Atypon Systems

World Navigator Final Project

Software Engineering and DevOps Bootcamp – spring 2020

Done By: Amal Taha

Date: 6-August-2020

Abstract

The purpose of this report is to represent the implementation and design details of the World Navigator Assignment, including a discussion on the clean code and SOLID principles according to Robert Martin, Effective Java principles according to Joshua Bloch, the Design Patterns used, Google Styling guide and the Data Structures and Algorithms presented in the assignment code.

Table of contents

Introduction

As a Software Developer, you’re asked to write a code that satisfies all the tests and conditions of the project you’re working on, you simply write the code with no bases, arbitrary names of variables, multiple functionalities for a method and just thousands of lines in one small class. You might reach a point where everything is implemented in one place with thousands of lines, assuming you want to run this code and many errors occur, will you be able to get through the code easily and find the errors then fix them? What if they were logical errors that are not easy to investigate? And even if you were able to run the code correctly that passes all test cases, yet when you need to modify it later you might need to change many lines of code and spend lots of time and effort. Now that’s you, the one who writes the code and modifies it, will get a job at a company where many people are supposed to read and modify your code from time to time, but this is not going to work because they won’t be able to understand the details without a reference and will take their time and efforts too.

But you’re now a Software Engineer who is supposed to write the same code but this time with many restrictions that will make your code flexible, readable and reusable with a high level of abstraction. This is the part where you will start thinking of a good design for your code such that you will apply appropriate design patterns that solves design issues and separate the low level from the high level details and such a client method can’t know much about the details of other methods implemented. You will make your code clear and easy to read with the clean code principles such that your code won’t smell bad. And will use the appropriate way of coding such that when working in larger companies everyone will be able to work on each other's code and finish the work faster with less effort and thus more productivity, and this is why a big company cares a lot about its employees as software engineers not only developers.

In the following assignment, we are asked to implement a maze with four-sided rooms that are connected together through doors between them. Each room side has one object that is Paint, Mirror, Chest, Door, Seller or just a Plain Wall. Each Room Object has a specific functionality, the mirror and paint might have keys the player can use, the Chest have objects inside the Player can take, but some chest will require a specific key to open. The Door is a way to move between rooms and might need a specific key to be able to pass through. Finally the Seller sells or buys objects from the Player and the exchange items. You can control the player movements through a collection of commands that are presented to you according to the current available movements. You need to reach the final room to win the game.

The Maze Game Design

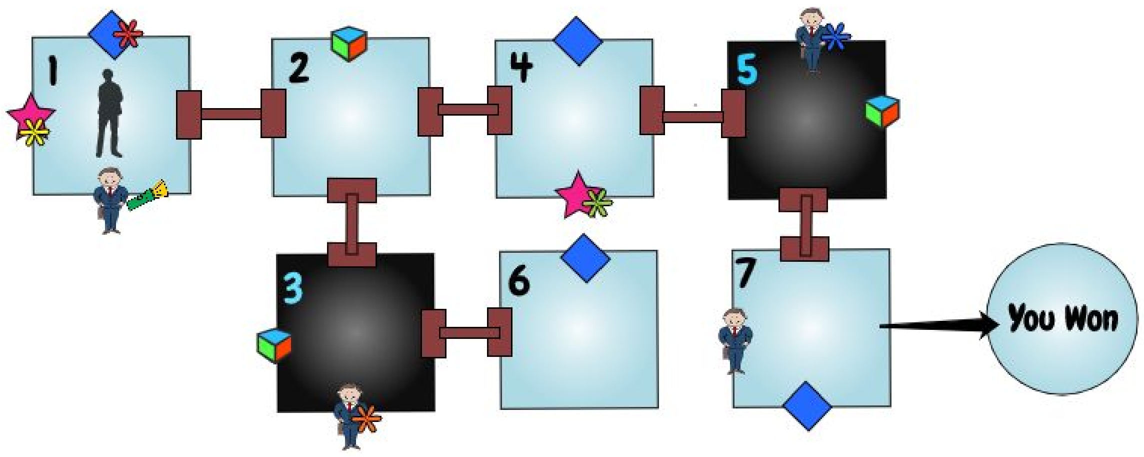


Figure1: The Design of the Maze

The design of the maze is based on making a graph such that the vertices are the rooms and the doors between them are the edges, the edges are always shared between vertices and thus a door between two rooms is the same door. It's somehow obvious from the shapes what each one represents, the gentleman represents a seller, the star represents a paint, the rhomboid represents a mirror, and the box represents a chest. The small flower shape represents a key that can be acquired at this room object and finally the light represents a flashlight the seller sells. Initially, the player is at room 1 and owns 100 maze points and can move on, when he faces a door it can be closed and asks for a specific key that should have been acquired before reaching this door. The commands are as follows:

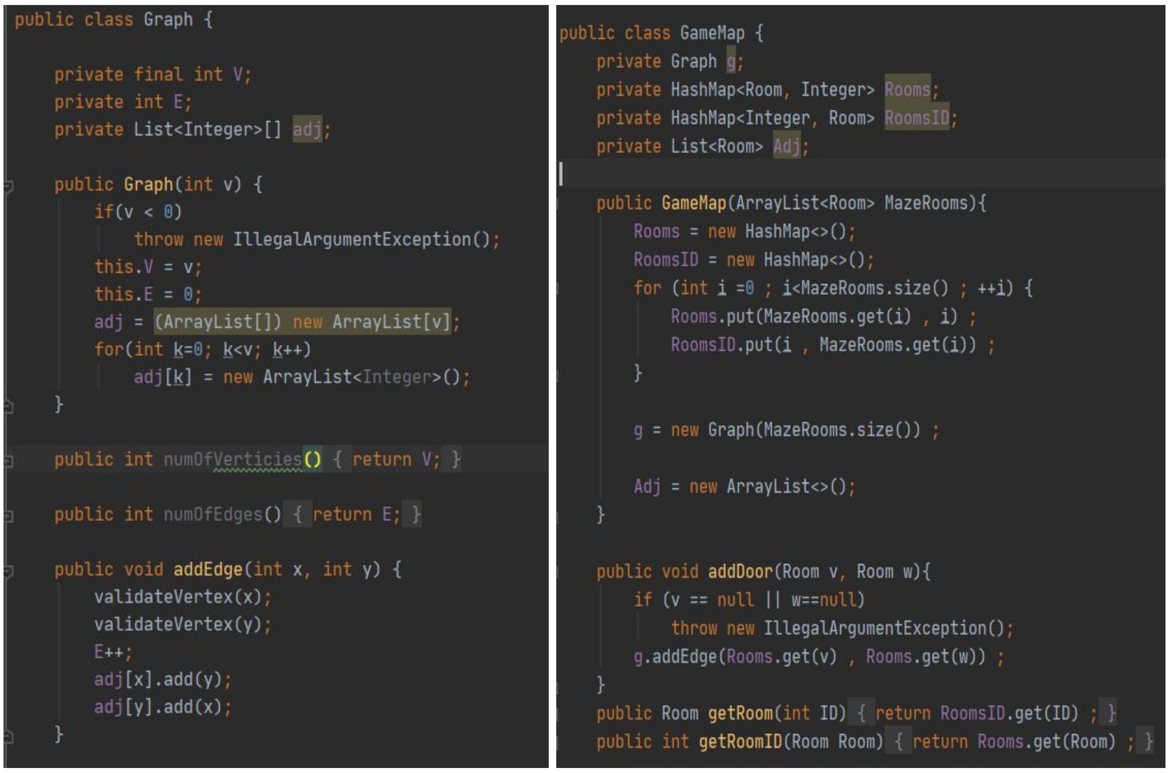
- Turn Left : The player turns to the Left of its current position
- Turn Right : The player turns to the right of its current position
- Move Forward : If the door is open the player can move to the next room.
- Look : To Look to the object the player faces.

- Check <Object> : This order to check the paint, mirror, chest or door if it has a key or requires a key
- Buy <Item> : when you are in front of a seller you can buy any item he has.
- Sell <Item> : when you are in front of a seller you can sell any item you have to him.
- List Player Items : List the items that the player bought or acquired earlier.
- List Seller Items : List the items that the seller currently has.
- List Chest Items : List the items that the chest currently has.
- Use <Item> : If you have a flashlight or a key you can use it
- Switch lights : if the room is dark and has lights you can turn it on.
- Restart : replay the game from the beginning
- Quit : Game is Over.

There is also a timer that is set for 120 minutes for the player to get outside the maze, otherwise he will lose and the program will terminate.

Data Structures and Algorithms

As mentioned before, The most obvious data structure used is the Graph data structure, The graph we have is represented in an adjacency list where each room represent a vertex and has all other rooms connected to it through door edges in a list of ArrayList type such that we can directly access the current room list but need to search through the rooms connected to it one by one. So in the worst case scenario we need to go through all the rooms connected to a specific room so it will cost O(V) time complexity for each command that needs to refer to the adjacency list of a specific room.

