World Navigator Report

Summary

This report is for World Navigator Game, created by **Samer Rawashdeh** as a solution for the Atypon's internship World Navigator assignment

This report will show the knowledge and experience I gained so far from Atypon's internship.

This version of the code, contains 50+ files with 2396 source code lines.

This report contains 5919 words and it' takes about 16 minutes to read.

Topics Covered

1. How my code satisfies the clean code principles according to Robert's '"Clean Code" book
2. How my code satisfies the effective Java code principles according to Joshua's "Effective Java" book
3. Discuss the design patterns I used to implement my solutions and why
4. How my code satisfies the SOLID principles?
5. How my code satisfies Google (or Twitter) styling guide
6. Concurrency issues in my code and how I dealt with it
7. Data structures I used and why

How my code satisfies the clean code principles according to "Clean Code" book

Chapter 2 **Meaningful Names** :

"Choosing good names takes time but saves more than it takes.", I find this quote from Robert's Book very important every time I refactored my code, sometimes miss naming classes, variables or functions could complicate thigs more than it should be

- **USE INTENTION-REVEALING NAMES**
    - Every part of my code satisfy this point, and I find it easy to achieve it, examples

```
private Direction facingDirection;
private Light heldLight;
private Room currentRoom;
```

```
private Room startRoom;
private Room goalRoom;
private int timeToFinish;
private int initialGold;
```

- **AVOID DISINFORMATION**
    - Satisfying this point was easy too, because World Navigator game is a simple game that doesn't have complex names
- **MAKE MEANINGFUL DISTINCTIONS**
    - The first version of my code didn't satisfy this point, this book helped me spot the problem and fix it

      Before the fix :

      ```
      RoomObject trader0 = getPlayer().facingRoomObject();

      if (!(trader0 instanceof Trader)) {
        System.out.println("You are not facing a trader or a seller! selling
      is canceled...");
        return;
      }

      Trader trader = (Trader) trader0;
      ```

      After the fix

      ```
      RoomObject roomObject = getPlayer().facingRoomObject();

      if (!(roomObject instanceof Trader)) {
        System.out.println("You are not facing a trader or a seller! selling
      is canceled...");
        return;
      }

      Trader trader = (Trader) roomObject;
      ```

- **CLASS NAMES**
    - "Classes and objects should have noun or noun phrase names like `Customer`, `WikiPage`, `Account`, and `AddressParser`. Avoid words like `Manager`, `Processor`, `Data`, or `Info` in the name of a class. A class name should not be a verb."
    - My code fulfill this point very good, we can see it every where examples : `Key`, `Lockable`, `LootHider`, `Painting` and more
- **METHOD NAMES**
    - "Methods should have verb or verb phrase names like `postPayment`, `deletePage`"
    - Examples where my code complied with this point : `getGoldBalance`, `giveItem`, `addItem`, `addDefaultCommands`, `toggle`

Chapter 3 **Functions**:

- **SMALL!**
  - The first rule of functions is that they should be small, and the second rule is that they should be smaller than that

    Pascal said "I am sorry this letter is so long, I didn't have time to make it smaller"

    making a small functions is not an easy task, but I think I did a good job regards to this rule, examples :

```
public void holdLight(Light light) {
  if (light == null) {
    throw new NullPointerException("light value is null!");
  }
  if (heldLight!= null) {
    giveItem(heldLight);
   }
    heldLight = light;
}
```

```
@Override
public Loot getLoot() {
  return keyLoot;
}
```

```
public void lootGold(Player player) {
  if (player == null)
    throw new NullPointerException("player value is null!");
  player.giveGold(goldLoot);
  System.out.println(goldLoot + " gold was acquired");
  goldLoot = 0;
}
```

```
public void loot(Player player) {
  lootGold(player);
  lootItems(player);
}
```

I had some difficulties applying this rule for all my code, since some of them need to have switch statements and while loops, example :

```
```

```
public static Direction oppositeDirection(Direction direction) {
  if (direction == null) {
    throw new NullPointerException("direction value is null!");
  }

  switch (direction) {
    case NORTH:
      return Direction.SOUTH;
    case EAST:
      return Direction.WEST;
    case SOUTH:
      return Direction.NORTH;
    default: // this default must be Direction.WEST
      return Direction.EAST;
  }

}
```

- **DO ONE THING**
  - This rule is well known, and it makes things easier while you advance in the code, it makes functions smaller and easer to deal with, examples :

```
public void switchRoomLight() {
  if (currentRoom.isLightSwitchAvailable()) {
    currentRoom.switchLightSwitch();
  } else {
    System.out.println("no light switch to switch");
  }
}
```

```
public void useHeldLight() {
  if (heldLight == null) {
    System.out.println("I am not holding a light source to use it!");
  } else {
    heldLight.toggle();
  }
}
```

- **ONE LEVEL OF ABSTRACTION PER FUNCTION**
  - **Reading Code from Top to Bottom: *The Stepdown Rule***

    "We want the code to read like a top-down narrative. We want every function to be followed by those at the next level of abstraction so that we can read the program" Robert

