Java and DevOps training: Week 7

Mazing: A Java console game

Mohammad Abdallah

December 30, 2020

Table of Contents

Objective

This report is written as a documentation for my work in the project called **Mazing**, the project is implemented in Java.

Mazing is a command-line game where you start the game in a certain room, and you gather items and use them to make your way out of the maze on time!
The maze is made up of several rooms connected by locked or unlocked doors.
You win if you make your way out of the maze and reach the end room.
You lose if you decide to quit or restart the game, or if the time needed to win elapses.

The goal of this project was to implement a certain behavior for this game, while keeping the code clean, simple, and easy to read and extend. This report justifies the way the project is implemented, and explains how design decisions were made.

How to play

Before we get to the point of this report, let us first explain the game logic, and how exactly it is played, so you can gain some insight on how it should be implemented

Note: You "choose" a certain command [COMMAND] by writing the name of command "COMMAND_NAME" in the console and clicking enter.

- When the map is chosen, your character starts in a certain room, with a certain amount of gold and a certain orientation direction.
- Each room is defined as a block, with 4 surrounding walls, each wall could be empty, or have one of the 5 wall objects (seller, chest, door, painting, mirror).
- Chests and doors could be locked, and if they are, you could open them with their corresponding keys.
- Chests could have keys, flashlights and gold.
- Sellers buy and sell certain items for a certain amount of gold.
- A painting or a mirror could have a key hidden behind it.
- Time needed, starting gold ,starting room and all other maze features are specified in the chosen map along with the exact structure of the maze.
- The goal of the game is to get your character out of the maze to the end room on time.
- **Choosing the map**: When you first open the game, it will ask you to choose a map.
    - There are two predefined maps; hard and medium, you can choose either one of them
    - You can also choose custom, which is an almost empty map that you can define yourself
    - You can also choose any map name if you already create one with that name in folder "json”

- **Commands**: After choosing the map, the game starts, and you play using the following commands:
    - Navigation:
        - **left**: The character turns left, changes orientation from north to west for example.
        - **right**: The character turns right, changes orientation from north to east for example.
        - **forward**: If the character is facing an open door, he walks through the door to the room behind it.
        - **backward**: If there is an open door behind the character, he walks through the door to the room behind it.
    - Status:
        - **status**: Prints the character's orientation direction, gold count and item
    - Time:
        - **time**: Prints elapsed time since the game started as well as the remaining time you have until you finish.
    - Quit/Restart:
        - **quit**: You quit and lose the game
        - **restart**: You restart and lose the game
    - Looking
        - **look**: Prints the type of wall object in front of you (chest, door, painting, mirror, seller, empty wall)

- Checking:
    - **check [wallObject]**: If [wallObject] is in front of your character, and it can be checked, it will be checked. Checkable wall objects->(chest, door, painting, mirror)
        - **check chest**: If chest is locked, it will print the name of the key needed to unlock, if it is open and has not been looted, it will be looted.
        - **check door**: It will print the door status (locked/unlocked), and the name of the key needed to unlock if it is locked.
        - **check mirror/painting**: It will acquire the key behind the painting/mirror if there is one.
- Using Items:
    - **use [itemName]**: If [itemName] is an item with your character, and is usable, it will be used. Usable item->(key, flashlight)
        - **use key[N]**: If there is a door/chest that is locked/unlocked with the key[N] in front of your character, it will unlock/lock it. N is the keyId, key[N] becomes key5 for example.
        - **use flashlight** Turns the flashlight on/off, If the current room is not lit, it will become lit
- Switch Lights:
    - **switchlights**: If the current room has lights, it will turn the lights on/off.

- Trading:
  - **trade**: If there is a seller in front of your character, it will enter trading mode with this seller, after which the following list of commands will be available:
    - **buy [itemName]**: will buy the item corresponding to [itemName] from the seller if it is listed in the seller's item, and your character has enough gold
    - **sell [itemName]**: will sell the item corresponding to [itemName] from the seller, if it is listed in the seller's item, and your character has it
    - **list**: will list the seller's item again
    - **end**: will exit the trading mode and go back to accepting main commands
- Help:
  - **help**: choosing [help] will list all of the available commands and describe their usage

How to define a new map

This is an important subject to the gameplay experience, and you should understand it in order to understand some of the design decisions explained later in this report.