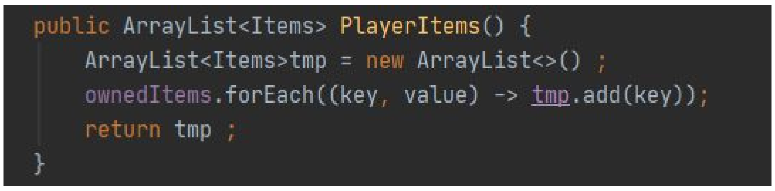


```java
public class Graph {

    private final int V;
    private int E;
    private List<Integer>[] adj;

    public Graph(int v) {
        if(v < 0)
            throw new IllegalArgumentException();
        this.V = v;
        this.E = 0;
        adj = (ArrayList[]) new ArrayList[v];
        for(int k=0; k<v; k++)
            adj[k] = new ArrayList<Integer>();
    }

    public int numOfVerticies() { return V; }

    public int numOfEdges() { return E; }

    public void addEdge(int x, int y) {
        validateVertex(x);
        validateVertex(y);
        E++;
        adj[x].add(y);
        adj[y].add(x);
    }
```

```java
public class GameMap {
    private Graph g;
    private HashMap<Room, Integer> Rooms;
    private HashMap<Integer, Room> RoomsID;
    private List<Room> Adj;

    public GameMap(ArrayList<Room> MazeRooms){
        Rooms = new HashMap<>();
        RoomsID = new HashMap<>();
        for (int i =0 ; i<MazeRooms.size() ; ++i) {
            Rooms.put(MazeRooms.get(i) , i) ;
            RoomsID.put(i , MazeRooms.get(i)) ;
        }

        g = new Graph(MazeRooms.size()) ;

        Adj = new ArrayList<>();
    }

    public void addDoor(Room v, Room w){
        if (v == null || w==null)
            throw new IllegalArgumentException();
        g.addEdge(Rooms.get(v) , Rooms.get(w)) ;
    }
    public Room getRoom(int ID) { return RoomsID.get(ID) ; }
    public int getRoomID(Room Room) { return Rooms.get(Room) ; }
```

Figure2: Basic graph implementation with the rooms graph built using it

ArrayLists and HashMaps are very common in the design, Hashmap is used to gather the items in the seller and player classes in order to refer to the items or remove them quickly in O(LogV) complexity rather than searching each element if it is equal to the current item in an ArrayList in O(V). HashMap is also used in listing the available commands for the player so that he can’t pick any other command, otherwise he will be notified that his command is not available since no command with the same string in the HashMap.



```
public ArrayList<Items> PlayerItems() {
    ArrayList<Items>tmp = new ArrayList<>() ;
    ownedItems.forEach((key, value) -> tmp.add(key));
    return tmp ;
}
```

Figure3: An example of using both a HashMap as well as an ArrayList

No specific or common algorithm was implemented in the code probably because it doesn’t require a specific way to move between rooms, it is just implementation to the different cases.

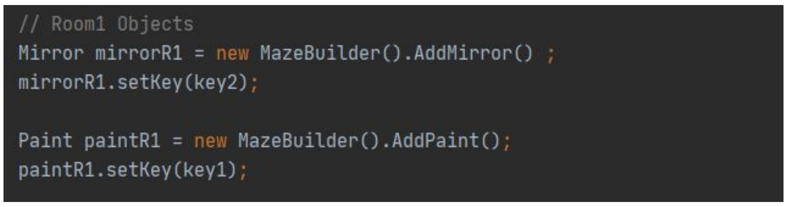
Clean Code Principles

Introduction

The main Idea behind the clean code principles is to make sure the code is readable, reusable, flexible and extendable to any developer other than the author of this code. In writing a code many pieces of advice should be taken into consideration to guarantee you're applying the clean code principles according to Robert Martin book. This report reflects the most considered parts of the clean code principles implemented in the assignment that I've been working on.

Comments

Starting with the comments, there are only a few comments that are brief and not redundant and are written for a specific purpose not generic like a signature.

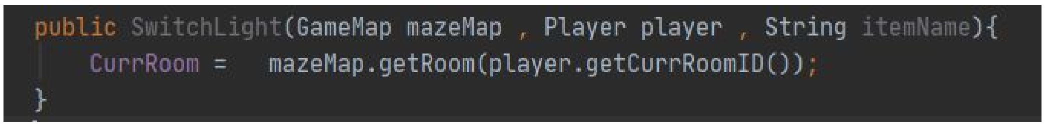


```
// Room1 Objects
Mirror mirrorR1 = new MazeBuilder().AddMirror() ;
mirrorR1.setKey(key2);

Paint paintR1 = new MazeBuilder().AddPaint();
paintR1.setKey(key1);
```

Figure4: An example of using a comment for a collection of statements to illustrate its use

Methods

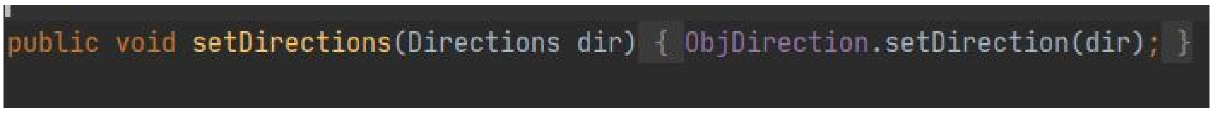
For the Functions part, more than three arguments for any method are not acceptable, it’s simply too much for a method to hold more than this, and thus no method in any class takes more than three arguments.



```
public SwitchLight(GameMap mazeMap , Player player , String itemName){
    CurrRoom =   mazeMap.getRoom(player.getCurrRoomID());
}
```

Figure5: An example of using no more than 3 arguments in a method

Furthermore, each method has only one functionality in order to apply the single responsibility principle, such that a boolean argument should be as few as possible and isn’t used to make multiple actions for a method.



```
public void setDirections(Directions dir) { ObjDirection.setDirection(dir); }
```

Figure6: An example of a method with one functionality only

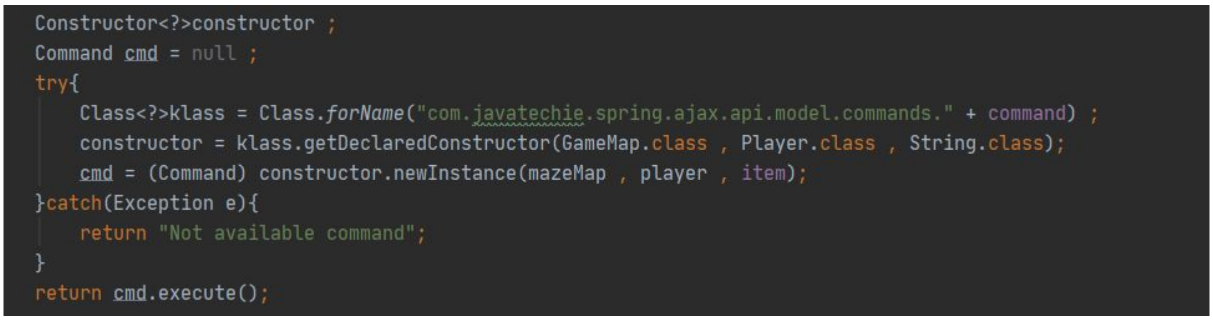
Also, when overriding a method from a parent or interface, we should take into consideration that this action is safe and doesn’t cause conflicts when implementing higher level details.

Moreover, the design is free of clutter such that no constructor or a method is left empty or not used at all, also no method in some class calls a method in another class since this features envy between classes according to Robert Martin book.

Thus here we should take into consideration that a method calling another method should be close to each other in distance, not separated by other classes which have other responsibilities and no interaction between them.

Duplicates and Redundancy

One of the most important principles in the Robert martin book is to reduce duplications and redundancy (Don't Repeat Yourself) or DRY such that instead of using if/else or switch case many times, use polymorphism instead, our implemented code includes polymorphism every where since there are items and room objects that has repeated functionality, and from here we can conclude that when there are many classes share the same functionality, this functionality might also be implemented in a class that a parent class or interface uses instead of implementing it over and over inside each class, here comes the beauty of structured programming such that a collection of methods builds larger methods, but it is more simple to read and apply the DRY principle.

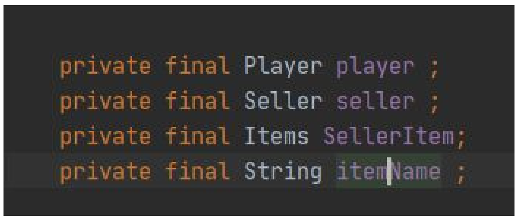


```
Constructor<?>constructor ;
Command cmd = null ;
try{
    Class<?>klass = Class.forName("com.javatechie.spring.ajax.api.model.commands." + command) ;
    constructor = klass.getDeclaredConstructor(GameMap.class , Player.class , String.class);
    cmd = (Command) constructor.newInstance(mazeMap , player , item);
}catch(Exception e){
    return "Not available command";
}
return cmd.execute();
```

Figure7: An example of using **Reflection** to convert a string into class instead of making many if/switch statements for generating commands

Variables

I have taken into consideration that the naming of variables doesn't cause inconsistency as much as possible like a Room Object variable is always the same (Obj) or (roomObject) in the main class, most of the variables are named the same way and the variable names are descriptive to their functionality such that it would be easy for normal people to understand especially in the high level classes.



```
private final Player player ;
private final Seller seller ;
private final Items SellerItem;
private final String itemName ;
```

Figure8: An example of naming variables at a specific implemented command

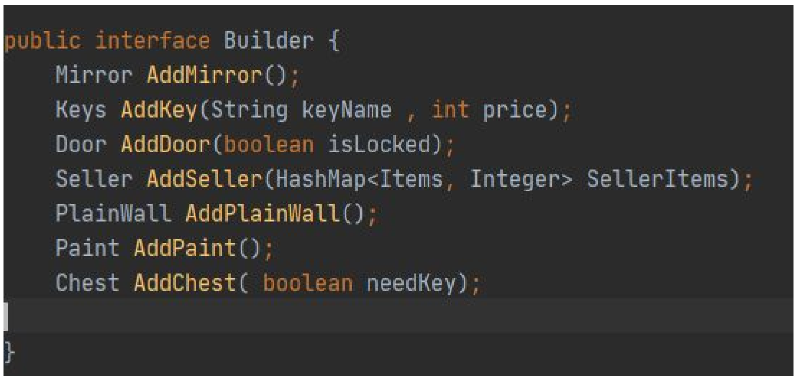
Effective Java Principles

Introduction

We apply the Effective Java principles according to Joshua Bloch book in order to effectively benefit from the java code we write, he explains the best practices in using java features.

