TEAM 10: REFACTORING

COURSE: SOEN 6441 (WINTER 21)

INSTRUCTOR- JOEY PAQUET

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. Add Next order function in Player Class
2. Implement State Pattern in:
    a. Map editor
    b. Gameplay - Startup, Issue and Order
3. Implement command syntax validation
4. Implement Command pattern for processing of orders
5. Remove logic from model and add to controller - (IssueOrder)
6. Change the function from Controller to player in Reinforcement
7. Change Few Naming convention that may not be understandable
8. Implement Exception handling - Adding country without continent, missing information , Handle the exception for possible typo while adding neighbours
9. Implement additional test cases for existing logic
10. Change Continent Check in Map validation
11. Add Javadoc for private data members
12. Prevent user from moving forward with game if there is less than two player
13. Make file path as constant
14. Implement Observer pattern for console log
15. Change the format of saving map as per domination map

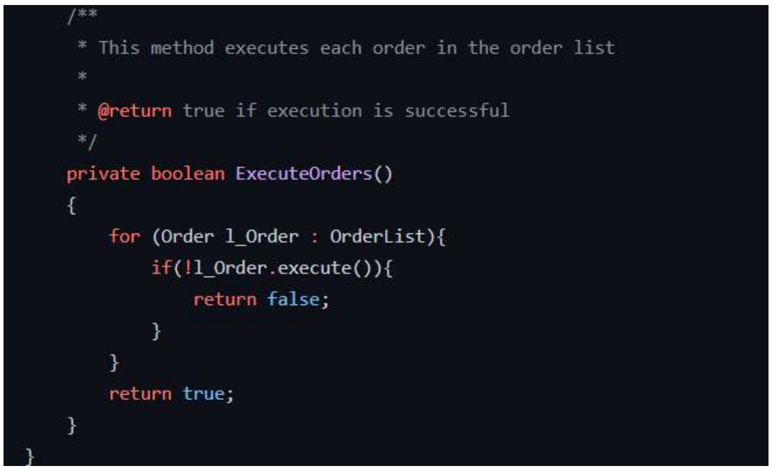
Actual Refactoring Targets:

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. Add next_order() function in Player Class, to get the next order of the player during the order execution phase:

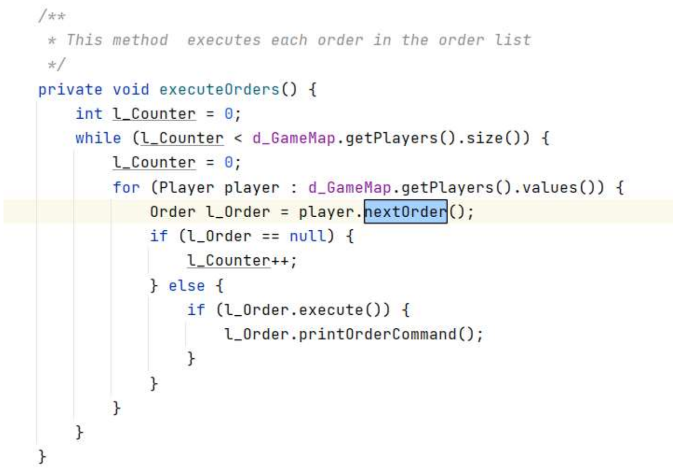
Before / After Refactoring:

We added the nextOrder() method as a refactor in this Build2, as we did not implement it in the first build. During the order execution phase, the GameEngine asks each Player for their next order using the nextOrder() method, then executes the order using the execute() method of the Order.



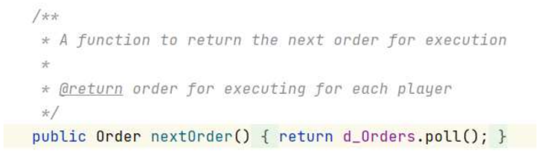
```
    /**
     * This method executes each order in the order list
     *
     * @return true if execution is successful
     */
    private boolean ExecuteOrders()
    {
        for (Order l_Order : OrderList){
            if(!l_Order.execute()){
                return false;
            }
        }
        return true;
    }
}
```

Before refactoring



```
/**
 * This method  executes each order in the order list
 */
private void executeOrders() {
    int l_Counter = 0;
    while (l_Counter < d_GameMap.getPlayers().size()) {
        l_Counter = 0;
        for (Player player : d_GameMap.getPlayers().values()) {
            Order l_Order = player.nextOrder();
            if (l_Order == null) {
                l_Counter++;
            } else {
                if (l_Order.execute()) {
                    l_Order.printOrderCommand();
                }
            }
        }
    }
}
```

