TEAM 18 : REFACTORING

SOEN 6441 Advanced Programming Practices

INSTRUCTOR: DR. AMIN RANJ BAR

TEAM MEMBERS:

1. Mariya Bosy Kondody

2.Reema Ann Reny

3. Meera Muraleedharan Nair

4.Madhav Anadkat

5. Jay Bhatt

6.Bhargav Fofandi

Potential Refactoring Targets:

The following list of refactoring targets have been taken mainly from the new requirements established in build 3, and based on pain points and inconsistencies encountered during the development of build 2.

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. Refactor the game with single & tournament mode
7. Validation in command pattern
8. Corrected and refactored according to the coding convention
9. Implement additional test cases
10. Added neutral country list
11. Refactored Show Map function
12. Refactor the Player's issueOrder() method to use Strategy pattern

Actual Refactoring Targets:

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

Following the Bild 3 requirements, we've redesigned the **issueorder()** method within the player class. Each player strategy now utilizes this method to issue its specific orders.

```
281     /**
282      * A function to get the issue order from player and add to the order list
283      */
284     public void deployOrder() {
285         Order l_Order = OrderCreator.CreateOrder(OrderIssue.Commands.split(regex:" "), this);
286         receiveOrder(l_Order);
287     }
```

Player.java

```
96          if (hasBombCard) {
97              Country enemyWithHighTroops = getEnemyWithHighTroops();
98              if (enemyWithHighTroops != null) {
99                  List<String> commands = Arrays.asList(...a:"bomb", enemyWithHighTroops.getCountryName());
100                 String[]l_commandsArr = commands.toArray(new String[0]);
101                 Order l_bombOrder = new BombingOrder();
102                 l_bombOrder.setOrderDetails(OrderCreator.GenerateBombOrderInfo(l_commandsArr, d_Player));
103                 OrderIssue.Commands = l_bombOrder.getOrderDetails().getCommand();
104                 d_LogEntryBuffer.logAction(
105                         String.format(format:"%s issuing new command: %s", d_Player.getPlayerName(), OrderIssue.Commands));
106                 d_Player.deployOrder();
107                 return true;
108             }
109         }
```

```
152             if (enemyCountry != null && fromCountry.getArmies() > 0) {
153                 List<String> commands = Arrays.asList(...a:"advance", fromCountry.getCountryName(),
154                         toCountry.getCountryName(),
155                         String.valueOf(fromCountry.getArmies()));
156                 String[] commandsArr = commands.toArray(new String[0]);
157                 Order l_advanceOrder = new AdvancingOrder();
158                 l_advanceOrder.setOrderDetails(OrderCreator.GenerateAdvanceOrderInfo(commandsArr, d_Player));
159                 OrderIssue.Commands = l_advanceOrder.getOrderDetails().getCommand();
160                 d_LogEntryBuffer.logAction(
161                         String.format(format:"%s issuing new command: %s", d_Player.getPlayerName(), OrderIssue.Commands));
162                 d_Player.deployOrder();
163                 flag = true;
164             }
165         }
```

AggressiveStrategy.java

```
105         for (Country l_Country : l_WeakestCountry.getNeighbors()) {
106             if (l_Country.getPlayer().getPlayerName().equals(d_Player.getPlayerName())) {
107                 l_Commands = new ArrayList<>();
108                 l_Commands.add(index:0, element:"advance");
109                 l_Commands.add(index:1, l_Country.getCountryName());
110                 l_Commands.add(index:2, l_WeakestCountry.getCountryName());
111                 l_Commands.add(index:3, String.valueOf(l_Country.getArmies()));
112                 l_CommandsArr = l_Commands.toArray(new String[0]);
113                 l_Order = new AdvancingOrder();
114                 l_Order.setOrderDetails(OrderCreator.GenerateAdvanceOrderInfo(l_CommandsArr, d_Player));
115                 OrderIssue.Commands = l_Order.getOrderDetails().getCommand();
116                 d_Player.deployOrder();
117                 d_LogEntryBuffer.logAction(
118                         String.format(format:"%s issuing new command: %s", d_Player.getPlayerName(), OrderIssue.Commands));
119                 l_WeakestCountry = l_Country;
120             }
121         }
```

BenevoIentStrategy.java

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

The refactoring was undertaken to accommodate a new requirement for the game to handle both Domination and Conquest map file formats during loading and saving operations. The primary goal was to minimize changes to the existing Domination processing code while adapting it to the distinct format of Conquest maps.