I achieved this rule in many places in my code, I used **Template Method Design Pattern**, we will talk about it later in details, example :

```
public final void sellItem(Trader trader, Item item) {
  if (trader == null) {
    throw new NullPointerException("trader value is null!");
  }
  if (item == null) {
    throw new NullPointerException("item value is null!");
  }
  if (!canSellItemTo(trader, item)) {
    cancelSelling();
    trader.cancelBuying();
    return;
  }
  proceedSelling(item);
  trader.proceedBuying(item);
}
```

- **USE DESCRIPTIVE NAMES**
  - "A long descriptive name is better than a long descriptive comment" Robert

    my longest function name is `initialValuesToRestart()`, I did use a descriptive names for my functions for other parts too

- **FUNCTION ARGUMENTS**
  - The maximum number of arguments in my functions is 2, most of my functions have zero or one argument, some of them has 2 arguments but it's rare, although the 2 arguments are acceptable by Robert's Book, examples :

```
public void addGold(int goldLoot) {
  if (goldLoot < 0) {
    throw new IllegalArgumentException("gold amount is negative!");
  }

  this.goldLoot += goldLoot;
}
```

```
public void useKey(Key key) {
  if (facingRoomObject() instanceof Lockable) {
    useKey((Lockable) facingRoomObject(), key);
  } else {
    System.out.println("What you are facing doesn't need a key!");
  }
}
```

```
private void useKey(Lockable lockable, Key key) {

  if (lockable == null) {
    throw new NullPointerException("lockable value is null!");
  }

  if (key == null) {
    throw new NullPointerException("key value is null!");
  }

  lockable.useKey(key);
}
```

Technically, a constructor usually is a method, I have a constructor with more than 3 arguments, it's actually the **Map** class and it has 5 arguments , and I solved that using **Builder Design Pattern **, more on that latter...

```
  public Map(Room startRoom, Room goalRoom, int timeToFinish, int
initialGold,
    Direction playerFacingDirection) {...}
```

```
public class MapBuilder {
  ...
  public Map buildMap() {
    return new Map(startRoom, goalRoom, timeToFinish, initialGold,
playerFacingDirection);
  }
}
```

- **Flag Arguments**

  "Flag arguments are ugly" Robert, my code is free of boolean arguments, a good way to solve boolean argument is to try and extract more functions from the function with boolean arguments

Chapter 7 **Error Handling** :

- **USE EXCEPTIONS RATHER THAN RETURN CODES**
  - "it is better to throw an exception when you encounter an error. The calling code is cleaner. Its logic is not obscured by error handling.", my code obey this rule so that no unexpected action happen, every error must be handled and fixed and not ignored, example :

```
public final void sellItem(Trader trader, Item item) {
  if (trader == null) {
    throw new NullPointerException("trader value is null!");
```

```
    }
    if (item == null) {
      throw new NullPointerException("item value is null!");
    }
    if (!canSellItemTo(trader, item)) {
      cancelSelling();
      trader.cancelBuying();
      return;
    }
  }
```

- **DON’T RETURN NULL**
  - Not a single function in my code return null, and this could help so much in exception handling, since as a function caller, I don't need to check if the return value of the called function is null or not
- **DON’T PASS NULL**
  - In my early versions of my code, I used to do this, but after reading from this Book I changed that

    I used to do that in my constructors, but not any more, examples :

    Before the fix :

    Having this key variable equals null was part of my code logic, and it meant that Painting doesn't hide a key

    ```
    public Painting(Key keyLoot) {
      this.keyLoot = new OneKeyLoot(keyLoot);
    }
    ```

    After the fix :

    ```
     public Painting() {
       keyLoot = new OneKeyLoot();
     }
     public Painting(Key keyLoot) {

       if (keyLoot == null) {
         throw new NullPointerException("key value is null!");
       }
       this.keyLoot = new OneKeyLoot(keyLoot);
     }
    ```

How my code satisfies the effective Java code principles according to Joshua's “Effective Java” book

- Item 2 : **Consider a builder when faced with many constructor parameters**
    - Map class used this Design Pattern, since to create a map you need 5 parameters, example :

```
public class MapBuilder {
  private Room startRoom;
  private Room goalRoom;
  private int timeToFinish;
  private int initialGold;
  private Direction playerFacingDirection;

  public MapBuilder setStartRoom(Room startRoom) {...}

  public MapBuilder setGoalRoom(Room goalRoom) {...}

  public MapBuilder setTimeToFinish(int timeToFinish) {...}

  public MapBuilder setInitialGold(int initialGold) {...}

  public MapBuilder setPlayerFacingDirection(Direction
playerFacingDirection) {...}

  public Map buildMap() {
    return new Map(startRoom, goalRoom, timeToFinish, initialGold,
playerFacingDirection);
  }
}
```

- Item 3 : **Enforce the singleton property with a private constructor or an enum type**
    - I used this technique, inside GameDriver class, the class that is responsible to run and creating new games :

```
private static GameDriver gameDriver = new GameDriver();
private ArrayList<MapLoader> maps;
private ArrayList<Mode> modes;

private GameDriver() {
  maps = new ArrayList<MapLoader>();
  modes = new ArrayList<>();
}

public static GameDriver getInstance() {
  return gameDriver;
}
```

Inside Main class :

```
GameDriver game = GameDriver.getInstance();

OnePlayerMode gameControllerOnePlayer = new OnePlayerMode();

game.addGameMode(gameControllerOnePlayer);

game.start();
```