After refactoring: ExecuteOrder.java

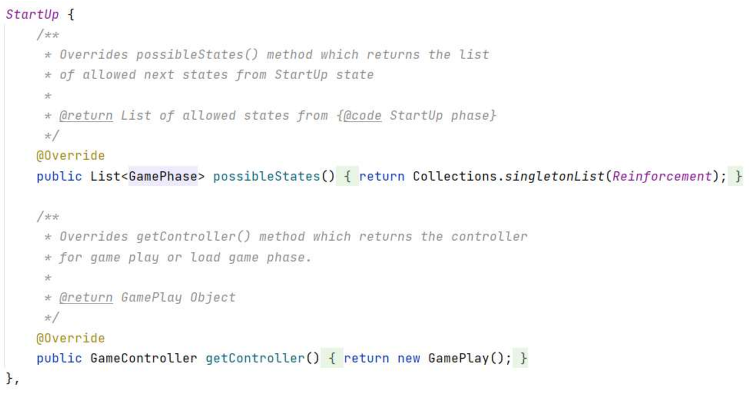


```
/**
 * A function to return the next order for execution
 *
 * @return order for executing for each player
 */
public Order nextOrder() { return d_Orders.poll(); }
```

Player.java

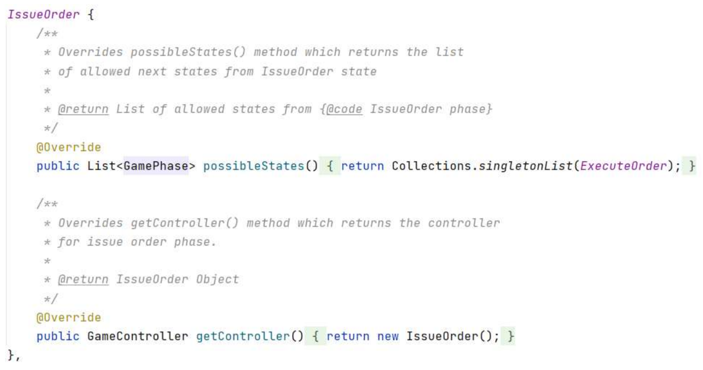
2. Implement State pattern for Phase Change:

This design pattern was chosen because before, the GameEngine held all the phase change logic centralized in several big methods. This made it difficult to maintain, enhance and unit test the phase changes. We implemented the pattern in the first build. In the second build, we made sure to refactor the code more specific towards the requirements. After the refactoring using the State pattern the phase change logic was easier to enhance and test.



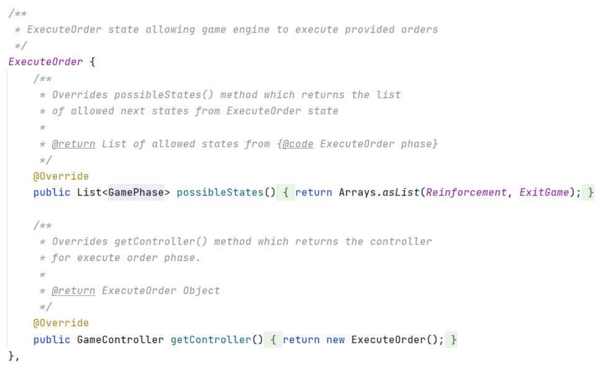
```
StartUp {
    /**
     * Overrides possibleStates() method which returns the list
     * of allowed next states from StartUp state
     *
     * @return List of allowed states from {@code StartUp phase}
     */
    @Override
    public List<GamePhase> possibleStates() { return Collections.singletonList(Reinforcement); }

    /**
     * Overrides getController() method which returns the controller
     * for game play or load game phase.
     *
     * @return GamePlay Object
     */
    @Override
    public GameController getController() { return new GamePlay(); }
},
```

Startup phase



```
IssueOrder {
    /**
     * Overrides possibleStates() method which returns the list
     * of allowed next states from IssueOrder state
     *
     * @return List of allowed states from {@code IssueOrder phase}
     */
    @Override
    public List<GamePhase> possibleStates() { return Collections.singletonList(ExecuteOrder); }

    /**
     * Overrides getController() method which returns the controller
     * for issue order phase.
     *
     * @return IssueOrder Object
     */
    @Override
    public GameController getController() { return new IssueOrder(); }
},
```

IssueOrder phase



```java
/**
 * ExecuteOrder state allowing game engine to execute provided orders
 */
ExecuteOrder {
    /**
     * Overrides possibleStates() method which returns the list
     * of allowed next states from ExecuteOrder state
     *
     * @return List of allowed states from {@code ExecuteOrder phase}
     */
    @Override
    public List<GamePhase> possibleStates() { return Arrays.asList(Reinforcement, ExitGame); }

    /**
     * Overrides getController() method which returns the controller
     * for execute order phase.
     *
     * @return ExecuteOrder Object
     */
    @Override
    public GameController getController() { return new ExecuteOrder(); }
},
```