Prior to the refactoring, the application exclusively supported the Domination map file format, with direct calls to the DominationMap class for loading or saving maps. After the refactoring, the code still directly calls the DominationMap class for Domination-style maps. However, for Conquest maps, an adapter class, Adapter, is now employed. This Adapter invokes the appropriate method in the Adaptee class, translating the Conquest-style map into the format expected by the DominationMap class.

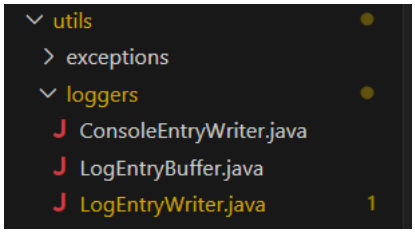
```
31      * This method loads the map file
32      * @param p_GameMap  the game map
33      * @param p_FileName the map file name
34      * @throws ValidationException files exception
```

```
143     public boolean saveMap(GameMap p_Map, String p_FileName) throws IOException {
144         StringBuilder mapDataBuilder = new StringBuilder(str:"[Map]\nauthor=Anonymous\n[Continents]\n");
145
146         for (Continent l_Continent : p_Map.getContinents().values()) {
147             mapDataBuilder.append(l_Continent.getContinentName()).append(str:"=").append(l_Continent.getBonusArmies()).append(str:"\n")
148         }
149
150         mapDataBuilder.append(str:"[Territories]\n");
151
152         for (Country l_Country : p_Map.getCountries().values()) {
153             mapDataBuilder.append(l_Country.getCountryName())
154                     .append(str:" ")
155                     .append(l_Country.getContinent())
156                     .append(str:" ")
157                     .append(createANeighborList(l_Country.getNeighbors()))
158                     .append(str:"\n");
159         }
```

adapter.java

3. Observer Pattern for Console Logs:

Before / After Refactoring: The decision to refactor was not solely driven by the requirement in build 2 but also aimed at enhancing the application's maintainability and testability.
The refactoring focused on logging mechanisms, specifically to write logs both to the console and a text file (demo.log).
The primary modification involved implementing the Observer pattern for console logs, utilizing the ConsoleWriter as the Observer. Prior to the refactoring, the observer was exclusively responsible for writing to the log file.



ConsoleWriter.class

```java
package utils.loggers;

import java.io.Serializable;

import utils.maputils.Observer;

Bhargav6100, 23 hours ago | 2 authors (Madhav Anadkat and others)
/**
 * A class to enable writing to console using the observer patter.
 *@author Jay Bhatt
 *@author Madhav Anadkat
 *@author Bhargav Fofandi
 */
public class ConsoleEntryWriter implements Observer, Serializable {

    /**
     * A function to update the string to observers
     *
     * @param p_s the message to be updated
     */
    @Override
    public void update(String p_s) {
        System.out.println(p_s);
    }

    /**
     * A function to clear the logs
     */
    @Override
    public void clearAllLogs() {
        System.out.print(s:"\033[H\033[2J");
    }
}
```