In the folder called "json", in the root directory, there is already two complete predefined maps (hard.json, medium.json), but what if you want to play a new map? Maps are designed using JSON, which uses a human readable text to define objects. To define a new map, you can either use the file (custom.json) inside of the "json" folder, or define a new file with a new unique name, which you can then write when the game asks you to choose a map.
In the json file, you will define exactly how the map is constructed using the following json objects:

- jsonMap:

```
{
  "goldCount":0,
  "direction":"east",
  "item":[],
  "secondsNeeded": 120,
  "rooms":
  [
  ]
}
```

- goldCount: **number**; initial gold count the character should have
- direction: **string**; initial direction for character(east, west, north, south)
- item: **list of jsonItem(jsonKey,jsonFlashlight)**; initial item that the character should have
- secondsNeeded: **number**; maximum time in seconds of how much this map should
- rooms: **list of jsonRoom**; all the rooms in this map with their exact specifications

- jsonRoom:

```json
{
  "id":1,
  "east":
  {
  },
  "west":
  {
  },
  "north":
  {
  },
  "south":
  {
  },
  "isLit":true,
  "hasLight":true
}
```

- id: **number**; Room id, used to identify the room
- east: **jsonWall(jsonChest,jsonDoor,jsonPainting,jsonMirror,jsonSeller)**; Wall object on the east of this room, with its exact specification, set to null if the wall is empty
- west: **jsonWall(jsonChest,jsonDoor,jsonPainting,jsonMirror,jsonSeller)**; Wall object on the west of this room, with its exact specification, set to null if the wall is empty
- north: **jsonWall(jsonChest,jsonDoor,jsonPainting,jsonMirror,jsonSeller)** ; Wall object on the north of this room, with its exact specification, set to null if the wall is empty
- south: **jsonWall(jsonChest,jsonDoor,jsonPainting,jsonMirror,jsonSeller)** ; Wall object on the south of this room, with its exact specification, set to null if the wall is empty
- isLit: **boolean**; are the lights in this room on?
- hasLights: **boolean**; does this room have lights?

- jsonPainting:

```json
{
  "type": "painting"
}
```

  - behind: **number**; keyId of the key behind this painting, set to null if there is no key

- jsonMirror:

```json
{
  "type": "mirror",
  "behind": 3
}
```

  - behind: **number**; keyId of the key behind this mirror, set to null if there is no key

- jsonSeller:

```json
{
  "type": "seller",
  "items":
  [

  ]
}
```

  - items: **list of jsonItem(jsonKey,jsonFlashlight)**; item on sale by this seller

- jsonChest:

```json
{
  "type": "chest",
  "locked": true,
  "keyId": 6,
  "inside": [

  ]
}
```

  - locked: **boolean**; is this chest locked?
  - keyId: **number**; keyId of the key used to lock this chest, no need to specify if chest is unlocked
  - inside: **list of jsonItem(jsonKey,jsonFlashlight)**; item inside this chest

- jsonDoor:

```json
{
  "type": "door",
  "locked": true,
  "keyId": 3,
  "from": 1,
  "to": 4
}
```

  - locked: **boolean**; is this door locked?
  - keyId: **number**; keyId of the key used to lock this door, no need to specify if door is unlocked
  - from: **number**; roomId of one of the rooms this door is connected to, other than **to** (does not actually matter if it is from or to)
  - to: **number**; roomId of one of the rooms this door is connected to, other than **from** (does not actually matter if it is from or to)

- jsonGold:

```json
{
  "type": "gold",
  "count": 10
}
```

  - count: **number**; Number of golds this gold item represents

- jsonKey:

```json
{
  "type": "key",
  "keyId": 1,
  "price": 5
}
```

  - keyId: **number**; Key id of this key
  - price: **number**; The worth of this key in gold, useful for sellers, no need to specify if 0 or if this key is not part of a seller's list

- jsonFlashlight:

```json
{
  "type": "flashlight",
  "price": 10
}
```

  - price: **number**; The worth of this flashlight in gold, useful for sellers, no need to specify if 0 or if this flashlight is not part of a seller's list

- Notes:

- The starting room should have a room id of **1**
- The end room should have a room id of **-1**, and should logically not have anything but the one door connecting it to the rest of the map
- You can delete the *hidden* property (or specify it as null) from a painting/mirror if it does not have a key
- You can delete the *locked* property (or specify it as null) from a door if it is unlocked
- You can delete the *east*/*west*/*north*/*south* property (or specify it as null) from a room if it has an empty wall there

All these json objects are defined in "objects.json" inside of the "json" folder, to see real examples of how a json map file would look like after it is constructed, look at "medium.json" and "hard.json".