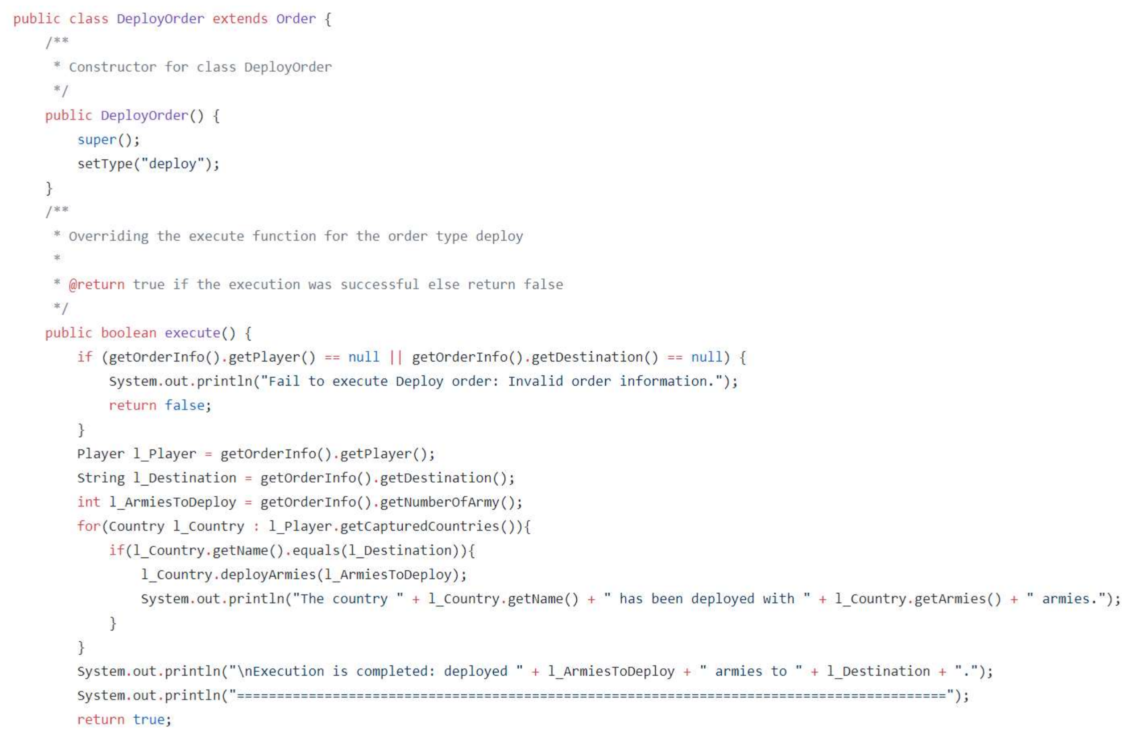
ExecuteOrder phase

Only upon successful execution of one phase, the next phase is entered.

3. Implement Command pattern for processing of orders:

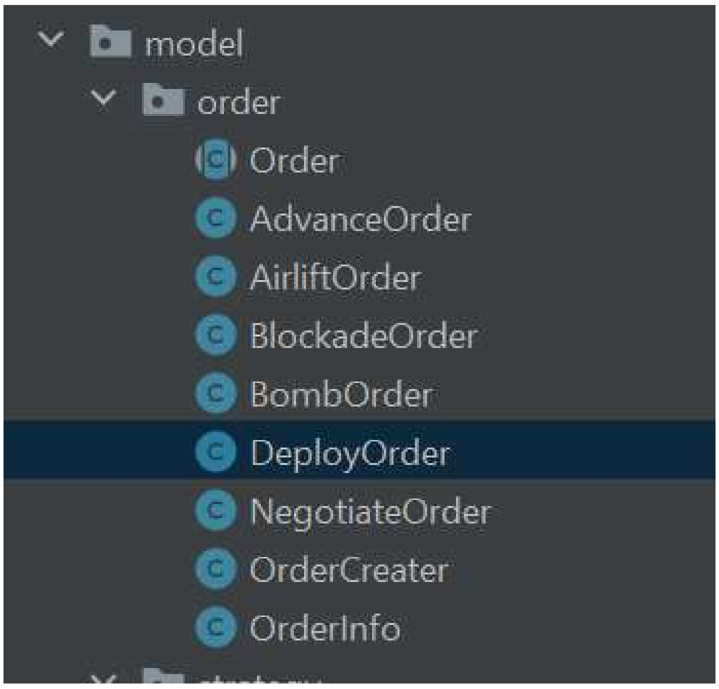
Before / After Refactoring:

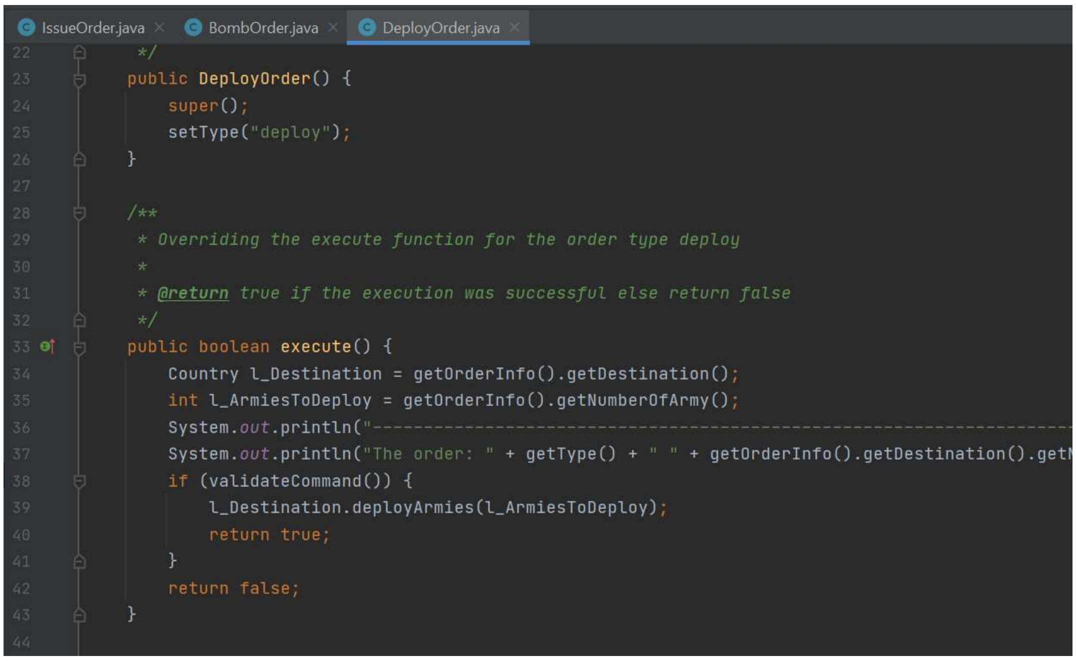
In Build1, the processing of deploy order did not follow command pattern, there was one function execute that had the logic. In build 2 all of the Commands - Deploy, Bomb, Advance, Blockade, Airlift, Bomb are following the command pattern. In addition to execute method, we also have Validation command method and print command method.



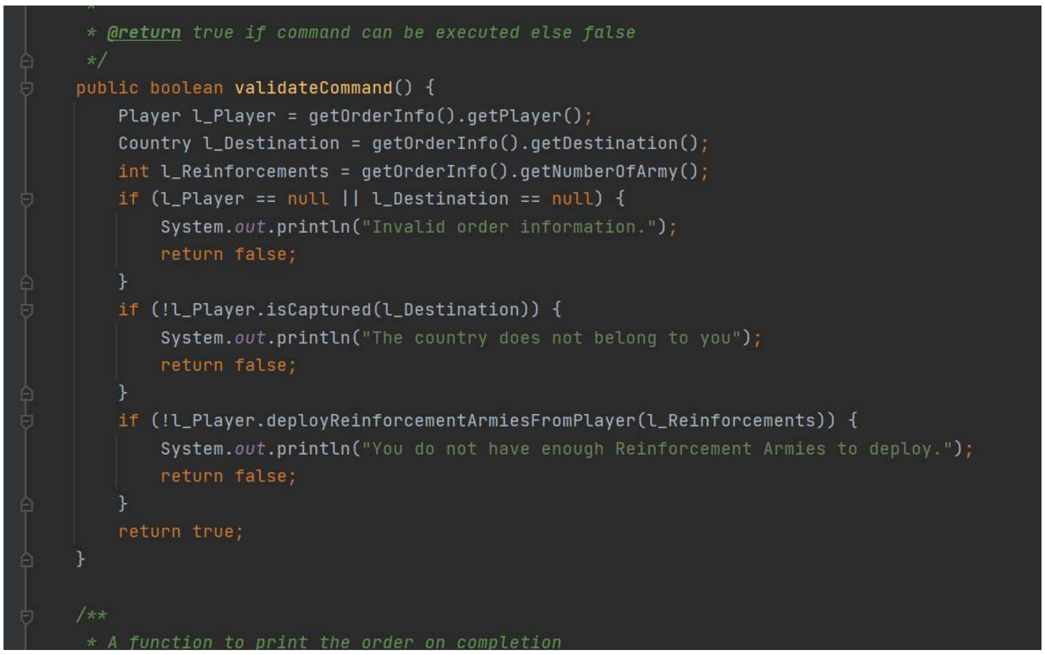
```java
public class DeployOrder extends Order {
    /**
     * Constructor for class DeployOrder
     */
    public DeployOrder() {
        super();
        setType("deploy");
    }
    /**
     * Overriding the execute function for the order type deploy
     *
     * @return true if the execution was successful else return false
     */
    public boolean execute() {
        if (getOrderInfo().getPlayer() == null || getOrderInfo().getDestination() == null) {
            System.out.println("Fail to execute Deploy order: Invalid order information.");
            return false;
        }
        Player l_Player = getOrderInfo().getPlayer();
        String l_Destination = getOrderInfo().getDestination();
        int l_ArmiesToDeploy = getOrderInfo().getNumberOfArmy();
        for(Country l_Country : l_Player.getCapturedCountries()){
            if(l_Country.getName().equals(l_Destination)){
                l_Country.deployArmies(l_ArmiesToDeploy);
                System.out.println("The country " + l_Country.getName() + " has been deployed with " + l_Country.getArmies() + " armies.");
            }
        }
        System.out.println("\nExecution is completed: deployed " + l_ArmiesToDeploy + " armies to " + l_Destination + ".");
        System.out.println("=====================================================================================");
        return true;
```

