SOEN 6441

Advanced Programming Practices – Winter 2021

Warzone Project - Build 3 – Refactoring – Team 10:

TEAM MEMBERS:

1. Dhananjay Narayan
2. Madhuvanthi Hemanthan
3. Prathika Suvarna
4. Neona Pinto
5. Surya Manian

Potential Refactoring Targets:

The following list of refactoring targets have been taken mainly from the new requirements established in build 2, and based on pain points and inconsistencies encountered during the development of build 1.

1. Observer pattern for console logs
2. Refactor Adapter pattern for loading and saving of Domination and Conquest map types
3. Refactor Strategy pattern to player behavioral strategies
4. Improved display of information in console
5. Refactor error handling via observer pattern
6. Separated and modularized observer to view directory
7. Refactor the game with single & tournament mode
8. Validation in command pattern
9. Corrected and refactored according to the coding convention
10. Implement additional test cases
11. Added neutral country list
12. Refactor by adding Save Game, Load game
13. Refactored Show Map function
14. Refactor the Player's issueOrder() method to use Strategy pattern
15. Add Javadoc to private data members

Actual Refactoring Targets:

The list of actual refactoring taken from the target list above were chosen mainly because of the new requirements established in build 2, and on the greatest pain points and inconsistencies encountered during the development of build 1.

1) Refactor the Player's issueOrder() method to use Strategy pattern:

As per the bild 3 requirements we have refactored the issueOrder() method of the player class and each of the player strategies are making use of this method to issue its respective orders.

```java
/**
 * A function to get the issue order from player and add to the order list
 */
public void issueOrder() {
    Order l_Order = OrderCreater.CreateOrder(IssueOrder.Commands.split( regex: " "), player: this);
    addOrder(l_Order);
}
```

Player.java

```java
List<String> l_Commands = new ArrayList<>();
if (flag) {
    l_Commands.add( index: 0, element: "bomb");
    l_Commands.add( index: 1, toCountry.getName());
    String[] l_CommandsArr = l_Commands.toArray(new String[l_Commands.size()]);
    Order l_Order = new BombOrder();
    l_Order.setOrderInfo(OrderCreater.GenerateBombOrderInfo(l_CommandsArr, d_Player));
    IssueOrder.Commands = l_Order.getOrderInfo().getCommand();
    d_Logger.log(String.format("%s issuing new command: %s", d_Player.getName(), IssueOrder.
    d_Player.issueOrder();
    return true;
} else if (fromCountry.getArmies() > 0) {
    l_Commands.add( index: 0, element: "advance");
    l_Commands.add( index: 1, fromCountry.getName());
    l_Commands.add( index: 2, toCountry.getName());
    l_Commands.add( index: 3, String.valueOf(fromCountry.getArmies()));
    String[] l_CommandsArr = l_Commands.toArray(new String[l_Commands.size()]);
    Order l_Order = new AdvanceOrder();
    l_Order.setOrderInfo(OrderCreater.GenerateAdvanceOrderInfo(l_CommandsArr, d_Player));
    IssueOrder.Commands = l_Order.getOrderInfo().getCommand();
    d_Logger.log(String.format("%s issuing new command: %s", d_Player.getName(), IssueOrder
    d_Player.issueOrder();
    return true;
}
```

z

AggressiveStrategy.java

```java
// move armies to the weakest country from the other neighbouring countries of the same player
        for (Country l_Country : l_WeakestCountry.getNeighbors()) {
            if (l_Country.getPlayer().getName().equals(d_Player.getName())) {
                l_Commands = new ArrayList<>();
                l_Commands.add( index: 0,  element: "advance");
                l_Commands.add( index: 1, l_Country.getName());
                l_Commands.add( index: 2, l_WeakestCountry.getName());
                l_Commands.add( index: 3, String.valueOf(l_Country.getArmies()));
                l_CommandsArr = l_Commands.toArray(new String[0]);
                l_Order = new AdvanceOrder();
                l_Order.setOrderInfo(OrderCreater.GenerateAdvanceOrderInfo(l_CommandsArr, d_Player));
                IssueOrder.Commands = l_Order.getOrderInfo().getCommand();
                d_Player.issueOrder();
                d_Logger.log(String.format("%s issuing new command: %s", d_Player.getName(), IssueOrder.
                l_WeakestCountry = l_Country;
            }
        }
        return "pass";
    }
```

BenevolentStrategy.java

2) Refactor Adapter pattern for loading and saving of Domination and Conquest map files:

This refactoring was chosen in an effort to support the new requirement that the game be able to load and save Conquest type map files in addition to Domination maps. The idea is that the existing Domination processing code be changed as little as possible, while tailoring to the different format of the Conquest map format.