Using Builders

I have started by making a builder that helps in limiting the amount of constructor arguments since they will be a lot without the builder, with one more thing is to separate the low level classes declaration and the high level classes.

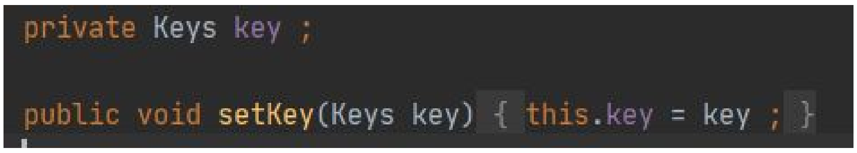


```java
public interface Builder {
    Mirror AddMirror();
    Keys AddKey(String keyName , int price);
    Door AddDoor(boolean isLocked);
    Seller AddSeller(HashMap<Items, Integer> SellerItems);
    PlainWall AddPlainWall();
    Paint AddPaint();
    Chest AddChest( boolean needKey);

}
```

Figure9: Using the builder to build the maze objects

Avoid creating objects

I have avoided creating objects over and over when they can only be initialized to have the same values or properties of an existing object to obey the rule of avoiding creating objects.



```
private Keys key ;

public void setKey(Keys key) { this.key = key ; }
```

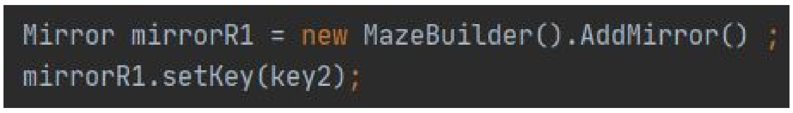
Figure10: Initializing objects through composition

Avoid Finalizers

Also, according to the book, Finalizeres are dangerous and unpredictable and thus we are not supposed to do something critical with finalizers, I preferred not to use any finalizers in my design.

Method common to all objects

Coming to the classes and objects, we try our best to make the classes inaccessible, only some member methods could be used outside the class, this will help to apply the abstraction and encapsulation principles and make isolated but can be developed modules.



```
Mirror mirrorR1 = new MazeBuilder().AddMirror() ;
mirrorR1.setKey(key2);
```

Figure11: Building a mirror through using methods of builder class

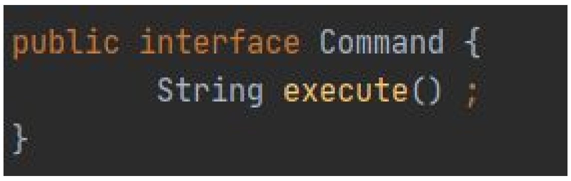
Accordingly, the fields of a public class shouldn’t be public for many reasons including that they won’t benefit from encapsulation as well as they can’t enforce variants, and thus they should be accessed through member methods like getters and setters.

Favour composition over inheritance

The other point is to favor composition over inheritance since inheritance violates encapsulation, and thus most of the code uses other classes with composition rather than inheriting.

Prefer Interfaces over abstract classes

Also, Interfaces are more preferable than the abstract classes, that's because Java only allows single inheritance, so that restricts the type of classes that you might inherit and thus an Interface works as abstract inheritance and thus more preferable to be used, also they enable safe and powerful functionality enhancements and many classes can use shared features of interfaces.



```java
public interface Command {
        String execute() ;
}
```

Figure12: Using the command pattern as an interface

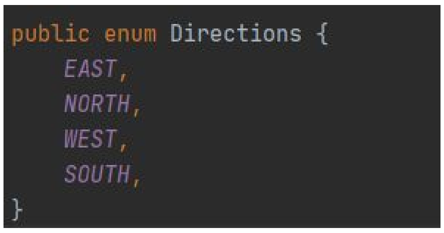
Generics

Now the generic parts of the book tells us many rules, first one is to eliminate unchecked warnings, and whenever they occur we should edit this part.

Also, it tells us that we should prefer Lists over Arrays, since Arrays are covariants but Lists are invariants and arrays are not supposed to be of a generic type, a parameterized type, or a type parameter, they enforce their elements type at runtime while Lists enforce their type at compile time but discard it through runtime.

Enums

For the enums part, I used enums to decide the directions of a room, it is mentioned in the book that enums are more preferable than integer constants.

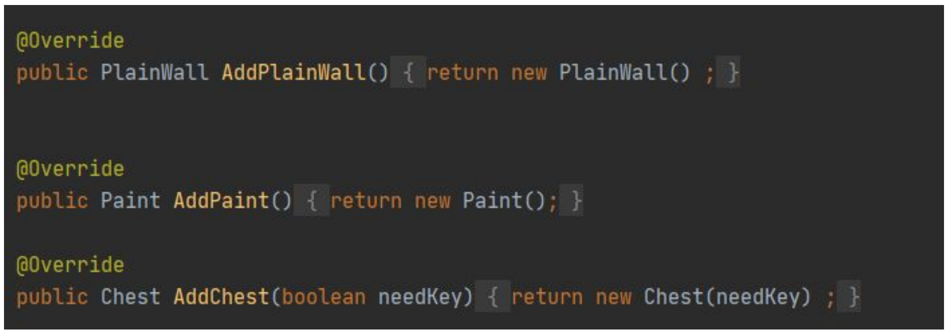


```java
public enum Directions {
    EAST,
    NORTH,
    WEST,
    SOUTH,
}
```

Figure13: Using Enum for the direction of player

Annotations

The book suggests that we should use Override annotation consistently , that is, never to forget to use the annotation when overriding a superclass or implementing the interface's declaration.

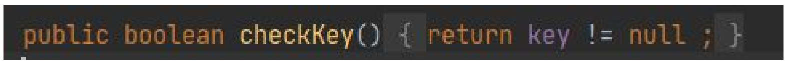


```java
@Override
public PlainWall AddPlainWall() { return new PlainWall() ; }


@Override
public Paint AddPaint() { return new Paint(); }

@Override
public Chest AddChest(boolean needKey) { return new Chest(needKey) ; }
```

Figure14: Overriding the builder interface's methods

Returning types

Furthermore, it is mentioned that when returning values from objects, we shouldn't return null values, but instead we can use boolean or empty arrays or collections.



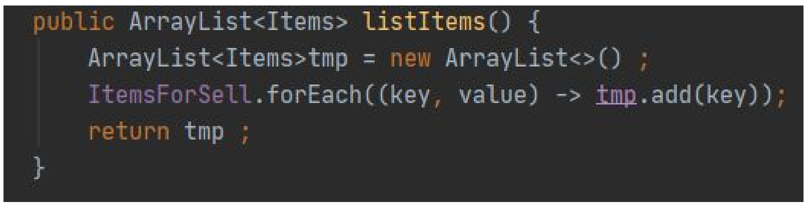
```java
public boolean checkKey() { return key != null ; }
```

Figure15: Returning a boolean instead of the key as null or not

General Programming

For the general programming part, I've minimized the scope of Local variables such that whenever it's used it's declared so that it doesn't cause unexpected exceptions of downcasting or upcasting somewhere in the code.

Most of the for loops I've used aren't the traditional ones, but the for-each loops since they are more powerful in structure and more used for effective java.



```java
public ArrayList<Items> listItems() {
    ArrayList<Items>tmp = new ArrayList<>() ;
    ItemsForSell.forEach((key, value) -> tmp.add(key));
    return tmp ;
}
```

Figure16: Using forEach loop to minimize the scope of the loop

Moreover, I've got to search java libraries to be able to use them effectively especially the Timer libraries, they are quite new to my knowledge.

Finally, there are lots of other details that were hidden in the code I’ve worked on such as, using string at appropriate cases, and compare them with equals, prefer primitive types to boxed primitives, referring to objects by their interface and downcast when using a specific object of a shared interface and so on, I’ve also faced errors regarding strings concatenation, but I worked on this part and it’s better now.

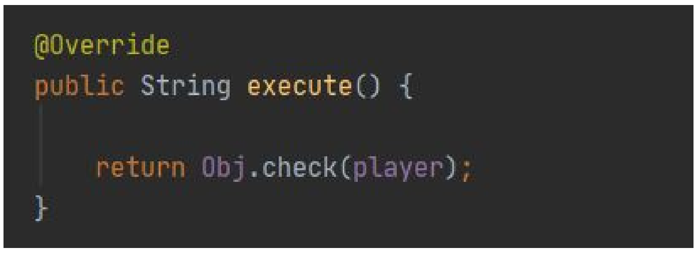
SOLID Principles

Introduction

There are 5 essential principles in Object Oriented Design according to Robert C Martin and are abbreviated as SOLID, Following the principles respectively:

Single-Responsibility Principle

This principle tells us in brief that no method should have more than one functionality or it would cause more coupling and less cohesion, and no class should have more than one responsibility. This should have been implemented in the design I've presented, even boolean arguments are never for choosing multiple options inside the method but to do an action just like any other arguments.