Before Refactoring: Order -> DeployOrder.java

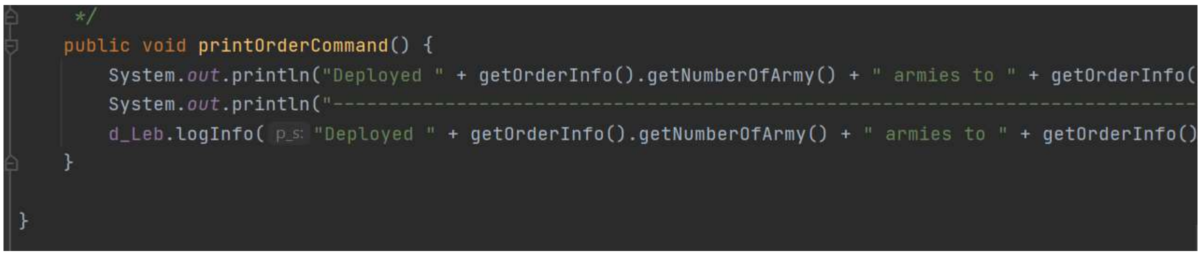




```java
     */
    public DeployOrder() {
        super();
        setType("deploy");
    }

    /**
     * Overriding the execute function for the order type deploy
     *
     * @return true if the execution was successful else return false
     */
    public boolean execute() {
        Country l_Destination = getOrderInfo().getDestination();
        int l_ArmiesToDeploy = getOrderInfo().getNumberOfArmy();
        System.out.println("-----------------------------------------------------------------------
        System.out.println("The order: " + getType() + " " + getOrderInfo().getDestination().getN
        if (validateCommand()) {
            l_Destination.deployArmies(l_ArmiesToDeploy);
            return true;
        }
        return false;
    }

```



```java
     * @return true if command can be executed else false
     */
    public boolean validateCommand() {
        Player l_Player = getOrderInfo().getPlayer();
        Country l_Destination = getOrderInfo().getDestination();
        int l_Reinforcements = getOrderInfo().getNumberOfArmy();
        if (l_Player == null || l_Destination == null) {
            System.out.println("Invalid order information.");
            return false;
        }
        if (!l_Player.isCaptured(l_Destination)) {
            System.out.println("The country does not belong to you");
            return false;
        }
        if (!l_Player.deployReinforcementArmiesFromPlayer(l_Reinforcements)) {
            System.out.println("You do not have enough Reinforcement Armies to deploy.");
            return false;
        }
        return true;
    }

    /**
     * A function to print the order on completion
```



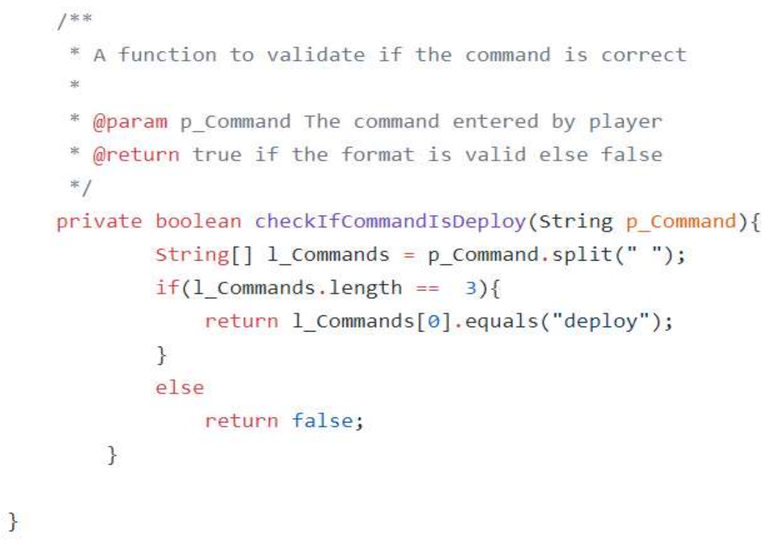
```java
     */
    public void printOrderCommand() {
        System.out.println("Deployed " + getOrderInfo().getNumberOfArmy() + " armies to " + getOrderInfo(
        System.out.println("---------------------------------------------------------------------------
        d_Leb.logInfo( p_s: "Deployed " + getOrderInfo().getNumberOfArmy() + " armies to " + getOrderInfo()
    }

}
```