Before / After Refactoring:

Before refactoring the application only supported the Domination map file format. The code made direct calls to the DominationMap class to load or save the map.

After the refactoring the code still makes direct calls to the Domination map class for domination style maps, but for Conquest maps the system now uses the adapter class Adapter. Adapter, invokes the appropriate method in Adaptee class, and translates the conquest style map into the format returned by DominationMap class.

```java
/**
 * Constructor to initialize adaptee object
 * @param adp Object of adaptee class
 */
public Adapter(Adaptee adp) {
    this.d_adp = adp;
}

/**
 * This method loads the map file
 * @param p_GameMap  the game map
 * @param p_FileName the map file name
 * @throws ValidationException files exception
 */

public void readMap(GameMap p_GameMap, String p_FileName) throws ValidationException {
    d_adp.readMap(p_GameMap, p_FileName);
}

/**
 *Saves the input map to a given file name. Overwrites any existing map with the same name.
 * The map will only save if it is valid.
 * @param map The map to save.
 * @param fileName The name of the file to save to, including the extension.
 * @return Whether the file was successfully saved.
 * @throws IOException files exception
 */
public boolean saveMap(GameMap map, String fileName)  throws IOException {
    return d_adp.saveMap(map, fileName);
}
```

Adapter.java

```
Enter your map operation:
1. Enter help to view the set of commands
2. Enter exit to end map creation and save phase
savemap qwerty
 Which format do you want to save the file? Type the number.
1. Domination map
2. Conquest map
2
The map has been validated and is saved.
The loaded file is of the format Conquest map
```

savemap in console

DominationMap.java | Adaptee.java | Adapter.java | qwerty.map

```
[Map]
 author=Anonymous
[Continents]\nAFRICA=9
USA=6
AUSTRALIA=9
ASIA=6
ANTARTICA=8
[Territories]
penquin ANTARTICA india Melbourne
Pak AFRICA india NY
NY USA Pak
india ASIA Pak penquin
Melbourne AUSTRALIA penquin
```

Conquest Map

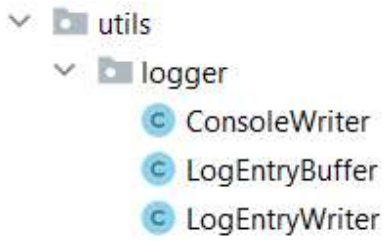
3) Observer pattern for console logs:

Before / After Refactoring:

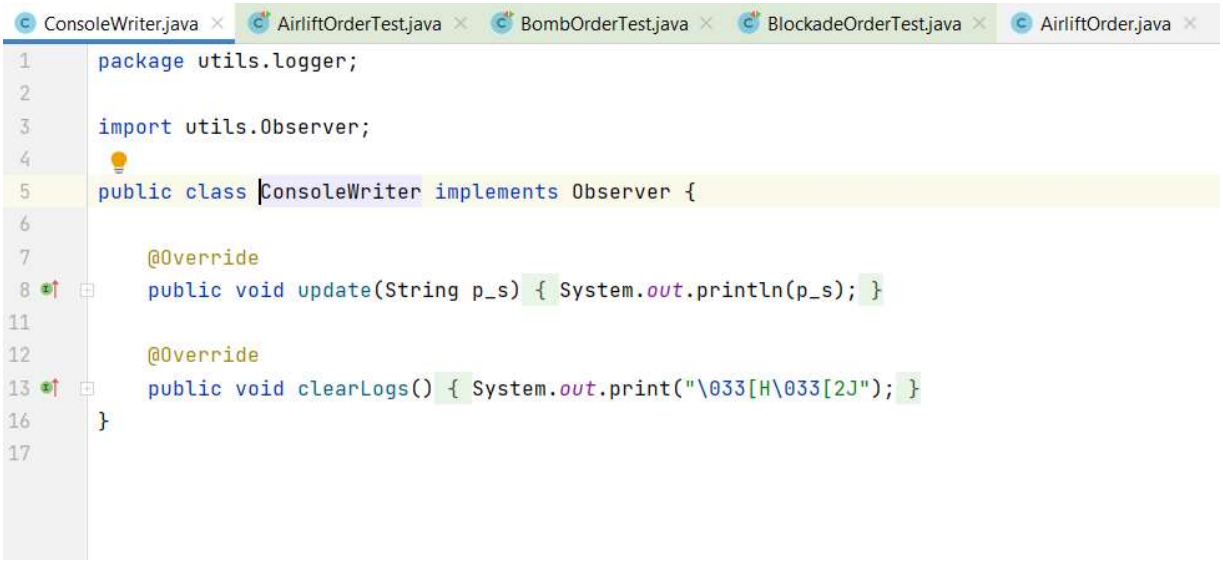
This refactoring was chosen not only because it is a requirement in build 2, but it makes the application easier to maintain, enhance and test.