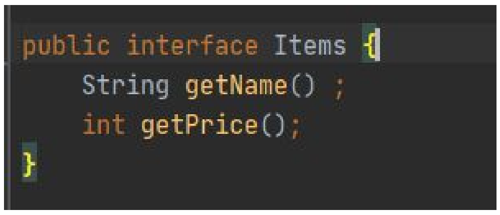


```java
@Override
public String execute() {

    return Obj.check(player);
}
```

Figure17: An example of an executor method that has only one functionality

Open-Closed Principle

This principle tells us that any class should be open for extension but closed for modification, such that if it happens to implement a class, you should provide all the related fields and methods for it and consider not modifying it later but to extend it to do more. Relatively, I've implemented the classes to apply 80% percent of the principle, not as complete as the principle restricts us to do.



```java
public interface Items {
    String getName() ;
    int getPrice();
}
```

Figure18: The Items interface is an interface that can always be implemented through new items kind

Liskov Substitution Principle

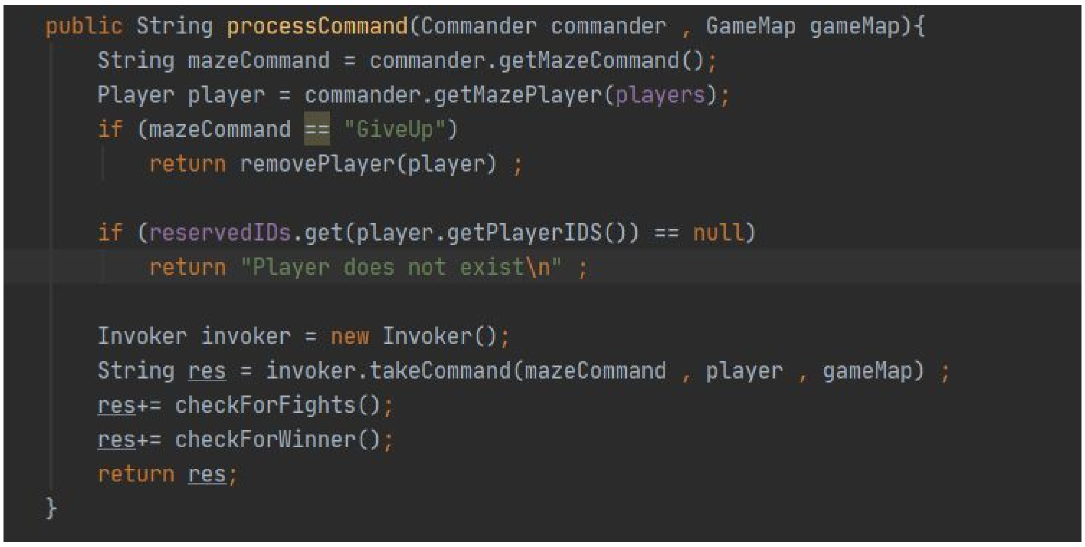
The Liskov Principle, as I've understood, tells us that whenever a class inherits another class, it should be able to use all the methods of the superclass without conflict in its real behaviour. I've benefited from using superclasses in my design without such conflicts, all the derived classes use their superclass methods and might override some of them to make additional or unique actions.

Interface Segregation Principle

This principle tells us that a client class shouldn’t implement an interface that generally won’t need or won’t use in the end. Since the interfaces help us to define classes of the same properties but unique behaviours and thus whenever there is an interface that won’t be used at all to separate between different classes why would it be implemented. I believe the design I am presenting doesn’t have such Interfaces such as Items and command, I use both of them to declare different types of classes.

Dependency Inversion Principle

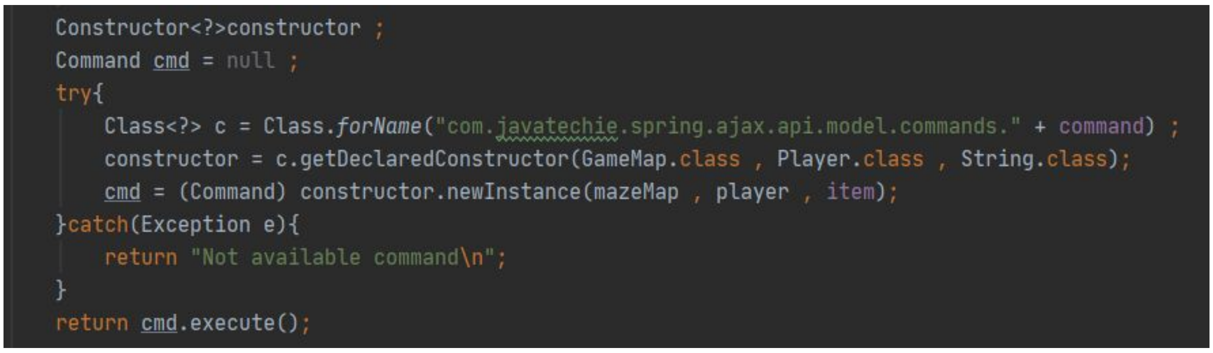
This principle is very important in the Object Oriented Design world, it can be described with many articles and everyone should apply it, it says that high level modules shouldn’t depend on the low level modules, but should depend on the abstraction, and abstraction shouldn’t depend on details, but details should depend on abstraction. It is obvious from the definition that abstraction is a must in any code and to reach a good level of abstraction we need to separate the low level details from the higher level details as well as classes of different levels, that’s why it’s mentioned that we should make builders instead of constructors in the Effective Java Principles and many other books and articles that have principles that apply this principle too. I’ve tried my best to apply in the design and hope it’s at least 80% level of abstraction to obey this principle.



```java
public String processCommand(Commander commander , GameMap gameMap){
    String mazeCommand = commander.getMazeCommand();
    Player player = commander.getMazePlayer(players);
    if (mazeCommand == "GiveUp")
        return removePlayer(player) ;

    if (reservedIDs.get(player.getPlayerIDS()) == null)
        return "Player does not exist\n" ;

    Invoker invoker = new Invoker();
    String res = invoker.takeCommand(mazeCommand , player , gameMap) ;
    res+= checkForFights();
    res+= checkForWinner();
    return res;
}
```

Figure19: This is an example of an 80% abstraction code for processing the commands in

the project



```java
Constructor<?>constructor ;
Command cmd = null ;
try{
    Class<?> c = Class.forName("com.javatechie.spring.ajax.api.model.commands." + command) ;
    constructor = c.getDeclaredConstructor(GameMap.class , Player.class , String.class);
    cmd = (Command) constructor.newInstance(mazeMap , player , item);
}catch(Exception e){
    return "Not available command\n";
}
return cmd.execute();
```

Figure20: This is how commands are directly processed through Reflection

Design Patterns

Introduction

A good design should have a high level of abstraction, and a collection of design patterns were set to help solve design issues by a collection of engineers who wrote the Head First design patterns. I’ve referred to the book and other online sources to choose which design is the most appropriate for each case in our assignment, I’ve implemented three design patterns which are , the singleton design pattern for the player , the builder design pattern for the maze objects, and the command design pattern for the commands. I believe there should be more designs used and I should have made the code more simple, so I will be working on it further to implement a better design. Following a brief description of each design I’ve used and how I used it.

Builder Design Pattern

The builder is a design of the Gang of the four design patterns and it is used widely in object oriented design. Its main idea is to separate construction of complex objects from its representation, such that the high level classes have a high level of abstraction and is easy to read and understand the purpose of each statement. This is why I’ve picked the builder and it really helped me a lot to define objects of different requirements and behaviours.

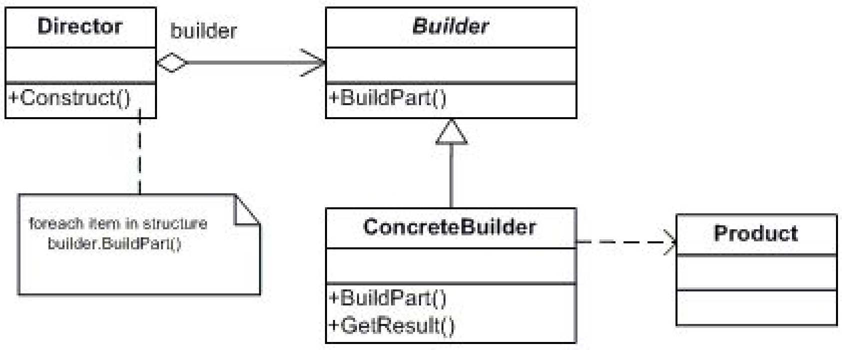
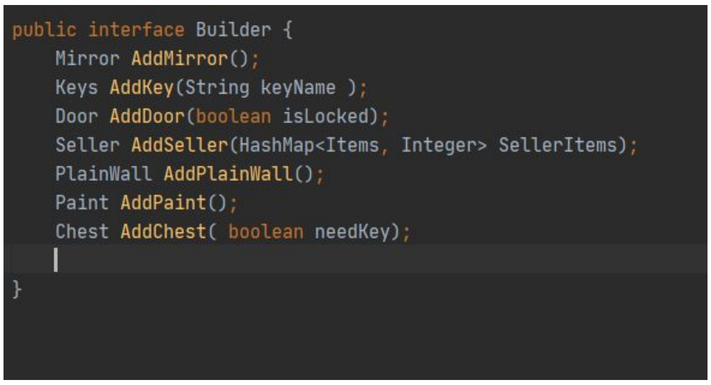


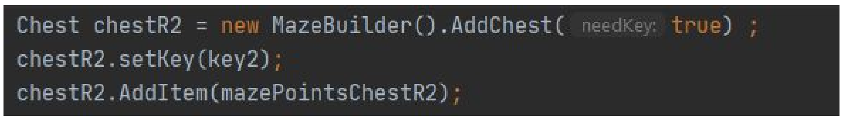
Figure21: A sample UML Diagram shows how Builder design pattern could be implemented

In the project code, the design is implemented through an interface of name (Builder), a class called (MazeBuilder) implements this interface and is used inside the MazeGame class which is responsible on implementing the game actions, the MazeBuilder class uses the original objects classes to build a collection of rooms that has objects and build the maze inside MazeGame.



