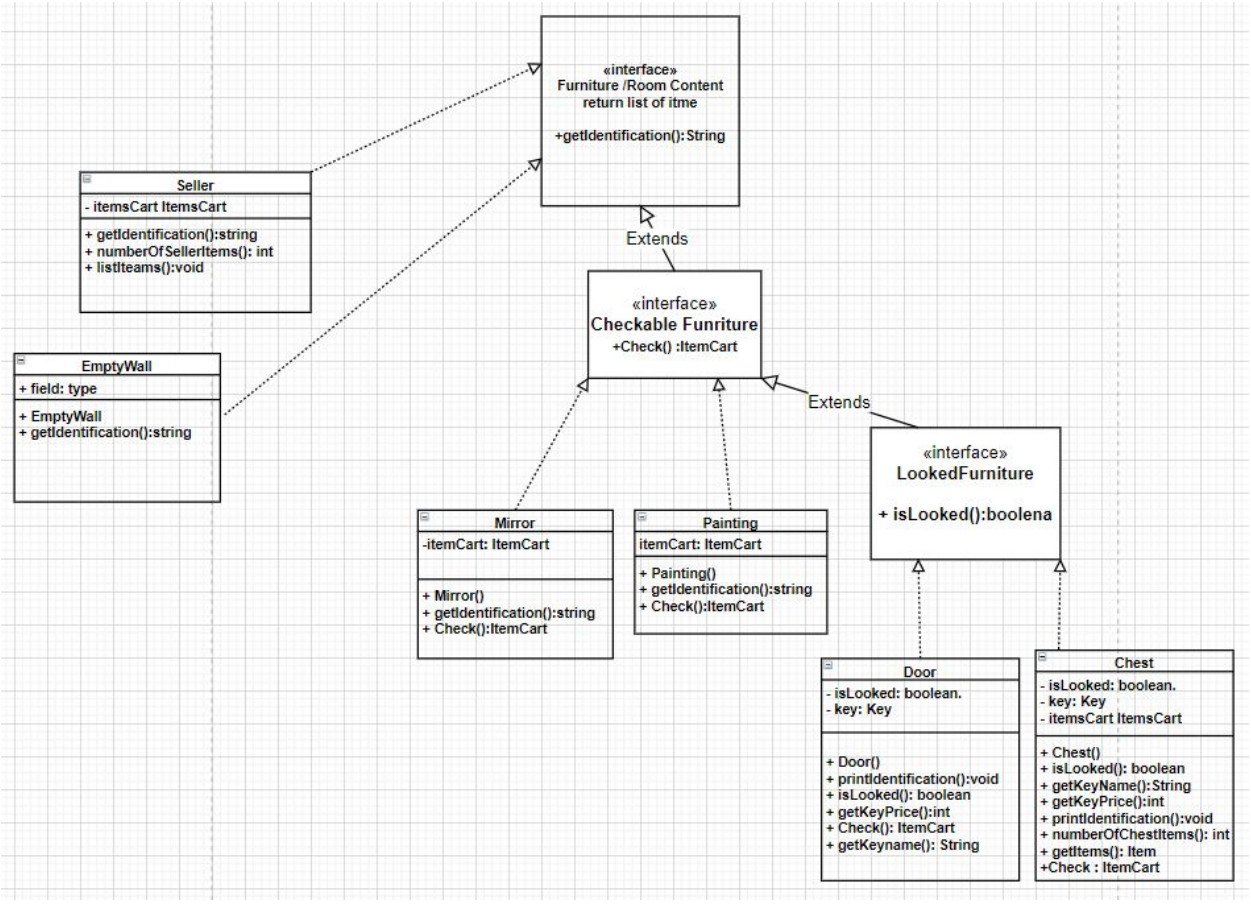


Figure 4-2

So here we can see that all of EmptyWall, Mirror, Painting and Seller have to implement isLooked method, but actually they are not lookable furniture, so using the solution that ISP provide, the hierarchy will look like this(Figure 4-3):



But still, are all the furniture checkable?! no actually, Seller and EmptyWall aren't, so we still violate the ISP by force them to implement a check method, let's divide the problem again, create another interface that represents the checkable furniture, and the final UML will look like in the (Figure 4-4) below.



Now it appears that the hierarchy follows the ISP.