- Item 5 : **Avoid creating unnecessary objects**
  - There is an example for this in the book and it's not to do this :

```
String s = new String("stringette");  // DON'T DO THIS!
```

and to do this insisted :

```
String s = "stringette";
```

this is a basic approach to create strings in any application, I did follow it and I followed another approach to achieve this rule by using **Proxy Design Pattern**, we can find that inside MapLoader class

```
public class MapLoader {
  private Map map;
  private File file;

  public MapLoader(File file) {
    if (file == null) {
      throw new NullPointerException("file value is null!");
    }
    this.file = file;
  }

  public Map getMap() {
    load();
    return map;
  }

  public String getName() {
    return file.getName();
  }

  private void load() {//proxy pattern
```

```
    if (map != null) {
      return;
    }

    try {
      FileInputStream fileInputStream = new FileInputStream(file);
      ObjectInputStream objectInputStream = new
ObjectInputStream(fileInputStream);
      map = (Map) objectInputStream.readObject();

    } catch (IOException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }

  }


}
```

as you can see, we just create one map from the serialized File

```
if (map != null) {
  return;
}
```

this is the magical part

- Item 6 : **Eliminate obsolete object references**
  - To achieve no memory leak or memory loitering in the code, object elimination must be done in the right way

```
  private ArrayList<Item> itemList;
```

Adding and removing from this ArrayList is done in a safe way

```
public void addItem(Item item) {
  if (item == null) {
    throw new NullPointerException("item value is null");
  }

  itemList.add(item);
}

public void removeItem(Item item) {
```

```
    if (item == null) {
      throw new NullPointerException("item value is null");
    }

    itemList.remove(item);
  }
```

- Item 8 : **Obey the general contract when overriding equals**
    - I did override equals, and doing it right is not easy, but with the help of Intellij IDE and some tips from the book it becomes easier, examples :

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

```
@Override
public boolean equals(Object o) {
  if (this == o) {
    return true;
  }
  if (o == null || getClass() != o.getClass()) {
    return false;
  }
  ItemStatus that = (ItemStatus) o;

  for (Item item : that.itemList) if (!itemList.contains(item)) return
false;

  return true;
}
```

- Item 9 : **Always override hashCode when you override equals**
    - I did this so we don't have a violation of the general contract for `Object.hashCode`, which will prevent our class from functioning properly in conjunction with all hash-based collections, including `HashMap`, `HashSet`, and `Hashtable`, example :

```
@Override
public int hashCode() {//this hasCode() is for ItemStatus class
  int hash = 31;
  for (Item item : itemList) hash = ((hash + item.hashCode()) * 31) %
Integer.MAX_VALUE;

  return hash;
}
```

- Item 13 : **Minimize the accessibility of classes and members**
  - "The single most important factor that distinguishes a well-designed module from a poorly designed one is the degree to which the module hides its internal data and other implementation details from other modules" Joshua's “Effective Java” book

    In this current version there is a 50+ classes, non of them have a public attributes, all of them have a private attributes, but there is one class with protected attributes and it's **Mode ** class, it has a **Map** object witch is protected, I find this okay since Map class is immutable (no setters)

    examples :

```
public abstract class Mode {

  protected Map map;
  private ArrayList<Command> commands;
  private boolean isFinished;
  private String serializedMap; // store original map
  ...
}
```

```
public abstract class Trader implements Serializable {
  private ItemStatus itemStatus;
  private GoldStatus goldStatus;
  ...
}
```

```
public class GameDriver {
  private static GameDriver gameDriver = new GameDriver();
  private ArrayList<MapLoader> maps;
  private ArrayList<Mode> modes;
  ...
}
```

- Item 14 : **In public classes, use accessor methods, not public fields**

- A basic rule that every programmer must obey, example :

```
public class ItemStatus implements Serializable {
  private ArrayList<Item> itemList;
  public ItemStatus() {
    itemList = new ArrayList();
  }

  public void addItem(Item item) {
    if (item == null) {
      throw new NullPointerException("item value is null");
    }

    itemList.add(item);
  }

  public void removeItem(Item item) {
    if (item == null) {
      throw new NullPointerException("item value is null");
    }

    itemList.remove(item);
  }
  public Item getItemByIndex(int index) {
    if (isValidIndex(index) == false) {
      throw new IndexOutOfBoundsException("This is not a valid
index!");
    }
    return itemList.get(index);
  }
...
}
```

- Item 18 : **Prefer interfaces to abstract classes**
  - Abstract classes and interfaces in Java are mechanisms for defining a type that permits multiple implementations, sometimes abstracting solve duplication problems, but we don't really face that all the time, the problem with abstract classes that we can't extend two of them for the same class, but that is not the case for interfaces, we can do as much as we need, examples :

```
public interface Command {
  void execute();
  String name();
}
```

```
public interface Checkable {
  String check();
}
```

```
public interface RoomObject {
  String look();
}
```

```
public interface LootHider {
  Loot getLoot();
  void loot(Player player);
}
```

```
public abstract class Entryway extends Lockable implements RoomObject,
Checkable {...}
```

- Item 19 : **Use interfaces only to define types**
    - All of the Interfaces that are implemented in the code are used to only define types

      '*constant interface*' is a type of interfaces that fails this rule, and I don't use it here
- Item 23 : **Don’t use raw types in new code**
    - "As mentioned throughout this book, it pays to discover errors as soon as possible after they are made, ideally at compile time. In this case, you don’t discover the error till runtime, long after it has happened, and in code that is far removed from the code containing the error."