How to read this report

We will first provide an explanation for how all the project works, a short story that goes from when a user types a command until he gets back a response, all in limited details. We will then go through the same story again with some examples and full details where we will point out why some features are implemented the way they are implemented and discuss their performance and degree of simplicity.

Project Structure: How does at all work?

The main class in this project is the **Starter** class, which, when you start, creates a new **Game** instance and then deliver it to the static method ***initializeGame()*** in the **Console** class.

The **Game** class stores the current status of the game as instances of other classes, and is used throughout the project as a carrier for these instances to other classes.

initializeGame(), as the name suggests, sets the initial state of all the instances in **Game** from an external *JSON* file representing the map using the **GameMap** interface and its implementing classes.

It then prepares it be ready for a command from the user from the command line using static method ***executeMainCommand()*** which can be called whenever you want the user to choose a main command

executeMainCommand() then gets the user command as a string, and then analyses this string and creates a new **Command** object based on this command and executes it.

There is an interface called **Command** which describes how a command should be executed using methods that should be declared in its implementing classes, and there are two abstract classes implementing this interface and describing some common behavior: **MainCommand** and **TradingCommand**.

Each command that should be part of the gameplay is implemented into a class that extends either **MainCommand** or **TradingCommand** and overrides its abstract methods.

Most of the commands operate on the structure facing the character in the room, this structure could be an instance of a **Chest, Painting, Mirror, Seller, Door** or it could be an empty wall. These structures are handled as if they are all wall types, and they all extend the abstract class **Wall** and override its abstract methods that provide way to access and change the state of a wall. The commands talk to the wall facing the character through the **Game** class which saves instances for both classes **Room** and **Character**.

Room stores the current state of the room including the state of each wall on its 4 sides, and **Character** stores the current state of the character including the current direction of the character, which can also be changed by some other commands.

Some other commands perform on items stored in the **Character** as instances of the classes **Key, Gold,** or **FlashLight**, which all extend the abstract class **Item.** These items, when used (aka when a command is performed on them) access methods on other instances, probably resulting in a change of the game status.

Each command, when executed, returns a query to the user as **Response** object that has a **ResponseType** object. In our cases this response object is only made to be printed, since it is a command line game where everything gets directly printed.

Each command then prepares the console to accept another command either from the family of **TradingCommand** or **MainCommand** by executing either ***executeMainCommand()*** or ***executeTradingCommand()***

Details

```java
public interface Command {
  Game getGame();

  void setGame(Game game);

  Response getResponse();

  void setResponse(Response response);

  default void execute() {
    applyEffect();
    printResponse();
    executeNext();
  }

  void applyEffect();

  default void printResponse() {
    System.out.println(getResponse());
  }

  void executeNext();
}
```

This is the code for the **Command** interface, the main command in this interface is the default command ***execute(),*** *this command is what gets executed.*

So, all commands work in this way, they ***applyEffect(), printResponse(),*** and then prepare the console for the next command (***executeNext()***), but they all do it in different ways.

Notice that ***printResponse()*** is another default method, it prints the **Response** instance that will be a local variable in implementing classes after they override the **setResponse()** method. **setResponse()** should be called inside **applyEffect()** if part of the effect of the command is returning a query.

This design allows room for a skeletal class implementation, and this is exactly what was done.

There are two skeletal abstract classes: **MainCommand** and **TradingCommand** that implement the **Command** interface. The Main difference between these two abstract classes is that each one provides a different implementation for the static method **getCommand()** that creates and returns a command based on a user command string, this is used in the console by both **executeMainCommand()** and **executeTradingCommand().**

Another different is that each one has a default implementation of **executeNext()** method. **TradingCommand** calls **executeTradingCommand()** while **MainCommand** calls **executeMainCommand()**, this could be overridden in some cases, and it indeed is in our case.