Note: There is a question that could be asked here, Is "seller" furniture?! When we say we want to design our project using oop, that means we want to divide our project into objects that represent project components, and if my project have Person as object, it's most likely will not implement or inherit for example Car Interface/Class, so why do we deal with "Seller Class" as Furniture ?
Actually because "It" is. When we want to apply the abstraction principle, in these circumstances the Seller is nothing but a vendor, and as the abstraction definition says : "Abstraction breaks a concept down into a simplified description that ignores unimportant details and emphasizes the essentials needed for the concept, **within some context.**"
And in this context the name, the age and other information about the seller are not important, what we care about here is the items that he has, so we are dealing with a vendor object called seller.

Google Java Style Guide

Source file basics

File name

Each source file name has the exact name of the top-level class it contains plus .java extension
For example the name of the file that contains Chest class is Chest.java.

Source file structure

Class declaration

Exactly one top-level class declaration

Each top-level class resides in a source file of its own.

Ordering of class contents

A source file consists of, **in order,** as below code show :

```
/*
 * World Navigator - file Seller.java
 *                                                    License or copyright information
 * copyright (c) 2020 - Muntaser A. Abukhadijah
 *
 */
package com.company;                                  Package statement

import java.util.Map;                                 Import statements
import java.util.HashMap;

public class Seller implements Furniture {            Exactly one top-level class
......
}
```

Formatting

Braces

Braces are used where optional

As google java style guide instructs to use the braces each time this option is available even when there is just one line of code, and that could save us some bugs,for example if we wanted to add new code to that specific one line if statement, we may forget to add the braces and that Mostly will cause logical error.
Below code illustrates why adding braces could save us some bugs.

```
int x=5;
if(x == 5 )
   System.out.println("Right");
   StartAgain();//Added after time.It should be run just when x equals 5
 for(int i=0;i<10;i++){
     System.out.println("");
 ……….
```

Nonempty blocks: K & R style

To illustrate the usage of K & R style in World navigator assignment let's see following code with some notes:

```
private void setColumnPosition(String direction) {
  if (direction.toLowerCase().equals("east")) {
    columnPosition = columnPosition + 1;
  } else if (direction.toLowerCase().equals("west")) {
    columnPosition = columnPosition - 1;
  }
}
```

- If we looked at the red brace there is no line break before it because it is the opening brace.
- The next statement after the opening brace is on new line.

- The closing brace should be on new line, that means there is line break before the closing brace.
- But the pink barac does not follow the the third point!, that is because third point applied only if that brace terminates a statement or terminates the body of a method, and in this cast it's not.

Block indentation: +2 spaces

As the white space in the previous figure shows, each new block starts after increasing two spaces.

One statement per line

To follow this rule each statement should be in a separate line, as we see in the previous code (orange highlight), even though that the "if" statement consists of one line, it's on a separate line.

Column limit: 100

The longest line consisted of 94 columns.

```
System.out.println("Your position is : " + getRowPosition() + " " + getColumnPosition());
```

And even if there is longer than that, they are going to be wrapped.

Line-wrapping (It should be reviewed, because of the number of parameters)

```
  private boolean switchLight;
  private boolean lightState;
/                                                                              /
public Room(
      int rowPosition,
      int columnPosition,
      List<Wall> walls,
      boolean switchLight,
      boolean lightState) {
    this.rowPosition = rowPosition;
    this.columnPosition = columnPosition;
    this.walls = walls;
    this.switchLight = switchLight;
    this.lightState = lightState;
  }
```

Whitespace

Vertical Whitespace

Blank line improves readability of the code and blank lines should be added between consecutive members or initializers of class and there are some exceptions, as we see in the last figer the white highlighted line represents one of the proper places to add a blank line.

But we don't see a blank line between the members as the google guide says!, that is because this is one of the exceptions, when we have two consecutive fields (having no other code between them) blank line becomes optional.

Horizontal whitespace

Horizontal whitespaces are limited by only 9 places, and one of them is optional, and figure below shows 6 out of 9 places that should have white space to follow the google java guide style, the dark green represent two places actually, after the closing parenthesis and before the opening curly brace.

```
String[] directions = {"north", "east", "south", "west"};

private ArrayList<ArrayList<Room>> map;

private void setColumnPosition(String direction) {
    if (direction.toLowerCase().equals("east")) {
      columnPosition = columnPosition + 1;
    } else if (direction.toLowerCase().equals("west")) {
      columnPosition = columnPosition - 1;
    }
  }
```

Horizontal alignment: never required

I think programmers who do align their code horizontally, they do that because it will look good but **there is a difference between it looks good and reads good, and what matters is reads good.**

Variable declarations

One variable per declaration

Every variable declaration (field or local) declares only one variable, this rule is applied on the whole assignment code even when there are multiple variables of the same type.