      Using raw types in code, is a good way to waste time searching through the code base looking for the method that caused `ClassCastException`, examples were i didn't make this mistake :

      ```
      private EnumMap<Direction, Room> nextRoom;
      ```

      ```
      private ArrayList<Item> itemList;
      ```
- Item 25: **Prefer lists to arrays**
    - I didn't use arrays in my code, since using List is a good way to find out about mistakes inside my code at compile time, rather than runtime
- Item 33: **Use EnumMap instead of ordinal indexing**
    - I used enum in my code for Directions, since Direction values are constants, and in my code I used EnumMap, it helped the code, since it's cleaner and it's optimized for **enum** keys while

HashMap is a general purpose **Map** implementation similar to Hashtable, examples :

```
public abstract class Entryway extends Lockable implements RoomObject,
Checkable {

  private EnumMap<Direction, Room> nextRoom;
  ...
}
```

```
public class Room implements Serializable {
  // ex : roomObjects.get(Direction.NORTH) gives us the roomObject on
the northern wall
  private EnumMap<Direction, RoomObject> roomObjects;
  ...
}
```

- Item 38 : **Check parameters for validity**
  - There is many forms to apply this rule, we could apply it to check if `int` value make since in a specific scenario, or we could use it to check if an object value is not null, this is what I done in my code to apply this rule, examples :

```
public void deposit(int amount) {
  if (amount < 0) {
    throw new IllegalArgumentException("This deposit value is
illegal(negative)!");
  }

  balance += amount;
}
```

```
public void removeItem(Item item) {
  if (item == null) {
    throw new NullPointerException("item value is null");
  }

  itemList.remove(item);
}
```

- Item 40 : **Design method signatures carefully**
  - **Choose method names carefully** : this point is reported in "**How my code satisfies the clean code principles according to “Clean Code” book**"

- "**Avoid long parameter lists**" : this point is reported in "**How my code satisfies the clean code principles according to "Clean Code" book**"

- Item 50 : **Avoid strings where other types are more appropriate**
  - "**Strings are poor substitutes for enum types**" : this is why I used enum for Direction, since it makes more since and easier to work with

```
public enum Direction {
  NORTH,
  EAST,
  SOUTH,
  WEST;
  ...
}
```

  - **Strings are poor substitutes for aggregate types** : I created a Key Class, because creating a string to represent a Key name and price is a really bad idea

- Item 51 : **Beware the performance of string concatenation**
  - Since String Objects are immutable, and building some data using them is not good for performance, so I used StringBuilder since it helped me build my Player Status in efficient way, example :

```
public StringBuilder getItemsStatus() {//inside ItemStatus class
  StringBuilder status = new StringBuilder("Item status : \n");
  for (int i = 0; i < itemList.size(); i++) {
    status.append("\t" + (i) + " : " + itemList.get(i) + "\n");
  }
  return status;
}
```

```
public StringBuilder getGoldStatus() {//inside GoldStatus class
  return new StringBuilder("Gold status : " + getBalance() + "\n");
}
```

```
public StringBuilder getStatus() {
  StringBuilder status = goldStatus.getGoldStatus();
  status.append(itemStatus.getItemsStatus());
  return status;
}
```

- Item 57 : **Use exceptions only for exceptional conditions**

- All my code exceptions satisfy this rule

- Item 60 : **Favor the use of standard exceptions**
  - Since the first version of my code, I used just standard exceptions, I didn't feel the need of creating new Exception types, but after reading this book I realized I'm not really using them the best way, I used `IllegalArgumentException` almost every where, but there is other standard exceptions that is more specific, like `NullPointerException` and `IndexOutOfBoundsException` , but eventually I fixed it

    example :

    Before the fix

```
public final void buyItem(
    Trader trader, Item item) {

  if (trader == null) {
    throw new IllegalArgumentException("trader value is null!");
  }

  if (item == null) {
    throw new IllegalArgumentException("item value is null!");
  }

  trader.sellItem(this, item);
}
```

    After the fix

```
public final void buyItem (Trader trader, Item item) {
  if (trader == null) {
    throw new NullPointerException ("trader value is null!");
  }

  if (item == null) {
    throw new NullPointerException ("item value is null!");
  }

  trader.sellItem(this, item);
}
```

    more

```
public Map( //map constructor
    Room startRoom,
    Room goalRoom,
    int timeToFinish,
```

```
      int initialGold,
      Direction playerFacingDirection) {

    if (startRoom == null) {
      throw new NullPointerException("start room value is null!");
    }

    if (goalRoom == null) {
      throw new NullPointerException("goal room value is null!");
    }

    if (timeToFinish < 0) {
      throw new IllegalArgumentException("time to finish value is
invalid");
    }

    if (initialGold < 0) {
      throw new IllegalArgumentException("initial gold value is
invalid");
    }

    if (playerFacingDirection == null) {
      throw new NullPointerException("player facing direction value is
null!");
    }

    this.startRoom = startRoom;
    this.goalRoom = goalRoom;
    this.timeToFinish = timeToFinish;
    this.initialGold = initialGold;
    this.playerFacingDirection = playerFacingDirection;
  }
```