After Refactoring: Order -> BombOrder.java

4. Implement Command syntax validation:

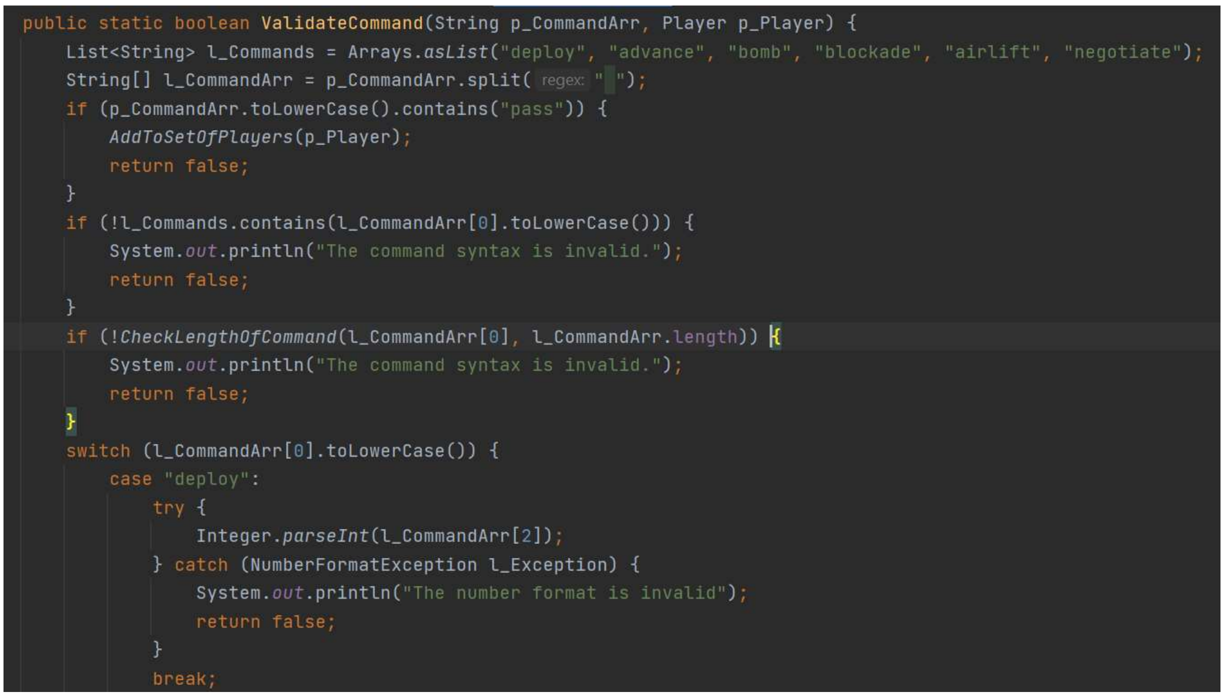
Before / After Refactoring:

In Build 1 the validation for command in IssueOrder.java file was done just for the deploy command as we had only one command. In Build 2 we have added the function ValidateCommand to check the validation for all the commands.