Declared when needed

When we minimize the scope of the variable that could save us some problems, like declaring another variable of the same name, for instance “index” name is widely used as variable name, so if “index” variable needed should be declared in the scope that needed in,

```
private String getNextDirection(String direction) {
   String[] directions = {"north", "east", "south", "west"};
   String currentDirection = getDirection();
   int index = 0;
   for (int i = 0; i < 4; i++) {
   if (currentDirection.toLowerCase().equals(directions[i].toLowerCase()))
     {
       index = i;
       break;
     }
   }
```

So in that case we could use “index” as a variable name anywhere else in the code when needed.

Arrays

Array initializers: can be "block-like"

Google Java Style Guide suggests four formats that the array could look like, but also says these formats are not exhaustive and if we checked google-java-format plugin it will format the array as one line with its initializer and this format is the used one in the code as shown in the above code.

Switch statements

Indentation

Twoindentation rule to be applied with switch statement

- the contents of a switch block are indented **+2**.
- After a switch label, there is a line break, and the indentation level is increased **+2**

```
switch (choice) {
  case 1:/*                                                    */
    new Buy(player, seller).performTrade();
    return new Buy(player, seller);
  case 2:
    new Sell(player, seller).performTrade();
    return new Buy(player, seller);
  case 3:
    seller.getItemCart().getList();
    return new Buy(player, seller);
  default:
    return new Sell(player, seller);
  }
```

The default case is present

In all the places that I used switch statements I used default and it contained code, but even if there is no code it should be present.

Annotations

Rules to be applied with Annotations:

- One annotation per line
- Indentation level is not increased

And these rules are applied as it appears in the code, for instance:

```
@Override
```

```
public String getKeyName() {
  return key.getItemName();
}
```

Modifiers

Modifiers should be ordered as the order recommended by the Java Language Specification, and in this case we have public and protected modifiers, and public should be first then protected.

```
public static String[] options = {"Buy", "Sell", "List", "Finish"};
protected Player player;
protected Seller seller;
protected int itemId;
```

Naming

Rules common to all identifiers

- prefixes or suffixes are not used in Google Style, so no name will be like: player_1 or cFurniture.

Rules by identifier type

Package names

Rules to follow when naming packages :

- Lowercase
- consecutive words
- no underscores

For instance :

```
package com.company;
```

Class names

Rules to follow when naming classes:

- names are written in UpperCamelCase.
- Class names are typically nouns or noun phrases.

Some classes names from the assignment :

- Chest - Door - FlashLight - Item - Player - Seller - Mirror

And all the classes in the assignment are following these rules, there is a class called “Buy” and buy could be noun as in “That jacket was a really good buy”.

Method names

Rules to follow when naming methods:

- Method names are written in lowerCamelCase.
- verbs or verb phrases

Some methods names from the assignment :

- makeTrade - unLock - getItem - getRowPosition - readMenuChoice

Parameter names

Rules to follow when naming parameters:

- Parameter names are written in lowerCamelCase.
- Parameter names should not be one character, because it is hard to tell the meaning of the parameter from one character.

Some methods names from the assignment :

- prompt - menu - message - direction - keyId

Programming Practices

@Override: always used

Two points here why i use `@Override` every time I override a method,

1- It makes the compiler check if the method overridden correctly, without any mistake of misspelling a method name or number of type of parameters.

2- The main point of coding style is to make the code readable, and adding `@Override` notation makes the code easier to understand.

Data structures

Let's list the objects that use data structures and the frequency of there usage,

	Accessing specific element	Add	Remove	Iterate
Player	High	High	Mid	High
Seller	High	Mid	Mid	High
Chest	Low	Never	High	Once
Painting	Low	Never	High	Once
Mirror	Low	Never	High	Once

Based on this table I decided to use ArrayList to store the items for each object that needs to store items.

We acquire all the elements from Chest, Painting, and mirror just once, so the only operation the could cause us some problems is removing each item after acquire it, but we could solve this by either acquire all the items first then remove all the items by initialize the ArrayList again, or another approach is to iterate on the ArrayList backward, and remove item by item, this is O(1) oberation.

The only high cost is when we want to remove items from player or seller data structure, remove and add the seller itemCart will not be frequently, but for the player will be.

I choose ArrayList over Linked list, because there will be a lot of fitching items so it will be more efficient to do that with ArrayList.

But what about HashMap, isn't it better than both of ArrayList and linked list to use?

I decided not to use Hashmap because some items could have the same name, but with different prices.