- Item 66 : **Synchronize access to shared mutable data**
    - latter on in this report we will discuss "**Concurrency issues in my code and how I dealt with it**"
- Item 74 : **Implement Serializable judiciously**

  I used serialization for two reasons, the first one is after creating a Map, we serialize it into a file, so we can read the file later and load the map, the second use of serialization in my code is for saving the original state of the selected map in a String Object, and when we want to restart the map we read the serialized String Object

    - To save a Map into File, so we can load them later

      I would create a map and serialize it into a file that we can read later, I created a **MapLoader** class, that reads a file and retrieve the data from it, I used **Proxy Design Pattern** here so we don't load the same map more than once, we talked about this design pattern earlier in this report and we will talk about it later in details

    - To save the original state of the selected map

After the map and game mode are selected, we serialize the selected map into a String Object, so after a restart we get the original state back, this code is inside **Mode** class

```
private String serializedMap; // store original map
```

```
public void setMap(Map map) {
  if (map == null) {
    throw new IllegalArgumentException("map value is null!");
  }
  this.map = map;
  saveOriginalMap();
  restartMap();
  initialValuesToRestart();
}
```

```
private void saveOriginalMap() {
  ByteArrayOutputStream bos = new ByteArrayOutputStream();
  ObjectOutput out = null;
  try {
    out = new ObjectOutputStream(bos);
    out.writeObject(map);
  } catch (IOException e) {
    e.printStackTrace();
  }
  final byte[] byteArray = bos.toByteArray();
  serializedMap = Base64.getEncoder().encodeToString(byteArray);
}
```

```
public void restartMap() {
  final byte[] bytes = Base64.getDecoder().decode(serializedMap);

  ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
  ObjectInput in = null;
  try {
    in = new ObjectInputStream(bis);
    map = (Map) in.readObject();
  } catch (IOException e) {
    e.printStackTrace();
  } catch (ClassNotFoundException e) {
    e.printStackTrace();
  }
}
```

Discuss the design patterns I used to implement my solutions and why

This is my favorite part of the report, we will discuss 5 design patterns and how I used them

- **Command Design Pattern**
- **Singleton Design Pattern**
- **Proxy Design Pattern**
- **Template Method Design Pattern**
- **Builder Design Pattern**

Command Design Pattern

This design made the most thing I was worried about in this project easy to solve, I was worried because having multiple commands and implementing the needed action for them inside a specific class, will make the commands and that specific class coupled and as a result of that we will suffer when we want to add a new command

The idea here is to separate the commands from the controller (in our case the controller is the game mode)

```
public interface Command {
  void execute();
  String name();
}
```

This is the Command interface

I Created two other Command sub types (later on we can add more sub types for commands if needed), the first sub type command is `GameModeCommand`, witch is responsible of the command that effect the Game Mode it self, like `restart` and `exit`

```
public abstract class GameModeCommand implements Command {
  Mode mode
  public GameModeCommand(Mode mode) {

    if (mode == null) {
      throw new NullPointerException("game controller value is null!");
    }
    this.mode = mode;
  }

  public Mode getMode() {
    return mode;
  }
}
```

```
```

```
public class RestartMapCommand extends GameModeCommand {
  public RestartMapCommand(Mode mode) {
    super(mode);
  }

  @Override
  public void execute() {
    getMode().restart();
  }

  @Override
  public String name() {
    return "Restart Map";
  }
}
```

This is a concreate class(`RestartMapCommand`) that extends `GameModeCommand`, there is another one which is similar `QuitMapCommand`

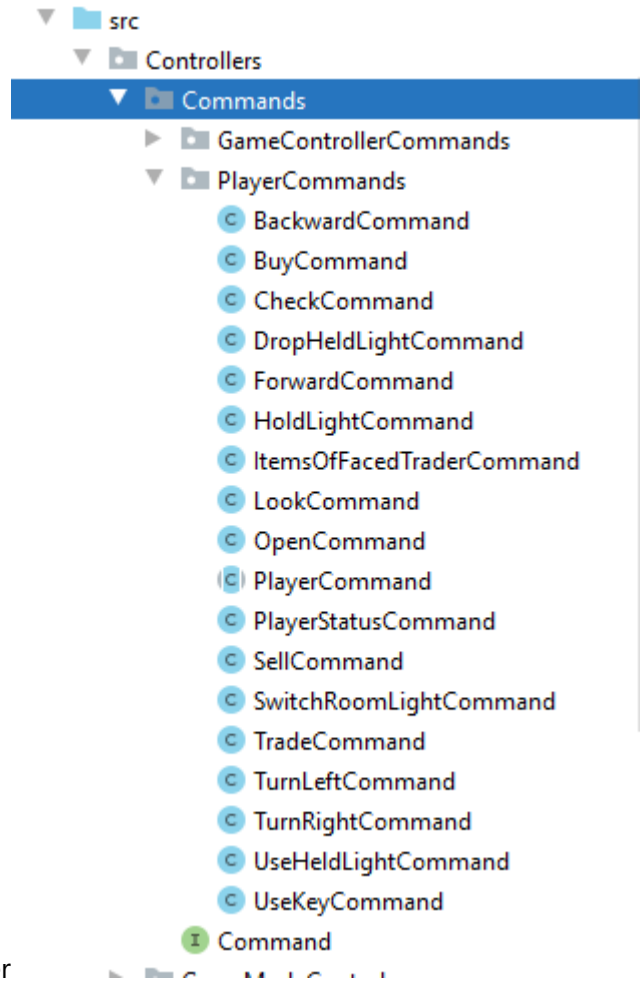
The other sub type that I created is `PlayerCommand` , which is responsible of the commands that effect the player and the gameplay, like `look` and `check`

```
public abstract class PlayerCommand implements Command {

  private Player player;

  public PlayerCommand(Player player) {
    if (player == null) {
      throw new NullPointerException("player value is null!");
    }

    this.player = player;
  }

  public Player getPlayer() {
    return player;
  }
}
```

```
public class SwitchRoomLightCommand extends PlayerCommand {
  public SwitchRoomLightCommand(Player player) {
    super(player);
  }

  @Override
  public void execute() {
    getPlayer().switchRoomLight();
  }
```

```
  @Override
  public String name() {
    return "Switch Room Light";
  }
}
```