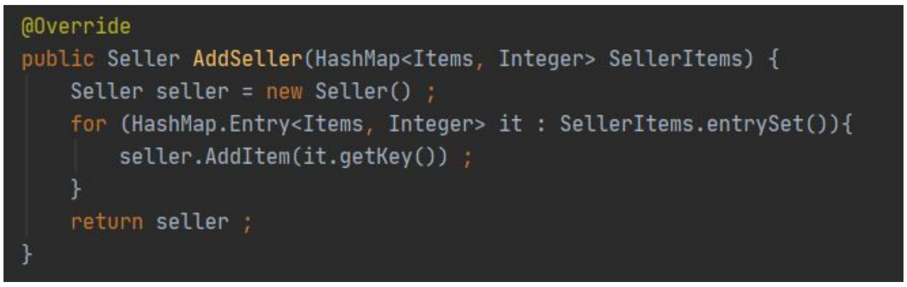
```
public interface Builder {
    Mirror AddMirror();
    Keys AddKey(String keyName );
    Door AddDoor(boolean isLocked);
    Seller AddSeller(HashMap<Items, Integer> SellerItems);
    PlainWall AddPlainWall();
    Paint AddPaint();
    Chest AddChest( boolean needKey);
    
}
```

Figure22: The Builder interface that is being implemented



```
Chest chestR2 = new MazeBuilder().AddChest( needKey: true) ;
chestR2.setKey(key2);
chestR2.AddItem(mazePointsChestR2);
```

Figure23: Example of using the builder design pattern to build the chest with a key



```
@Override
public Seller AddSeller(HashMap<Items, Integer> SellerItems) {
    Seller seller = new Seller() ;
    for (HashMap.Entry<Items, Integer> it : SellerItems.entrySet()){
        seller.AddItem(it.getKey()) ;
    }
    return seller ;
}
```

Figure24: Example of building the Seller inside the builder

Command Pattern

The command pattern is a behavioural pattern that is used to encapsulate the information and implementation of an action and help to send a request that is processed under an invoker object, this invoker assigns the most appropriate object type to the requested object and executes it. Since our design depends mostly on commands I've decided to pick this design and it helped me a lot to process the commands.

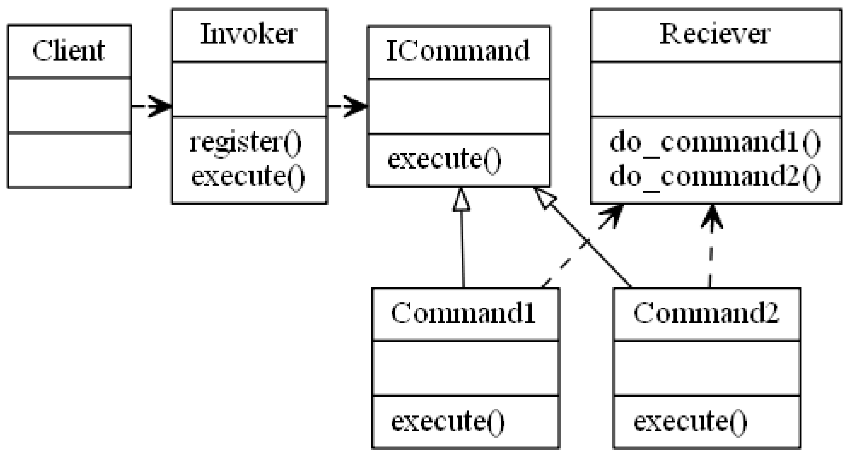
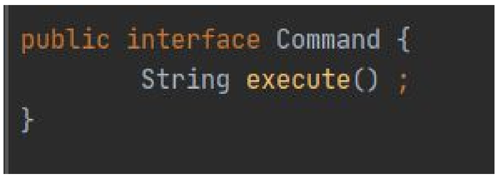


Figure25: The Command design pattern UML Diagram as implemented in the project

In the code design, the Command design pattern is implemented like the above UML with a small modification to improve the level of abstraction. There is the Command interface that has only one method which is (execute) and is responsible for implementing the commands that are received through the (Invoker).

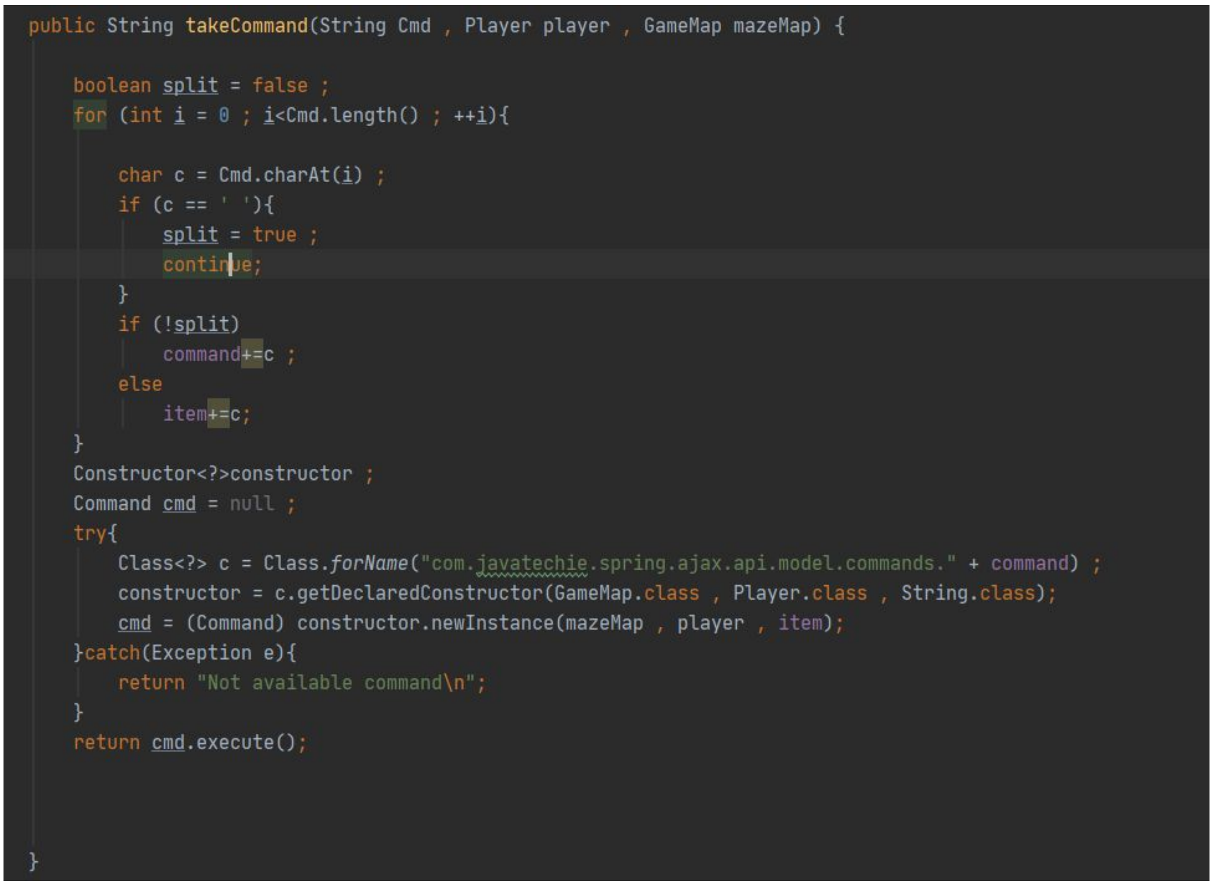
The Invoker is implemented through reflection, which means converting a string into a class name with specific parameters, so that it helps very much in getting the command executed directly through three lines of code instead of making many lines of if/else or switch/case statements.

While the Invoker actually receives the Command string through another ProcessCommand method inside MazeGame Class which is responsible on preparing the Invoker to take the command with its parameters and check the functionality of the sender state and the command state, and this one also receives the command through a commander class that saves the sender information that comes through the REST API Controller.



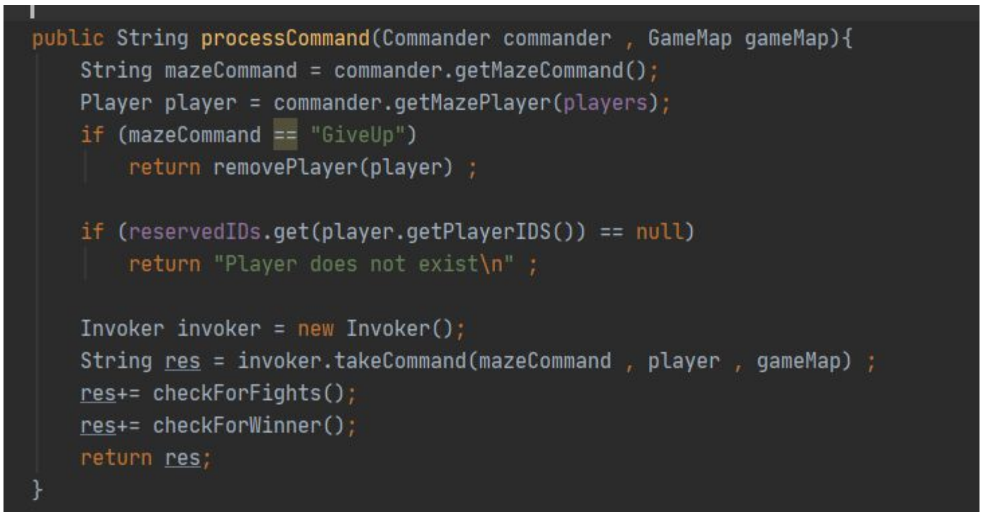
```java
public interface Command {
        String execute() ;
}
```