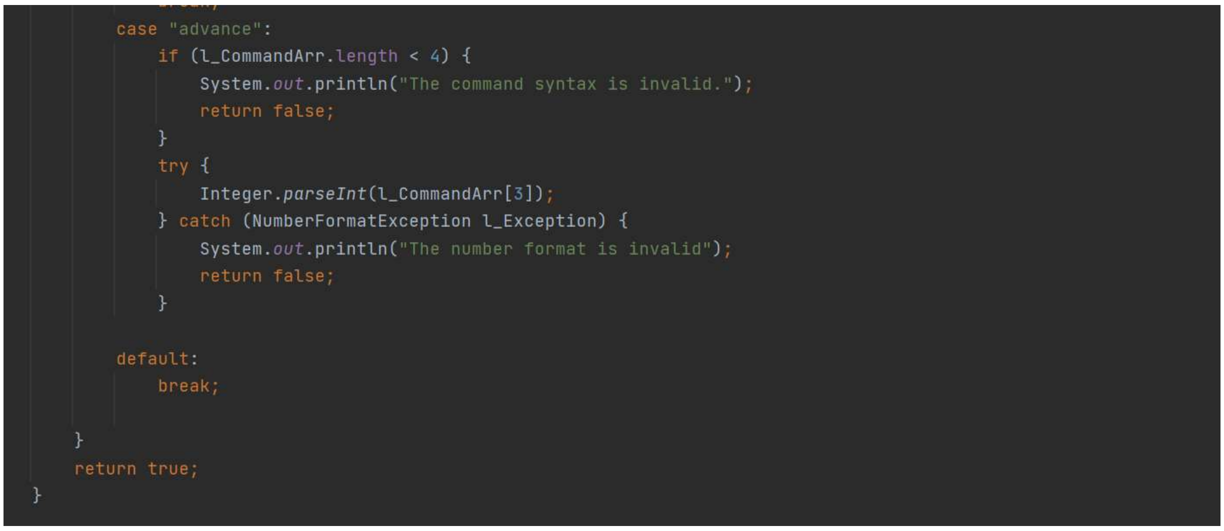


```
    /**
     * A function to validate if the command is correct
     *
     * @param p_Command The command entered by player
     * @return true if the format is valid else false
     */
    private boolean checkIfCommandIsDeploy(String p_Command){
            String[] l_Commands = p_Command.split(" ");
            if(l_Commands.length ==  3){
                return l_Commands[0].equals("deploy");
            }
            else
                return false;
        }

}
```

Before Refactoring: IssueOrder.java



```
public static boolean ValidateCommand(String p_CommandArr, Player p_Player) {
    List<String> l_Commands = Arrays.asList("deploy", "advance", "bomb", "blockade", "airlift", "negotiate");
    String[] l_CommandArr = p_CommandArr.split( regex: " ");
    if (p_CommandArr.toLowerCase().contains("pass")) {
        AddToSetOfPlayers(p_Player);
        return false;
    }
    if (!l_Commands.contains(l_CommandArr[0].toLowerCase())) {
        System.out.println("The command syntax is invalid.");
        return false;
    }
    if (!CheckLengthOfCommand(l_CommandArr[0], l_CommandArr.length)) {
        System.out.println("The command syntax is invalid.");
        return false;
    }
    switch (l_CommandArr[0].toLowerCase()) {
        case "deploy":
            try {
                Integer.parseInt(l_CommandArr[2]);
            } catch (NumberFormatException l_Exception) {
                System.out.println("The number format is invalid");
                return false;
            }
            break;
```



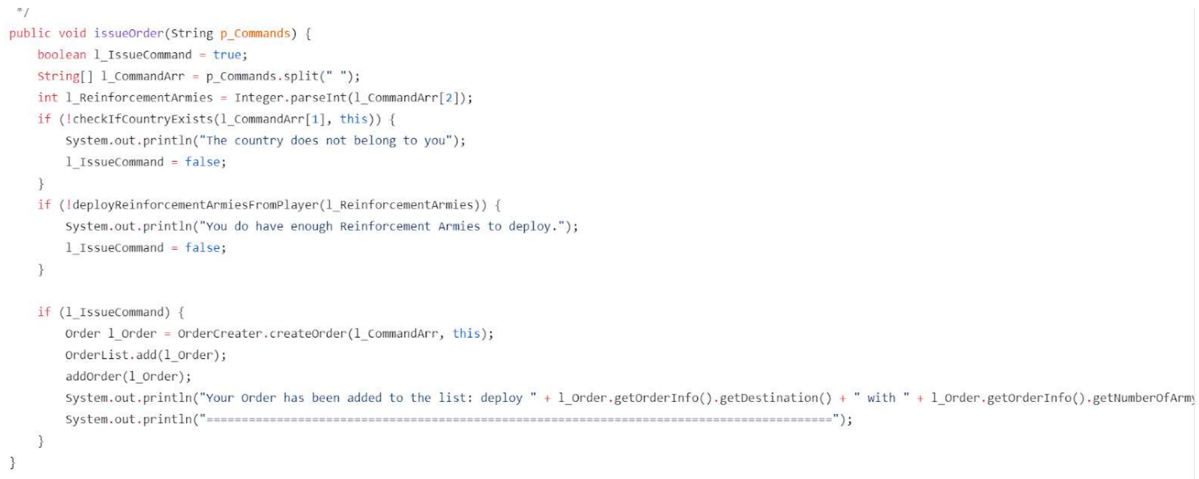
```java
        case "advance":
            if (l_CommandArr.length < 4) {
                System.out.println("The command syntax is invalid.");
                return false;
            }
            try {
                Integer.parseInt(l_CommandArr[3]);
            } catch (NumberFormatException l_Exception) {
                System.out.println("The number format is invalid");
                return false;
            }

        default:
            break;

    }
    return true;
}
```

