Coding Report

Name: Issa Anagreh
Project: World Navigator

◆ Clean Code

- Naming:

- Constants were refactored to be all uppercase.
- Some methods naming were forced to be used to match the requirements naming.
- No abbreviations were added to the project to achieve the highest understanding.
- IDE spelling check.
- The naming of methods were almost as uncle bob recommended by starting with a verb then a noun.
- Boolean variable naming always starts with is; if variable it's isValue, and if final value IS_VALUE.

- Constructors:

- Multi constructors chains are being used for multiple reasons:
    - Classes are made to separate responsibilities but are never to be instantiated unless a previous constructor is called.
    - Some classes have a private generating method which is called in the constructor to restrict the generating process.

- Methods:

- Nulls are never returned.
- Special codes are used as parameters in some methods such in trading; let me talk about an example -it is found in the trading class-, I have a method that

expects an integer to choose from a list that are ID's enumerated, but users might want to exit the list without choosing so I made a decision of using -1 which is an impossible ID.

- The maximum number of parameters used is 3.
- Flag arguments are used once; it's restricted by the requirements so that one command can switch the lights of the room on or off, so I saw that to have a method for one requirement is better than having two.
- Methods were strictly sized less than 10 lines of code.

- Values:

- Magic numbers are never used, constants have been used in the GameTimer to convert milliseconds to seconds.
- Booleans are named depending on the usage, so no !isCorrect ever used.

- Failures and Exception handling:

- Made my best to not implement any failure and not to use try and catch, why; I assumed the client is the gamer of the game who doesn't like to see a crash or an error of the game, so what I have done is making notations when anything illegal or non-sensible, so the user can go on with the game all the way and redo any mistake they did.
- Returning is fast as possible, but unfortunately, some if-else's are added to achieve the previous point and avoid many user's mistakes, plus they give many hints.

- SOLID:

- SRP:
    - Maps factory and its next classes (rooms, walls, …etc.) have a good example of the single responsibility principle.
    - I tried to be as creative as possible; imagined Maps factory class, Game Menu class, Game MVC classes; PlayerModel that has all player's

details, PlayerController that allows user to control his player, PlayerViewer that can be the GUI once), Key class, GameTimer class, ContentsManager (this prepares items such as keys, flashlights and golds and so on), Room class, Wall class, Items classes (such as Painting class, Mirror class, Seller class, Door class, and Chest class), and the ItemsFactory class.

- PlayerController and PlayerModel are the most challenging classes; Player model is the model of the most engaging part of the game which is the player which interacts with mostly everything in the application -except some parts such as menus and maps factory-, so what I have seen is that splitting any part of this Class can increase the complexity as the model class must achieve its existence purpose which is to have all the information of the player. The Player controller class is a bit weird, usually, controllers are customizable so what I did is to make a Player controller master then extended to a custom player controller (PlayerController class). More of deeply digging inside of some classes are to be discussed later

- Coupling: unfortunately coupling frequently appears in the project, let's discuss, I have a MapFactory which prepares a map including its rooms, then each room prepares its walls, then each wall prepares its items (Door, Chest, Mirror, Painting or Seller). I made my best thinking of this chain but I lastly found that this is a normal correct chain, where there is no map without a room at least, and room without 4 walls as the requirements said, and no wall without items then these items are generated via ItemsFactory class that can be updated if any new item was to be added later.

- OCP:

  - New items (doors, chests, paintings, mirrors, and sellers) and contents (golds, flashLights, and keys) are possibly can be added to the game, contents and items are changeable too. So I implemented ItemsFactory and Content Manager that helps to achieve OCP principle to add new items as simple as possible with changing only these classes.

- Customized player controllers class is also extended from a master that has some certain controlling commands and methods that must not be changed can be over-ridden or added to.

- LSP:

- Honestly, this principle was the most helpful which used in my implementation of this project in one of the main scenarios; when having an Item class that is extended to all items such as door, chest, seller …etc. but still, each one has it’s own specific fundamentals so doors are openable but not checkable, but the chest is openable and checkable, and the mirror is checkable but not openable. So having this principle to implement this critical scenario was helpful by making Item, Locked_Checkable, Unlocked_Checkable, Uncheckable, Openable, CheckBehavior interface, and UseKeyBehavior interface, to deal with all possible scenarios with all the items and any future added items.

- ISP:

- No unused implemented methods and no unused forced implemented methods in the classes.

- DIP:

- Items are the most changeable parts in the whole project so it needed to be extremely isolated, that was the inspiration of implementing of DIP by adding the ItemsFactory so it only gives the chance to deal with it to invoke a certain method or to read a certain attribute from a certain item such as Door, Chest, Paintings …etc.

- Comments:

- Comments were used twice; the first usage was for a while loop that will take the commands all time as long as the game is on, while’s argument is player.isPlaying, the argument is clear enough but still I thought it may need more clarification, the second usage was about telling the developer about the timer starting method while its class's name is GameTimer and the run of the timer in its constructor.