Figure26: The Command Interface



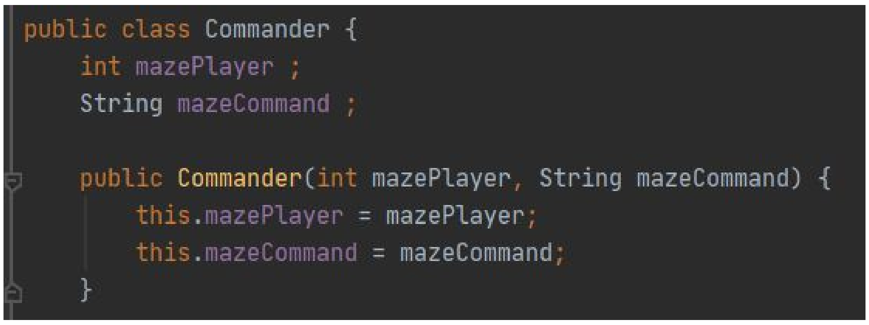
```java
public String takeCommand(String Cmd , Player player , GameMap mazeMap) {

    boolean split = false ;
    for (int i = 0 ; i<Cmd.length() ; ++i){

        char c = Cmd.charAt(i) ;
        if (c == ' '){
            split = true ;
            continue;
        }
        if (!split)
            command+=c ;
        else
            item+=c;
    }
    Constructor<?>constructor ;
    Command cmd = null ;
    try{
        Class<?> c = Class.forName("com.javatechie.spring.ajax.api.model.commands." + command) ;
        constructor = c.getDeclaredConstructor(GameMap.class , Player.class , String.class);
        cmd = (Command) constructor.newInstance(mazeMap , player , item);
    }catch(Exception e){
        return "Not available command\n";
    }
    return cmd.execute();


}
```

Figure27: The Invoker class which has the takeCommand method for processing string commands



```
public String processCommand(Commander commander , GameMap gameMap){
    String mazeCommand = commander.getMazeCommand();
    Player player = commander.getMazePlayer(players);
    if (mazeCommand == "GiveUp")
        return removePlayer(player) ;

    if (reservedIDs.get(player.getPlayerIDS()) == null)
        return "Player does not exist\n" ;

    Invoker invoker = new Invoker();
    String res = invoker.takeCommand(mazeCommand , player , gameMap) ;
    res+= checkForFights();
    res+= checkForWinner();
    return res;
}
```

Figure28: The ProcessCommand method inside MazeGame that takes the commander argument



```
public class Commander {
    int mazePlayer ;
    String mazeCommand ;

    public Commander(int mazePlayer, String mazeCommand) {
        this.mazePlayer = mazePlayer;
        this.mazeCommand = mazeCommand;
    }
```

Figure29: The Commander class that takes the sender information through the rest api controller

MVC design pattern (Using Spring Boot Framework)

Introduction

MVC is a software design pattern that includes three main parts, the Moder, View and Controller.

Model

Model is the data about the state of the application or its components. May include routines for modification or access. In the code design all the objects and commands as well as the dto's classes are considered part of the Model package as follows:

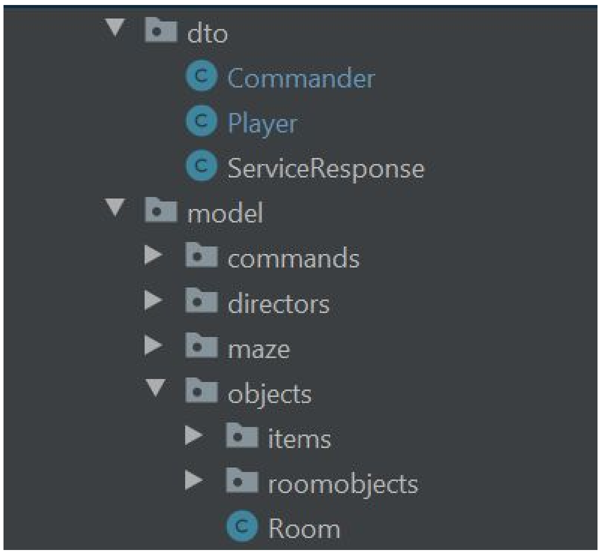


Figure30: The main parts of the model inside the main package

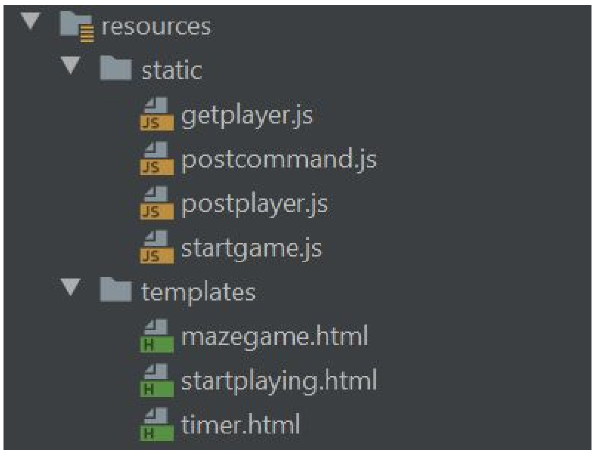
The dto or data transfer objects are objects used to transfer data from the controller to the model.

View

View is an interpretation of the data (model). This is only limited to a visual representation, but could be audio, derived information (e.g. statistics piped into another model object), etc. Furthermore, a single model may have multiple views.

According to our project, the view was about the graphical user interface which has a simple console and buttons to show the players and other details.

I've implemented it using thymeleaf HTML pages as well as ajax using jquery javascript library.



```
resources
  static
    getplayer.js
    postcommand.js
    postplayer.js
    startgame.js
  templates
    mazegame.html
    startplaying.html
    timer.html
```

Figure31: The view part of the design

Controller

The **Controller** handles external input to the system invoking modifications on the model. The control/view may be closely related (in the case of a UI). However, other external input (such as network commands), may be processed which are completely independent of the view.

In the code design it's implemented through two main parts, the REST APIs and the controllers in the spring boot framework. The Rest APIs control starting the game, adding players and processing commands, and the Controller controls directing users to the web pages through, and this is all through GET and POST requests (in json form) where I used the GET for non-parameterized requests and POST for the parameterized requests.

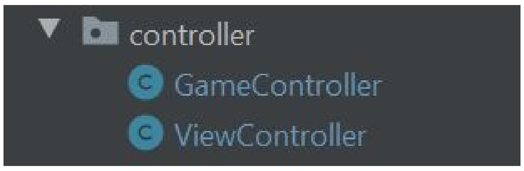
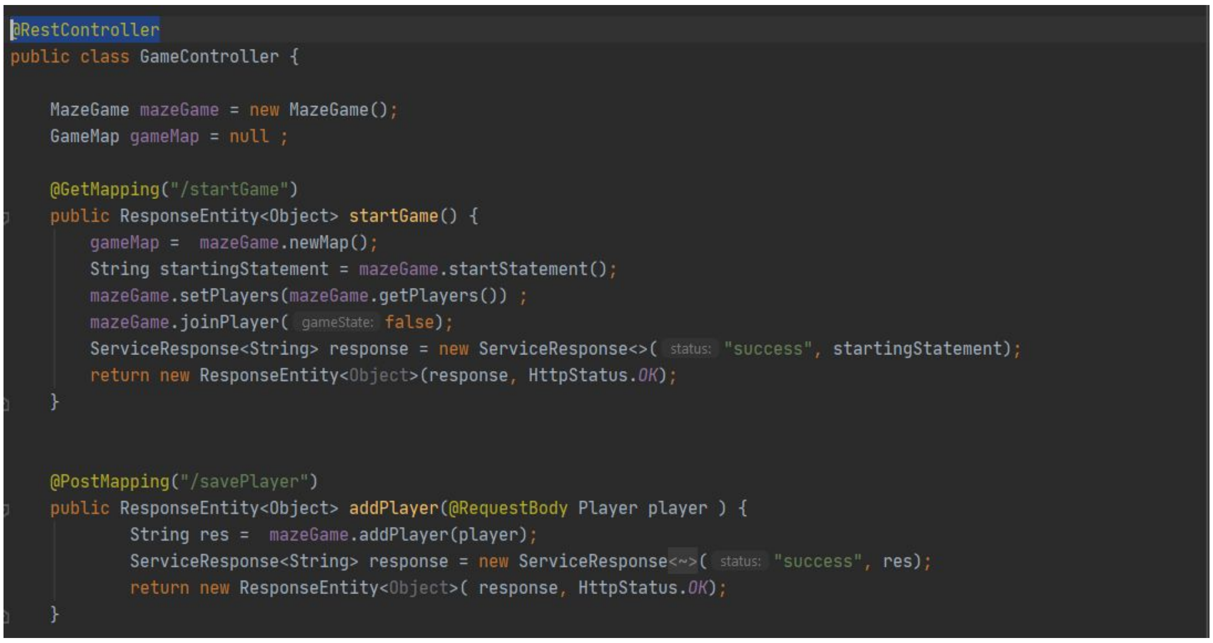