This is a concreate class(`SwitchRoomLightCommand`), in this version there is 17 player commands, we can add more later



The game could consist of different game modes, the current version has just one mode, and that's for one player, we could create a multiplayer game mode.

```
public abstract class Mode {
  protected Map map;
  private ArrayList<Command> commands;
  private boolean isFinished;
  private String serializedMap; // store original map

  public Mode() {
    commands = new ArrayList();
  }

  public void setMap(Map map) {...}

  /**
```

```
    * save/serialize the map object into (serializedMap : String) so we can get the
original map back
    * after a restart
    */
   private void saveOriginalMap() {...}
   /** restart the map to it's original state */
   public void restartMap() {...}

   public boolean isFinished() {...}

   public void addCommand(Command command) {...}

   public void showCommands() {...}

   public void pickCommand() {...}

   public final void restart() {...}

   public final void start() {...}

   public abstract void run();

   public abstract void initialValuesToRestart();

   public abstract void quit();

   public abstract String getName();

   public abstract void addDefaultCommands();
}
```

as you can see, **Mode** class has `private ArrayList<Command> commands`, this list has commands that is ready to be executed, there is two ways to add commands to this game mode, first there is `addDefaultCommands`, this method adds the default commands that is part of the game mode, and the second way to add commands is to call `addCommand(Command command)` with the command that you want to add from the **Main** class, before you start the game

more information about the game structure :

- **GameDriver** is responsible to pick a map and to pick a game mode
- **GameDriver** has `private ArrayList<Mode> modes`, to add modes to the game driver we do that from the **Main** class
- **GameDriver** has `private ArrayList<MapLoader> maps`, to add a map, the serialized map file must be inside `resources\maps`
- The **Main** class adds the game modes to the **GameDriver** then it starts the **GameDriver**

```
public class Main {
```

```
    public static void main(String[] args) {

      GameDriver game = GameDriver.getInstance();

      OnePlayerMode gameControllerOnePlayer = new OnePlayerMode();

      game.addGameMode(gameControllerOnePlayer);

      game.start();
    }
  }
```

Singleton Design Pattern

Earlier when I started implementing the code for this project, I built the **Player** class as a singleton, but that was wrong, because I restricted the game to have just one player instead of having two or more or more players, eventually I made just one singleton class and that is **GameDriver**, because this project is about one game, so we just need one game driver to run the game

```
/**
 * GameDriver is the main entity in this game It controls the different maps and
game modes that it
 * has to offer
 */
public class GameDriver {
  private static GameDriver gameDriver = new GameDriver();
  private ArrayList<MapLoader> maps;
  private ArrayList<Mode> modes;

  private GameDriver() {
    maps = new ArrayList<MapLoader>();
    modes = new ArrayList<>();
  }

  public static GameDriver getInstance() {
    return gameDriver;
  }

  public void start() {...}

  public void addGameMode(Mode mode) {...}

  private void startNewGame() {...}

  private void findMaps() {...}

  private void showPossibleMaps() {...}

  private Map getMap(int index) {...}

  private Map pickMap() {...}
```

```
  private void showGameModes() {...}

  private Mode getMode(int index) {...}

  private Mode pickMode() {...}
}
```

Proxy Design Pattern

There are 3 reasons/ways to use **Proxy Design Pattern**

Remote : when you want to access something that is remote(another server or namespace, ...) **Virtual** : controls access to a resource that is expensive to create (image, ...) **Protection** : control access for a resource based on users' access rights