- Style conventions:

- Google style guide has been used, an IntelliJ plugin is installed (google-java-format)

- Url reference check by me and mostly did not find any issue.

◆ Design Patterns

This game project has many related classes and so it needed many design patterns to achieve this principle, starting with the keys usage that is working for chests and doors, but chests are checkable (can acquire items from) but doors are not, mirrors and paintings are checkable but not openable with keys, and many other applications.

- Maps: originally maps are .json files that must be read and converted into java classes and objects somehow; "hide creation logic" is the decision of mine to be implemented here, depending on this, Factory Method Pattern was the best for this application, but actually I made a different implementation, no interfaces were made as it's not actually needed in my case because map items have different methods content and there is no need for a special interface for them in this part of the designation -they have a special interface but in different scope-.

Let us dig deeper here, .json files needed a special library installed that can convert the .json file to jsonOBJECTS, jsonARRAYS, and jsonVALUES, and these files must be refactored to be some java's classes and objects which are complicated to go through every time a developer wants to implement a new item -let us say a developer wants to add tables in the rooms-. So, MapFactory is to parse all the .json map to java map, even if a developer wants to add a new item -a table as an example- can add it in a simple way depending on the readme file and changing the ItemsFactory class -this will be discussed later-. However, the only reason to get into the MapFactory class is if the developer wants to add a new object to contents such as golds, flashLights …etc, -a hummer as an example new content object-.

- Items: items are such as paintings, mirrors, doors, sellers …etc. The implementation here mainly was about having multiple shared behaviors, but multiple other different behaviors are also there, so "Separate what varies from what remains the same" was the decision-maker of mine, depending on this, Strategy Design Pattern was the best choice for me here.
Let us dig deeper, logically items must have a superclass called Item because multiple behaviors are repetitive such as useKey, AcquireContent, and some other abstract methods and attributes, but as said before the behaviors are various as we may have locked chests that need keys to open them or unlocked chests that can be checked and acquire their contents directly. On the other hand, some Items are not checkable at all such as doors -doors do not have contents to be acquired- so here Unlocked_Checkable, Locked_Checkable, and Uncheckable classes were made and implemented CheckBehavior interface. Moreover, Some items are openable such as doors and chests, so they need a UseKeyBehavior interface to implement. Actually we have only one class that implements UseKeyBehavior as I took a decision of leaving this behavior in the unopenable items as null, but I am sure of using it later if we want to separate opening chests from acquiring their contents and this will create a new class and let us call it unlocked_openable when it needs no keys to open.

- Player: rarely I like to take a common decision like this but MVC Pattern here is the wisest decision to take I believe, we have a view that can be once a GUI, we have a data pool of the player that can be the model, and a controller which connects the view with the model mainly by the player commands.

- Player model: this can be debatable actually but from my point of view is that my PlayerModel class is implementing the Facade Pattern; this class has access on every playing class in the software, and what I mean by playing classes is that about classes which have a direct link with the player and the player's interactive objects such as doors, chest, sellers, rotation, move …etc. However, the player model class does not have access to Menu class as an example, but still, my point is valid; how, the menu is a high part of the game that has nothing related to the player (the virtual creation which seeks the golden door), as menu decides the map or restarting and quitting the game.

- Player controller master: PlayerControllerMaster is a class that I believe is one of the most that has a chance to be changed as each user might want to change the commands he uses, "forward" command is a long command actually when I tried my software it made me really bored, so I made up an extended PlayerController which is extended from the master implementing the Decorator Pattern and added "f" as a command to move forward with my virtual player in the game -by the way, I made an improvement to the game that the map has more of real 2D rooms the user can really move in, look at the readme-.

- Commands: as simple as this decision was to take this Command Design Pattern was the latest taken, why; because this pattern fills the command inside of a hash map with some more simple details, and the problem was with how to put a value as a method, thankfully this pattern helped me by making a command class in each key's value of the commands Hashmap.
Switch statement was a valid choice here, but it was really long to be acceptable, and this was the main reason for taking this pattern implementation.

- Keys: this is the most complex decision has been taken, keys must be a

special class that can open doors or chests but, in the future, more kind of keys or content items such as keys can open object items such as chests, so the decision of implementing Visitor Pattern was a must for me.
It gives me a wide number of choices when it comes to how to open things this is the main idea, plus this pattern can simplify the using key process even if having another item that can open objects is not possible.

◆ Interfaces

- CheckBehavior & UseKeyBehavior: are to implement the strategy design pattern.
- Command: is to implement the command pattern.
- KeyChecker: is to implement the visitor pattern.
- NextGoing: Openable class is not enough to express a going through items, such as doors or gates, actually it expresses about an item that uses key to be opened. So NextGoing is a mandatory interface for items such as doors or gates.
- PlayerControllerInterface: is to implement the Decorator pattern.

◆ Transitioning and Rotating

Transition and Rotate classes are actually optional that can be included to the responsibility of the player controller, but after implementing their logic I saw to separate them to special classes because their logic has bad looking switch cases.