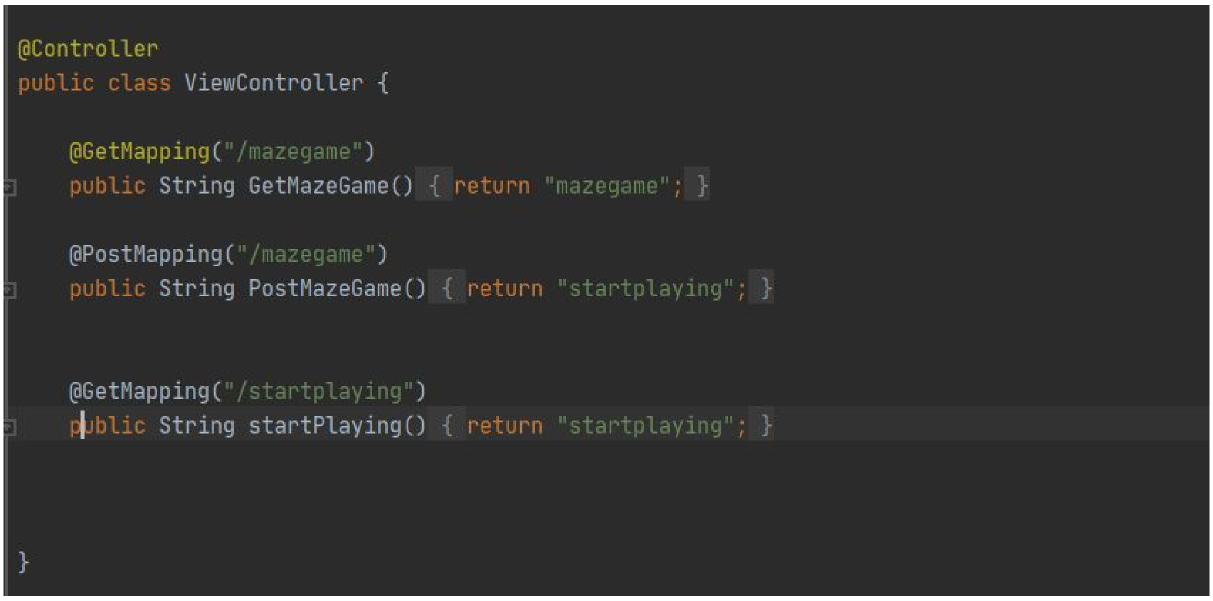
Figure32: The Controller part of the design

The Rest Controller is as follows: (with two sample requests)



```
@RestController
public class GameController {

    MazeGame mazeGame = new MazeGame();
    GameMap gameMap = null ;

    @GetMapping("/startGame")
    public ResponseEntity<Object> startGame() {
        gameMap =  mazeGame.newMap();
        String startingStatement = mazeGame.startStatement();
        mazeGame.setPlayers(mazeGame.getPlayers()) ;
        mazeGame.joinPlayer( gameState: false);
        ServiceResponse<String> response = new ServiceResponse<>( status: "success", startingStatement);
        return new ResponseEntity<Object>(response, HttpStatus.OK);
    }


    @PostMapping("/savePlayer")
    public ResponseEntity<Object> addPlayer(@RequestBody Player player ) {
            String res =  mazeGame.addPlayer(player);
            ServiceResponse<String> response = new ServiceResponse<~>( status: "success", res);
            return new ResponseEntity<Object>( response, HttpStatus.OK);
    }
```

Figure33: The GameController class



```
@Controller
public class ViewController {

    @GetMapping("/mazegame")
    public String GetMazeGame() { return "mazegame"; }

    @PostMapping("/mazegame")
    public String PostMazeGame() { return "startplaying"; }


    @GetMapping("/startplaying")
    public String startPlaying() { return "startplaying"; }



}
```

Figure34: The ViewController class

Styling

According to google styling guide (the link below), I've noticed that it also depends on and uses the Effective Java, Clean Code and SOLID Principles. I've concentrated on the Formatting, Naming and some Programming practices of Google Java Style. For the formatting, first of all, the braces should always exist with if, else, for and do statements and they have a collection of rules, such as: No Line break before or after the opening brace, no line break before or after the closing brace except when brace terminates the body of a method or constructor or named class. The empty block should be concise, I personally removed them from my code since there is either a default constructor instead of an empty constructor or no need for an empty method, according to Effective Java Principles. Also, there should be one statement only at each line then followed with a break line, and after 100 line characters this line should be wrapped. For the whitespaces, there should be a single blank line between consecutive members or initializers of a class. For the variable declaration, each variable should be declared individually i.e. not more than one object is declared with the object type, and they shouldn't be declared far away from where they will be used first time, but they should be declared whenever needed, not necessary at the beginning of the block. For the switch cases, since I've used them in my code many times, they should have a *default:* presented even if it is left empty. Finally, for naming variables, just as mentioned in the Clean Code Principles above, the variables shouldn't have special characters or suffix and prefix in them. The classes names should follow the UpperCamelCase for each different word they should start with upper case letter, while for the method names they should follow the lowerCamelCase, the same as the previous one except that the first word in the name should be lower case. These are the most obvious styles I've tried to obey in my design, probably not 100%, but as much as possible.

https://google.github.io/styleguide/javaguide.html#s3.2-package-statement

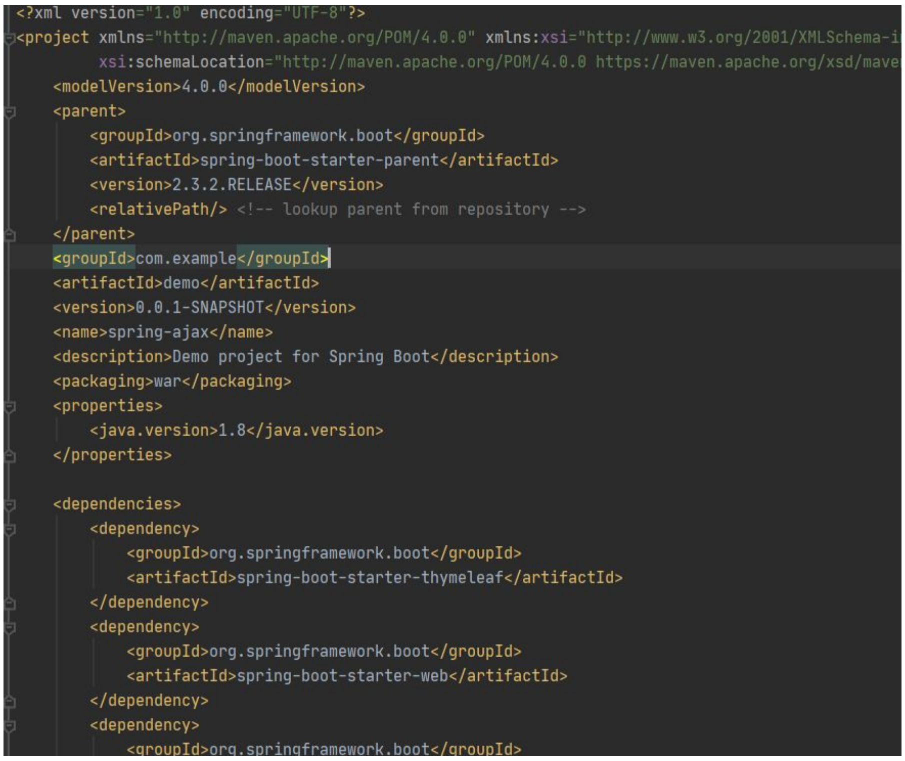
DevOps

Introduction

While DevOps is a shortcut for the words Developers and Operators, it shows how the workflow is done between those two parts, where it is times easier to develop and deploy the code through a pipeline using Jenkins for example and having containerized software using docker.

Maven

Maven is the building tool that is responsible for running the application according to a number of dependencies picked to proportion with the application like spring-boot, bootstrap, jquery, tomcat server..etc., with a number of plugins and other packaging details like war file and name of the artifact and Id. They are all embedded inside the POM.xml file that is very essential in building the application and marking its success or failure, the POM.xml file of the project we are working on:



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-i
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/mave
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.2.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>spring-ajax</name>
    <description>Demo project for Spring Boot</description>
    <packaging>war</packaging>
    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

Figure35: Part of the POM.xml that shows main parts of it

Git and GitHub

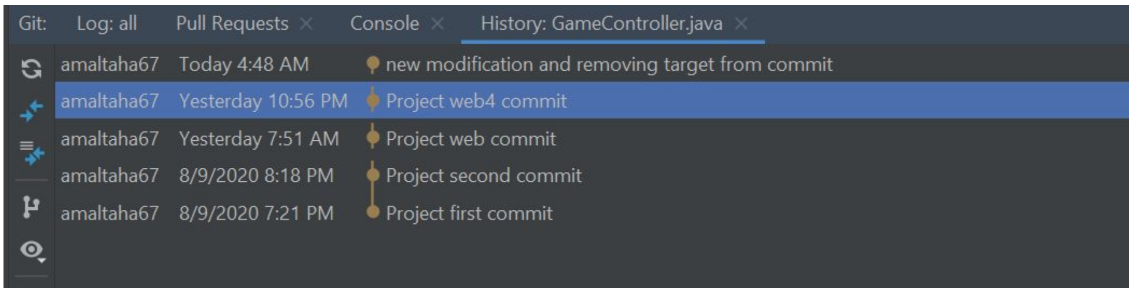
Git is version control system that helps to track the code versions and branches, especially when there are many people work on the same code together, each one is capable of pulling the code from one server and request to push a modification, they are also capable of knowing what was changed and get back to the version the want or work on a branch rather than on the same code together and then merge the correct changes together again. The log provides a visualization of the branches and the master branch.

GitHub is where the code published from the Git and can be viewed and use source code through the people with different credentials.

This is exactly how I used Git and GitHub in developing my web application, keep committing modifications and track the workflow.