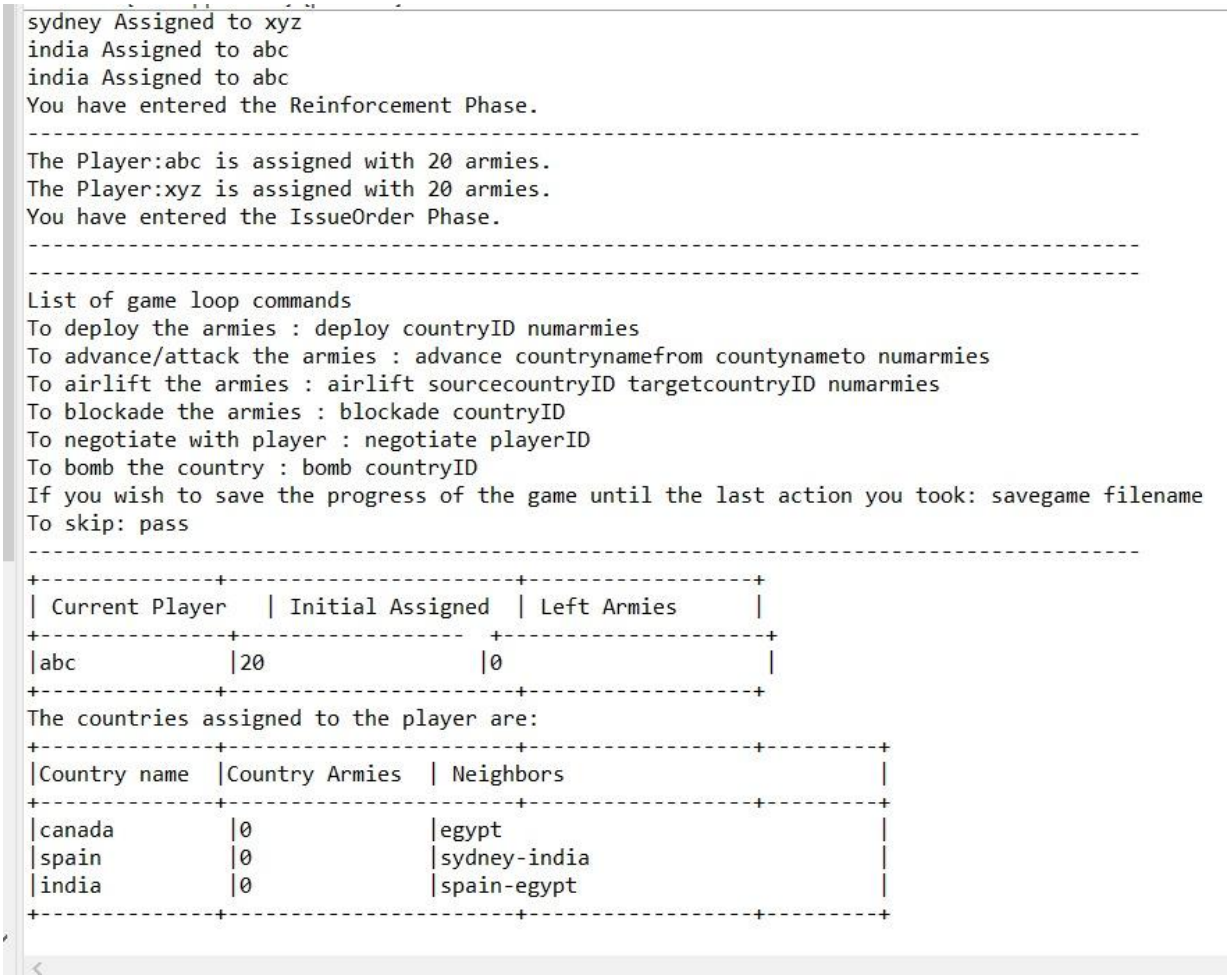
ConsoleWriter implements Observer, to write in the console

```java
110     /**
111      * Outputs the details of the bomb order.
112      */
113     @Override
114     public void printOrderCommand() {
115         String l_message = "Bombing order by " + getOrderDetails().getPlayer().getPlayerName() +
116                 " targeting " + getOrderDetails().getTargetCountry().getCountryName();
117         System.out.println(l_message);
118         System.out.println(x:"-----------------------------------------------------------------");
119         d_logEntryBuffer.logAction(l_message);
120     }
```

printing the log output in the console and the log file

4. Enhanced Console Information Display:

Previously, the information presented on the console during game execution was limited, hindering a seamless demonstration. In build 3, we have augmented the display with additional details such as the number of armies per player and the neighboring countries of each player. This improvement is intended to facilitate a more user-friendly experience, aiding users in formulating their next commands.



Console

5. Refactored by incorporating alterations in the Command Pattern:

In the OrderInfo.java file, modifications were made by introducing a private string variable to facilitate the implementation of the getCommand and setCommand methods.

```java
    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 the previous version, we established a log instance for each of the Orders and integrated it within every validateCommand method in the current build.

```java
public boolean validateCommand() {      JayRBhatt, 3 weeks ago • Order Card Implementations
    Player l_Player = getOrderDetails().getPlayer();
    Country l_from = getOrderDetails().getDeparture();
    Country l_to = getOrderDetails().getCountryWhereDeployed();
    int l_armies = getOrderDetails().getAmountOfArmy();
    boolean l_success = true;
    String l_log = "Failed due to some errors\n";
    if (l_Player == null || l_to == null || l_from == null) {
        System.err.println(x:"The information of the order is incorrect.");
        return false;
    }
    if (!l_Player.isCaptured(l_from)) {
        l_log += String.format(format:"\t-> Country %s does not belong to player %s\n", l_from.getCountryName(),
                l_Player.getPlayerName());
        l_success = false;
    }
    if (!l_from.isNeighbor(l_to)) {
        l_log += String.format(format:"\t-> Destination country %s is not a neighbor of %s\n", l_to.getCountryName(),
                l_from.getCountryName());
        l_success = false;
    }
    if (l_armies > l_from.getArmies()) {
        l_log += String.format(format:"\t-> Attacking troop count %d is greater than available troops %d in %s", l_armies,
                l_from.getArmies(), l_from.getCountryName());
        l_success = false;
    }
    if (!l_success) {
        System.err.println(l_log);
        d_leb.logAction(l_log);
    }
    return l_success;
}
```

AirIiftingOrder.java