This is how **MainCommand** looks like:

```java
public abstract class MainCommand implements Command  {
  private Game game;
  private Response response;

  public static MainCommand getCommand(String command,String arg){
    switch(command.toLowerCase()){
      case "look" -> {return new LookCommand();}
      case "check" ->{return new CheckCommand(arg);}
      case "status" ->{return new PlayerStatusCommand();}
      case "time"->{return new TimeCommand();}
      case "use" ->{return new UseCommand(arg);}
      case "trade"->{return new TradeCommand();}
      case "quit" ->{return new QuitCommand();}
      case "restart"->{return new RestartCommand();}
      case "switchlights"->{return new SwitchLightsCommand();}
      case "left"->{return new LeftCommand();}
      case "right"->{return new RightCommand();}
      case "forward"->{return new ForwardCommand();}
      case "backward"->{return new BackwardCommand();}
      case "help"->{return new MainHelpCommand();}
      default -> {return new NoMainCommand();}
    }
  }

  @Override
  public Game getGame() {
    return game;
  }

  @Override
  public void setGame(Game game) {
    this.game = game;
  }

  @Override
  public void executeNext(){
    Console.executeMainCommand(game);
  }

  @Override
  public Response getResponse() {
    return response;
  }

  @Override
  public void setResponse(Response response){
    this.response=response;
  }
}
```

Notice the switch statement, which is an unavoidable smell that we have to bear, but at least it is buried down in this abstract class and not repeated. We tried to do what uncle bob advised here:

“It’s hard to make a small switch statement. Even a switch statement with only two cases is larger than I’d like a single block or function to be. It’s also hard to make a switch statement that does one thing. By their nature, switch statements always do N things. Unfortunately, we can’t always avoid switch statements, but we can make sure that each switch statement is buried in a low-level class and is never repeated. We do this, of course, with polymorphism.” -Robert Martin, Clean Book

Let us have a look on 2 of **MainCommand** children

1. UseCommand

```java
public class UseCommand extends MainCommand {
  String itemName;

  public UseCommand(String itemName) {
    this.itemName = itemName;
  }

  @Override
  public void applyEffect() {
    Item item = Item.getItemFromList(itemName, getGame().getCharacterItems());
    setResponse(item.use(getGame()));
  }
}
```

2. TradeCommand

```java
public class TradeCommand extends MainCommand {

  @Override
  public void applyEffect() {
    setResponse(getGame().getFacingWall().list());
  }

  public void executeNext() {
    if (isResponseInvalid()) Console.executeMainCommand(getGame());
    else Console.executeTradingCommand(getGame());
  }

  private boolean isResponseInvalid() {
    return getResponse().getType() == ResponseType.INVALID;
  }
}
```

Notice that **UseCommand** only overrides the **applyEffect()** method, while **TradeCommand** overrides both **applyEffect()** and **executeNext()**, and that is because the trade command takes us from the main menu to the trading menu.

Notice also how the **applyEffect()** sets the response to what the user should see printed on the command line by using **item.use(getGame) (**after getting the item based on the string from user) inside **setResponse()**, which raises 3 important questions:

- How do we get the item? And what happens if the string does not represent an item?

We get the item by using the static method **getItemFromList()** in the abstract class **Item,** let us have a look at this method:

```java
public static Item getItemFromList(String itemName, List<Item> items) {
  for (Item item : items) {
    if (item.name().equalsIgnoreCase(itemName)) {
      return item;
    }
  }
  return NoItem.getInstance();
}
```

This method is stateless, you should provide both the *itemName* string and the list of items to get from.

A small note, this method used to use the **toString()** to compare with *itemName*, but using the toString() method for code flow is a code smell, so I defined a new method called **name()** for that purpose.

So, we get the list of items that we have stored in the **Character** instance, and then we provide to this method along with the *itemName* string from the user input and look for this item in that list, and return it if it is not there.

What if we do not have this item in the list? Normally you would think of two solutions, return a null, or raise an exception.

“Use exceptions only for exceptional conditions”

“Return empty collections or arrays, not nulls”

These 2 quotes are from the famous Effective Java book, one could argue that this indeed is an exceptional condition, but I think that raising an exception only to catch it later and return some response in a program that all its logic is returning responses based on status is making the exception part of the code flow, which is a code smell.