A URL for the GitHub repository I used for the project:

https://github.com/amaltaha67/World-Navigator-Project.git


Git: Log: all Pull Requests Console History: GameController.java

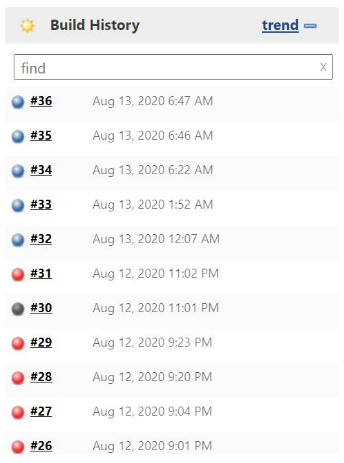
```
amaltaha67   Today 4:48 AM        new modification and removing target from commit
amaltaha67   Yesterday 10:56 PM   Project web4 commit
amaltaha67   Yesterday 7:51 AM    Project web commit
amaltaha67   8/9/2020 8:18 PM     Project second commit
amaltaha67   8/9/2020 7:21 PM     Project first commit
```

Figure36: The Log that tracks commits history

Jenkins

Jenkins is a continuous integration (CI) server that helps to build a pipeline where the code is being developed and deployed directly after each time a new modification is committed to the code, such a tool will allow to release the software for each modification and check its reliability. It can also be used for simple continuous deployment (CD) to a specific container like docker and tomcat server and so on.

In the project, Jenkins server was set on an AWS EC2 instance and was used to continuously integrate the project through the GitHub repository, where the Java, git and maven paths were set in its configuration to be used to build the project successfully, and it has maven invoker and integration plugins, as well as github and deploy to container plugins. A tomcat server was put as a container to the war file deployment through the url mentioned below in AWS EC2 Instances part.


Build History trend

find

#36	Aug 13, 2020 6:47 AM
#35	Aug 13, 2020 6:46 AM
#34	Aug 13, 2020 6:22 AM
#33	Aug 13, 2020 1:52 AM
#32	Aug 13, 2020 12:07 AM
#31	Aug 12, 2020 11:02 PM
#30	Aug 12, 2020 11:01 PM
#29	Aug 12, 2020 9:23 PM
#28	Aug 12, 2020 9:20 PM
#27	Aug 12, 2020 9:04 PM
#26	Aug 12, 2020 9:01 PM

Figure37: The Log that tracks commits history

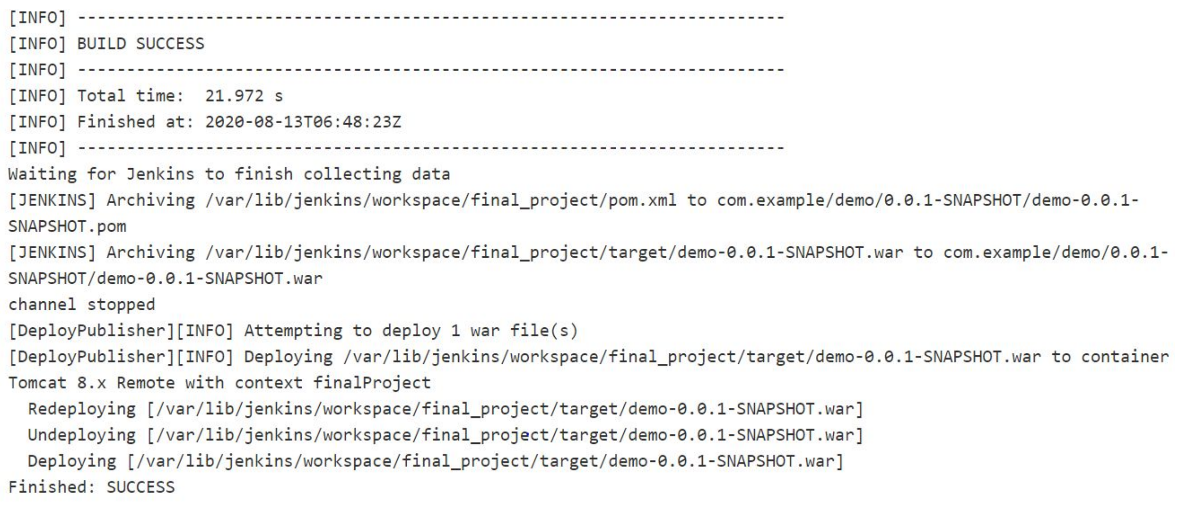
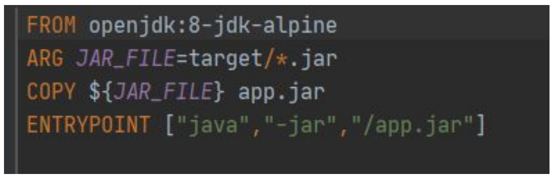

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  21.972 s
[INFO] Finished at: 2020-08-13T06:48:23Z
[INFO] ------------------------------------------------------------------------
Waiting for Jenkins to finish collecting data
[JENKINS] Archiving /var/lib/jenkins/workspace/final_project/pom.xml to com.example/demo/0.0.1-SNAPSHOT/demo-0.0.1-
SNAPSHOT.pom
[JENKINS] Archiving /var/lib/jenkins/workspace/final_project/target/demo-0.0.1-SNAPSHOT.war to com.example/demo/0.0.1-
SNAPSHOT/demo-0.0.1-SNAPSHOT.war
channel stopped
[DeployPublisher][INFO] Attempting to deploy 1 war file(s)
[DeployPublisher][INFO] Deploying /var/lib/jenkins/workspace/final_project/target/demo-0.0.1-SNAPSHOT.war to container
Tomcat 8.x Remote with context finalProject
  Redeploying [/var/lib/jenkins/workspace/final_project/target/demo-0.0.1-SNAPSHOT.war]
  Undeploying [/var/lib/jenkins/workspace/final_project/target/demo-0.0.1-SNAPSHOT.war]
  Deploying [/var/lib/jenkins/workspace/final_project/target/demo-0.0.1-SNAPSHOT.war]
Finished: SUCCESS
```

Figure38:Building through the POM.xml file shows success and deployment to a container also finishes with success

Docker

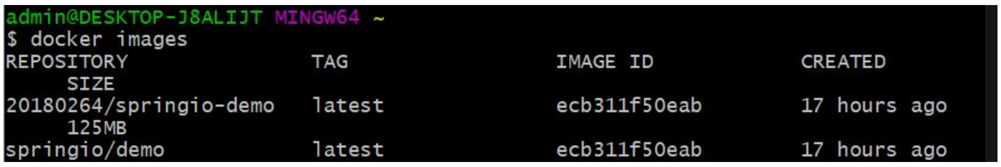
Docker is a software for containerizing other softwares and systems and embedded it into images which can be pulled from the Docker Hub and be used normally like virtual machines but without loading the whole operating system and its layers on the image which make it size doable and works fine the same as if it is working on the developer device.

The Project implemented uses a Dockerfile that uses the .jar in order to make an image of the application on the docker desktop and be able to push it to the Docker Hub.



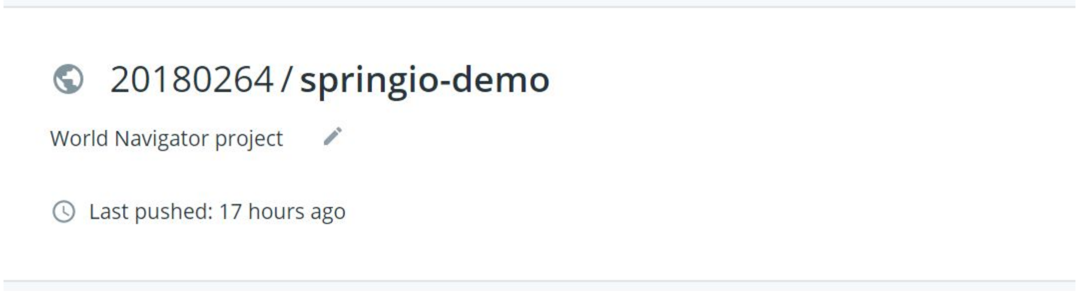
```
FROM openjdk:8-jdk-alpine
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Figure38:Dockerfile contents



```
admin@DESKTOP-J8ALIJT MINGW64 ~
$ docker images
REPOSITORY                TAG                 IMAGE ID            CREATED
     SIZE
20180264/springio-demo    latest              ecb311f50eab        17 hours ago
     125MB
springio/demo             latest              ecb311f50eab        17 hours ago
```

Figure39:Images running on my device



20180264 / springio-demo

World Navigator project

Last pushed: 17 hours ago

Figure40:Image repository that was pushed from the local one and can be fetched and used publicly by any user.

Can be accessed through this link:

https://hub.docker.com/repository/docker/20180264/springio-demo

AWS EC2 Instances

Finally, Amazon provides the cloud services that allows developers to use tools and systems or softwares without the need of downloading it, i.e. all on the web, using its servers.

The project development and deployment was built on AWS EC2 two instances where the first runs a jenkins server on port 8080 of its IPv4 address with other installations like java and maven on it.

This server can be accessed through the URL:

http://100.26.239.54:8080/

The second server is the tomcatx8 server which is used as a container for the project deployed on jenkins with war file, then the war file is being installed on the tomcat server and any edit or new build refresh the tomcat server too so that you can monitor the changes automatically.

The tomcat server on AWS can be accessed through the URL:

http://3.81.47.73:8080/finalProject/mazegame

Conclusion

This assignment was generally about building a console maze game that is played and controlled with commands, the core part of the assignment is not only to implement the game, but to use Object Oriented Design to design it so that to reach a higher level of abstraction and efficiency of the code. We've got to read and learn many things and to obey a collection of rules to benefit as much as possible from the design and make it flexible, readable, extendable, and reusable. We've worked with the Clean Code Principles, Effective Java Principles, SOLID Principles, Design Patterns, Data Structure and Google Java Style all in this project. And since this assignment was big and needed to code a lot and think a lot about the designs trying to make it as simple as possible with high level of abstraction, I really benefited from every single information I've got to learn through the lectures and searching the web for best practices and the different principles and so on, I need to modify the design a bit more, but I believe working on it taught me a lot and polished my Java programming experience, it is now easier for me to think of the design patterns with tips in mind about the Java clean code and SOLID principles.