I used **Virtual Proxy Design Pattern**, because it makes my code avoid reading any file twice, I have **MapLoader** class that reads from files to deserialize Objects, and this design pattern helps with performance, since if you want to start a new game with a map that you played before, then no need to re read the file again and deserialize it's content

```
public class MapLoader {
  private Map map;
  private File file;

  public MapLoader(File file) {
    if (file == null) {
      throw new NullPointerException("file value is null!");
    }
    this.file = file;
  }

  public Map getMap() {
    load();
    return map;
  }

  public String getName() {
    return file.getName();
  }

  private void load() { // proxy pattern

    if (map != null) {
      return;
    }

    try {
      FileInputStream fileInputStream = new FileInputStream(file);
      ObjectInputStream objectInputStream = new
ObjectInputStream(fileInputStream);
```

```
        map = (Map) objectInputStream.readObject();

      } catch (IOException e) {
        e.printStackTrace();
      } catch (ClassNotFoundException e) {
        e.printStackTrace();
      }
    }
  }
```

as we mentioned before **GameDriver** class has `private ArrayList<MapLoader> maps` and `private void pickMaps()` this is the implementation of `pickMaps` method

```
  private Map pickMap() {
    Scanner scanner = new Scanner(System.in);
    int index = 0;
    Map map;

    while (true) {
      System.out.print("Enter map index : ");

      try {
        index = Integer.parseInt(scanner.next());
        map = getMap(index);
      } catch (NumberFormatException | IndexOutOfBoundsException e) {
        System.out.println("Not a valid index!");
        continue;
      }

      return map;
    }
  }
```

- If a map is picked, it gets loaded, otherwise it wont
- If a map is picked and it was picked before, the **MapLoader** will not read the map file again, because the map is already loaded

Builder Design Pattern

This design pattern helped me with the creation of **Map** Object

Map class has a lot of attributes that we need to specify, so instead of using the **Map** constructor with all of these arguments, we could implement a **MapBuilder** to make the map creation easier on us

```
public Map(Room startRoom, Room goalRoom, int timeToFinish, int
initialGold,Direction playerFacingDirection)
{...}//This is the Map constructor
```

this is the **MapBuilder** class

```
public class MapBuilder {
  private Room startRoom;
  private Room goalRoom;
  private int timeToFinish;
  private int initialGold;
  private Direction playerFacingDirection;

  public MapBuilder setStartRoom(Room startRoom) {...}

  public MapBuilder setGoalRoom(Room goalRoom) {...}

  public MapBuilder setTimeToFinish(int timeToFinish) {...}

  public MapBuilder setInitialGold(int initialGold) {...}

  public MapBuilder setPlayerFacingDirection(Direction playerFacingDirection)
{...}

  public Map buildMap() {
    return new Map(startRoom, goalRoom, timeToFinish, initialGold,
playerFacingDirection);
  }
}
```

Template Method Design Pattern

Template Method Design Pattern made me build some functionality easier, and I have used it more than once inside this project

these are some `final` methods that I consider **Template Method**

```
public final void restart() {//this is inside Mode class
  System.out.println("starting...");
  restartMap();
  isFinished = false;
  commands.clear();
  initialValuesToRestart();
  addDefaultCommands();
  commands.add(new RestartMapCommand(this));
  commands.add(new QuitMapCommand(this));
}

public final void start() {
  restart();
  run();
}
public abstract void initialValuesToRestart();
```

```
public abstract void addDefaultCommands();
```

```
public final void sellItem(Trader trader, Item item) {//inside Trader Class
  if (trader == null) {
    throw new NullPointerException("trader value is null!");
  }

  if (item == null) {
    throw new NullPointerException("item value is null!");
  }

  if (!canSellItemTo(trader, item)) {
    cancelSelling();
    trader.cancelBuying();
    return;
  }

  proceedSelling(item);
  trader.proceedBuying(item);
}

public abstract void cancelBuying();

public abstract void cancelSelling();

public abstract void proceedBuying(Item item);

public abstract void proceedSelling(Item item);
```

How my code satisfies the SOLID principles?

- **Single responsibility principle**
    - "*Do one thing and do it well*", We can see this in every class that I implemented

      To ensure this principle for functions, following "**SMAL!** rule from **chapter 3 functions ** in **clean code** Book" is a good idea

      every time I see a big function, I extract more functions out of it and make every function do just one thing, look to **Template Method Design Pattern** for examples

      Single responsibility principle is also for classes, there are 3 classes responsible with how to work with maps

        1. **Map**
        2. **MapBuilder**
        3. **MapLoader**

also **Trader** class (which is the parent class for **Player** and **Seller**) has two composite object of type `GoldStatus` and `ItemStatus`, these two classes are responsible to keep track with gold status and item status, there functionality should be encapsulated, a **bad** thing here is not to use these two classes and just implement there functionality inside **Player** class

- **Open/closed principle**
  - "*So, you should be able to extend your existing code using OOP features like inheritance via subclasses and interfaces. However, you should never modify classes, interfaces, and other code units that already exist*"

    That's why I used **Command Design Pattern**, to implement a new command no need to refactor anything, we can just simple implement our new command in a new class and just add it to the needed game mode

- **Liskov substitution && Interface segregation principles**
  - I put these two principles together because I see them very similar, no class should ever inherent a method that he cant use, that's why I created these interfaces and abstract classes, they help with that

```
public interface RoomObject {
  String look();
}
```

```
public interface Checkable {
  String check();
}
```

```
public abstract class Closeable implements Serializable {
  private boolean isOpen;
  public boolean isOpen() {
    return isOpen;
  }
  public void open() {
    isOpen = true;
  }
  public void close() {
    isOpen = false;
  }
}
```

```
public abstract class Lockable extends Closeable {
  private Key lockKey;
  private boolean isUnlocked;
```

```
  ...
}
```

```
public interface LootHider {
  Loot getLoot();
  void loot(Player player);
}
```