So, what I did here is that I followed the Special Case Pattern. I made a class called **NoItem** that extends the **Item** abstract class, this **NoItem** has methods that describe what happens when you use an item that you do not have, Classical Polymorphism.

- What happens inside **item.use()?**

Use(Game game) is a method defined inside of the abstract class **Item**, and overridden based on need by each item, see how it is implemented in some cases:

Item

```java
public Response use(Game game) {
  return new Response(ResponseType.INVALID, "This item is unusable");
}
```

Key

```java
@Override
public Response use(Game game) {
  Wall facing = game.getFacingWall();
  return facing.toggleWithKey(this);
}
```

FlashLight

```java
@Override
public Response use(Game game) {
  game.getCharacter().setFlashLightOn(!game.getCharacter().isFlashLightOn());
  return new Response(
      ResponseType.SUCCESS,
      "Your flashlight is " + (game.getCharacter().isFlashLightOn() ? "ON" : "OFF"));
}
```

NoItem

```java
@Override
public Response use(Game game) {
  return new Response(
      ResponseType.INVALID, "How come you want to use an item that you do not have?");

}
```

In the abstract class, we defined the default behavior, normally we want and item to be unusable, unless we specify otherwise, and in each case, we specified how exactly we wanted to use the item, and returned the response after using it.

Notice **NoItem**'s implementation of the **use()** command tells us that we do not have the item if we tried to use it, that is what we were talking about above. It is also worth mentioning that **NoItem** is a singleton class, it does not make sense to have multiple **NoItem** instances so we could only have one **NoItem** instance. There are multiple singleton classes in the project for reasons similar to this reason.

NoItem:

```java
public class NoItem extends Item {

  private static NoItem instance;

  private NoItem() {}

  public static synchronized NoItem getInstance() {
    if (instance == null) {
      instance = new NoItem();
    }
    return instance;
  }

  @Override
  public Response use(Game game) {
    return new Response(
        ResponseType.INVALID, "How come you want to use an item that you do not have?")
;
  }

  @Override
  public Response buy(Game game) {
    return new Response(ResponseType.INVALID, "The seller does not have what you desire
");
  }

  @Override
  public Response sell(Game game) {
    return new Response(ResponseType.INVALID, "The seller does not want to buy this ite
m");
  }

  @Override
  public ItemType getType() {
    return ItemType.NOITEM;
  }
}
```

- What about CQS?

CQS, Command Query Separation states: Every method should either be a *command* that performs an action, or a *query* that returns data to the caller, but not both.

CQS could be looked at from the point of SOLID principles as SRP, Single Responsibility Principle, each component should have one responsibility, when a method both change status and return a response, it is considered a violation of both CQS and SRP.

It is obvious in our case that our commands are all command and query fashion which is a clear violation to the CQS principle.

I think this is justified, because it is a command line application.

If it was not, each command would print a message after successful or failing, the problem in command line applications are that messages, queries and responses all mean the same thing; **System.out.println().**

I should probably mention that what applies on items, also applies on walls. There is an abstract **Wall** class that has underneath it the classes **Chest, Seller, Door, Painting, Mirror** and **Empty**. They all override method from **Wall** that the commands from the interface **Command** rely on.

It is also worth mentioning that both **Door** and **Chest** are constructed using the Builder Design Pattern. This is because they both require a so much parameters to be instantiated. For example, a door should have both connecting doors ids, as well as a Boolean describing if it is locked or unlocked, and if it is locked, the key id that should be used to open it. Doing this with a constructor would require a lot of arguments, which is a code smell.

This is how the **Door** class is implemented using the Builder Design Pattern:

```java
public class Door extends Wall {

  public static List<Door> doors = new ArrayList<>();
  private final Key key;
  int[] connectingRoomsId;
  private boolean isLocked;

  public static class Builder {

    int[] connectingRoomsId;
    private boolean isLocked;
    private Key key;

    public Builder(int from, int to) {
      connectingRoomsId = new int[] {from, to};
      isLocked = false;
      key = NoKey.getInstance();
    }

    public Builder lockedWithKey(int keyId) {
      isLocked = true;
      this.key = new Key(keyId);
      return this;
    }

    public Door build() {
      Door newDoor = new Door(this);
      int index = doors.indexOf(newDoor);
      if (index != -1) {
        return doors.get(index);
      }
      doors.add(newDoor);
      return newDoor;
    }
  }

  public Door(Builder builder) {
    connectingRoomsId = builder.connectingRoomsId;
    isLocked = builder.isLocked;
    key = builder.key;
  }
```

Bounded Wildcards

“Use bounded wildcards to increase API flexibility”

This is one of the items in Effective Java, I had it in its own section because it helped me learn something that I have never understood before, and I had a great use for it in my project.