The refactoring was done to write the logfile in the console and as a text file (demo.log).

The main refactoring was to make the observer write in the console using the ConsoleWriter implementing the Observer. Before refactoring, the observer wrote only to the log file.



ConsoleWriter class



ConsoleWriter implements Observer, to write in the console

```
    /**
     * This is the method to print
     */
    @Override
    public void printOrderCommand() {
        d_Logger.log( p_s: "Bomb Order issued by player: " + getOrderInfo().getPlayer().getName()
                + " on Country: " + getOrderInfo().getTargetCountry().getName());
        d_Logger.log( p_s: "----------------------------------------------------------------------" +
                "----------");
    }
}
```

d_Logger printing the log output in the console and the log file

4) Improved Display of information on Console:

Earlier the information displayed on the console during execution of the game was limited and this made it difficult for a smooth demonstration. In build 3 we have added additional information like number of armies per player and the players neighbouring countries in the display to make it easy for the user to write his next command.

```
/************************* You have entered the Reinforcement Phase ************************/
The Player:player1 is assigned with 15 armies.
The Player:player2 is assigned with 28 armies.



/************************* You have entered the IssueOrder Phase ************************/
List of game loop commands
To deploy the armies : deploy countryID numarmies
To advance/attack the armies : advance countrynamefrom countynameto numarmies
To airlift the armies : airlift sourcecountryID targetcountryID numarmies
To blockade the armies : blockade countryID
To negotiate with player : negotiate playerID
To bomb the country : bomb countryID
To skip: pass
================================================================================
Current Player Details Are:

+--------------+----------------------+------------------+
| Player Name   | Initial Assigned  | Left Armies          |
+---------------+------------------  +--------------------+
|player1        |15                 |15                    |
+--------------+----------------------+------------------+
The countries assigned to the player are:
+--------------+----------------------+------------------+---------+
|Country name  |Country Armies  | Neighbors                        |
+--------------+----------------------+------------------+---------+
|Melbourne      |0              |britain-penquin                   |
|NY             |0              |britain-Pak                       |
+--------------+----------------------+------------------+---------+
================================================================================
deploy NY 15
player1 has issued this order :- deploy NY 15
The order has been added to the list of orders.
```

Console

5) Refactor by adding changes in command pattern:

Added a private string variable to implement getCommand and setCommand methods in the OrderInfo.java file.

```java
/**
 * Getter for Command
 *
 * @return command
 */
public String getCommand() { return d_Command; }

/**
 * Setter for command
 *
 * @param p_Command command
 */
public void setCommand(String p_Command) { d_Command = p_Command; }
```

OrderInfo.java

Before/After Refactoring:

In each of the Orders we have created a log instance and implemented it inside every validateCommand method in this build.

```java
@Override
public boolean validateCommand() {
    Player l_Player = getOrderInfo().getPlayer();
    Country l_fromCountry = getOrderInfo().getDeparture();
    Country l_toCountry = getOrderInfo().getDestination();
    int p_armyNumberToAirLift = getOrderInfo().getNumberOfArmy();

    //check if the player is valid
    if (l_Player == null) {
        System.out.println("The Player is not valid.");
        return false;
    }
    //check if the player has an airlift card
    if (!l_Player.checkIfCardAvailable(CardType.AIRLIFT)) {
        System.out.println("Player doesn't have Airlift Card.");
        return false;
    }
```

Before: AirliftOrder.java

```java
@Override
public boolean validateCommand() {
    Player l_Player = getOrderInfo().getPlayer();
    Country l_fromCountry = getOrderInfo().getDeparture();
    Country l_toCountry = getOrderInfo().getDestination();
    int p_armyNumberToAirLift = getOrderInfo().getNumberOfArmy();

    //check if the player is valid
    if (l_Player == null) {
        d_Logger.log( p_s: "The Player is not valid.");
        return false;
    }
    //check if the player has an airlift card
    if (!l_Player.checkIfCardAvailable(CardType.AIRLIFT)) {
        d_Logger.log( p_s: "Player doesn't have Airlift Card.");
        return false;
    }
```

After: AirliftOrder.java