- **Dependency inversion principle**
  - "*High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*"

    To show how I used this principle let's look at these two classes, `Chest` and `Mirror`, as we know behind the mirror there could be a hidden key, and the chest could contain keys, flashlights and gold, If we use the `Check` command on then, the chest and the mirror would be looted, but the looting mechanism is different in those two classes, since mirror can only have a key and chest could have any type of items and gold

    This is why I created these Interfaces and abstract classes

```
public interface Loot {
  void loot(Player player);
}
```

```
public interface LootHider {
  Loot getLoot();
  void loot(Player player);
}
```

```
public class OneKeyLoot implements Loot, Serializable {
  private Key key;
```

```
public class FullLoot implements Loot, Serializable {
  private int goldLoot;
  private ItemStatus lootStatus;
  ...
}
```

    This is `Chest` and `Mirror` classes

```
public class Chest extends Lockable implements RoomObject, Checkable,
LootHider {
  private FullLoot loot;
  ...
  @Override
  public void loot(Player player) {
    if (isUnlocked()) {
      loot.loot(player);
    }
  }
  ...
}
```

```
public class Mirror implements LootHider, Checkable, RoomObject,
Serializable {
  private OneKeyLoot keyLoot;
  ...
  @Override
  public void loot(Player player) {
    keyLoot.loot(player);
  }
  ...
}
```

This is `OneKeyLoot` and `FullLoot` classes

```
public class OneKeyLoot implements Loot, Serializable {
  private Key key;
  public void loot(Player player) {
    if (player == null) {
      throw new NullPointerException("player value is null!");
    }

    if (key == null) {
      System.out.println("No Key is hidden here");
      return;
    }
    ...
}
```

```
public class FullLoot implements Loot, Serializable {
  private int goldLoot;
  private ItemStatus lootStatus;
  public void lootGold(Player player) {...}
  public void lootItems(Player player) {...}
```

```
    public void loot(Player player) {
     lootGold(player);
     lootItems(player);
    }
  }
```

- As we can see, every loot type has it's own implementation, let's go back to "*High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*", we can see this example really followed the rule

How my code satisfies Google (or Twitter) styling guide

A team following a style guide helps everyone write code in a consistent way, and consistent code is easier to read and faster to update.

After looking at Google styling guide, I realized it's hard to go back and update 3000+ lines of code to follow the styling guide, so I found about a tool/plugin in the intellij IDE that makes this reformatting much easier and I used it

Concurrency issues in my code and how I dealt with it

There is a **GameTimer** class that is responsible for keep tracking of how many seconds have passed and how much is left to finish the map

```
public class GameTimer extends Thread {
  private int seconds;
  private boolean timeOut;
  public GameTimer(int seconds) {...}
  @Override
  public void run() {...}

  public boolean isTimeOut() {...}

  public void setSeconds(int seconds) {...}

  public synchronized int getRemainingSeconds() {...}

  public void addSeconds(int seconds) {...}

  public void removeSeconds(int seconds) {...}
}
```

Item 78 from Joshua's “Effective Java” book says : **SYNCHRONIZE ACCESS TO SHARED MUTABLE DATA**

- The values of any object of this class could be modified from multiple threads, and to make sure no ***race condition*** happen here we need to synchronize the **critical section** of the methods

Item 79 from Joshua's “Effective Java” book says : **AVOID EXCESSIVE SYNCHRONIZATION**

- "*Excessive synchronization can cause reduced performance, deadlock, or even nondeterministic behavior*", let's take a look at `removeSeconds(int seconds)` function

```
public void removeSeconds(int seconds) {
  if (seconds < 0) {
    throw new IllegalArgumentException("seconds value is invalid!");
  }
  synchronized (this) {
    setSeconds(getRemainingSeconds() - seconds);
  }
}
```

as you can see the synchronized part is just the part where the data is modified not the whole class, if i synchronized the whole function that would be bad for performance

now let's look at `getRemainingSeconds()`

```
public synchronized int getRemainingSeconds() {
  return seconds;
}
```

I did synchronize the whole function since it's just one line, but in my first version of the code, I didn't synchronize this get function at all, but after reading this part from "Java concurrency in Practice" book chapter 3 "sharing objects" : "Synchronizing only the setter would not be sufficient: threads calling get would still be able to see stale values"

And this is how I solved my concurrency issues

Data structures I used and why

There are these rules in Joshua's “Effective Java” book that effected what data structures to use

- Item 25: **Prefer lists to arrays**
    - This is why you will not see any arrays in my code, I just used ArrayList since it's easier to deal with and has more functionality

```
private ArrayList<Item> itemList;//inside itemStatus class
```

```
private ArrayList<MapLoader> maps;//inside GameDriver class
private ArrayList<Mode> modes;
```

```
private ArrayList<Command> commands;//inside Mode class
```

- Item 33: **Use EnumMap instead of ordinal indexing**
    - I have one data structure type in my code that deals with enum types and it's **EnumMap**, since it's more efficient than any type of maps when it comes to make the key in the map enum type

```
//inside Room class
// ex : roomObjects.get(NORTH) gives us the roomObject on the northern
wall
private EnumMap<Direction, RoomObject> roomObjects;
```

```
private EnumMap<Direction, Room> nextRoom;//inside EntryWay class
```