We said that the **Character** class stores the status of the character and provides method to change it, one of the local variables that make this status is called *items* and it is a list that contains objects of type **Item**, and it represents the items that the character currently has.

To provide ways to easily interact with these items, along with many methods, I added the method **addItems()** which takes a list of items, loops over them and add them one by one to the character’s current items using another method called **addItem()**, basic stuff, but let us look at the implementation I currently had.

```java
public void addItems(List<Item> items) {
  for (Item item : items) {
    addItem(item);
  }
}
```

This implementation worked, there was no problem, whenever there was a chest that had multiple items, it returned a **List<Item>** that has all the items, and it worked like a charm.

Now imagine that in the future, we decided to add another **Wall** structure, let us call it **Drawer**, let us say that this drawer can have multiple keys that could return whenever checked using the check command, we would probably just return them as **List<Key>**, and we are totally right, they are all keys, and keys are item. LSP states that we can always substitute a superclass by its subclass.

“parameterized types are *invariant*. In other words, for any two distinct types `Type1` and `Type2`, `List<Type1>` is neither a subtype nor a supertype of `List<Type2>`”- Joshua Block, Effective Java

This means that **List<Key>** could never be a subtype of **List<Item>**, and therefore if we tried to use the **addItems()** method on the **List<Key>** returned by the drawer, we would get the following error: incompatible types: List<Key> cannot be converted to List<Item>.

The fix to this issue, as the quote at the top of this section suggests is to use bounded wild cards, transforming the method to the following implementation:

```java
public void addItems(List<? extends Item> items) {
  for (Item item : items) {
    addItem(item);
  }
}
```

GameMap

```java
public interface GameMap {

  default void setUp(Game game){
    setUpRooms(game);
    setUpCharacter(game);
    setUpStopWatch(game);
  }

  void setUpRooms(Game game);

  void setUpCharacter(Game game);

  void setUpStopWatch(Game game);
}
```

We talked all about how a command gets executed, but in our short story we also discussed how a game is initialized using the **GameMap** class, so how does this exactly work?

GameMap interface gets implemented only by the class **JsonGameMap,** but an interface exists so we leave space for another way to setup the map, maybe using an XML file, which would make another class called **XMLGameMap**, this is all a shot into trying to make the code more adhering to the OCP.

JsonGameMap gets defined by the static method **setUpMap()** inside the **Console** class, as part of the initializing done after we call **initializeGame()** in the **Starter** class, and then the method **setup()** of the **JsonGameMap** gets called inside that same **setUpMap()** method.

We could easily conclude what happens inside **setUpRooms(), setUpCharacter()** and **setUpStopWatch()**. It initializes all of the instances of game using what is already defined inside of the JSON file of the map that gets chosen. It takes information from the JSON file and creates object based on these information.

Translating the information from JSON and reformatting it in a way that allows it to be processed by Java could be a complex job, and that is I used the library "json-simple" from google to do this. "Know and use the libraries"

The classes that I intensively had to use by this library are **JSONObject** and **JSONArray**, and they really made my life easier, the only problem is that each time you use **JSONObject** to get an instance based on a key, it returns an instance of type Object, which I then have to cast it to the type of object that I am expecting, which could be exhausting. It fells the whole file with if statements.

To solve this problem, I wrapped **JSONObject** with **JSONObjectWrapper**, and **JSONArray** with **JSONArrayWrapper**.

In **JSONObjectWrapper** I defined different methods to get an object from key, based on the type of object; **getString()**, **getJsonObject(), getInt(), getBoolean(), getJsonArray**.

And in **JSONArrayWrapper** I defined only one method, **getList()** which returns an ArrayList of the JSONObjectWrapper instances that it creates from the **JSONObject** inside the original **JSONArray**.