After Refactoring: IssueOrder.java

5. Removed Logic from Model and added to Controller in IssueOrder:

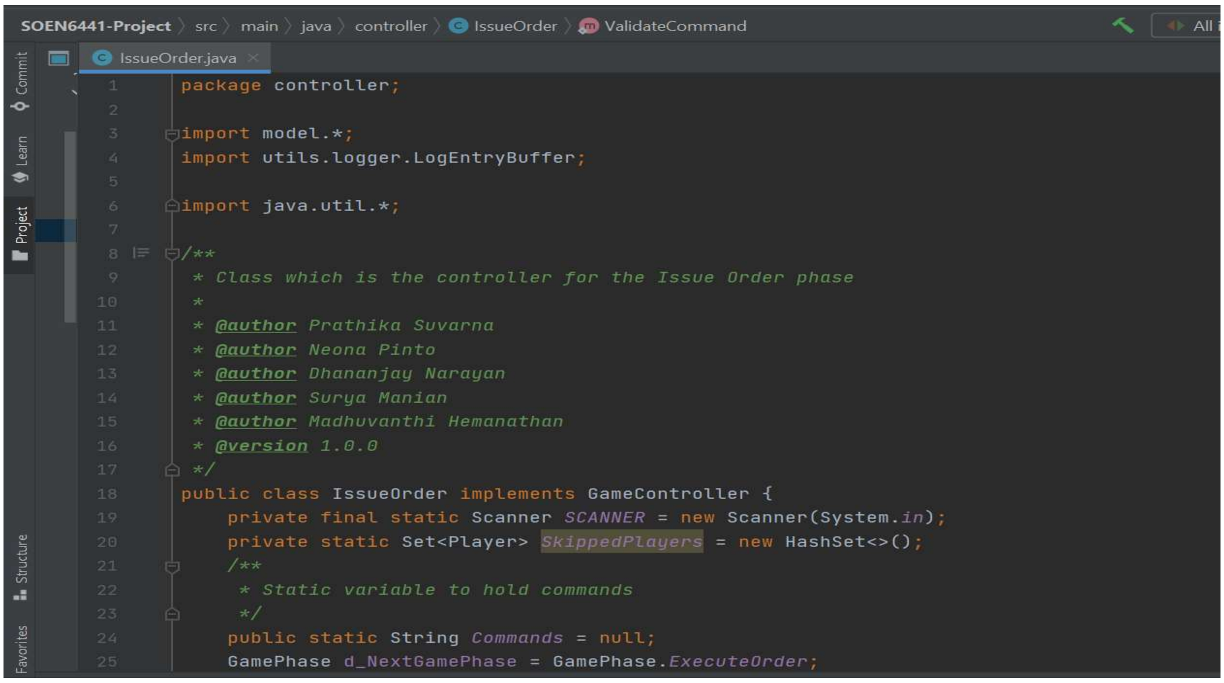
Before / After Refactoring:

In Build 1, the IssueOrder function was placed inside the Player Class under Model. We have done the refactoring by removing this function from the Player Class and implementing it separately as a IssueOrder.java class under Controller in Build 2.



```java
 */
public void issueOrder(String p_Commands) {
    boolean l_IssueCommand = true;
    String[] l_CommandArr = p_Commands.split(" ");
    int l_ReinforcementArmies = Integer.parseInt(l_CommandArr[2]);
    if (!checkIfCountryExists(l_CommandArr[1], this)) {
        System.out.println("The country does not belong to you");
        l_IssueCommand = false;
    }
    if (!deployReinforcementArmiesFromPlayer(l_ReinforcementArmies)) {
        System.out.println("You do have enough Reinforcement Armies to deploy.");
        l_IssueCommand = false;
    }

    if (l_IssueCommand) {
        Order l_Order = OrderCreater.createOrder(l_CommandArr, this);
        OrderList.add(l_Order);
        addOrder(l_Order);
        System.out.println("Your Order has been added to the list: deploy " + l_Order.getOrderInfo().getDestination() + " with " + l_Order.getOrderInfo().getNumberOfArmy
        System.out.println("=========================================================================================");
    }
}
```

Before Refactoring: Model->Player.java


SOEN6441-Project 〉 src 〉 main 〉 java 〉 controller 〉 IssueOrder 〉 ValidateCommand
IssueOrder.java

```java
package controller;

import model.*;
import utils.logger.LogEntryBuffer;

import java.util.*;

/**
 * Class which is the controller for the Issue Order phase
 *
 * @author Prathika Suvarna
 * @author Neona Pinto
 * @author Dhananjay Narayan
 * @author Surya Manian
 * @author Madhuvanthi Hemanathan
 * @version 1.0.0
 */
public class IssueOrder implements GameController {
    private final static Scanner SCANNER = new Scanner(System.in);
    private static Set<Player> SkippedPlayers = new HashSet<>();
    /**
     * Static variable to hold commands
     */
    public static String Commands = null;
    GamePhase d_NextGamePhase = GamePhase.ExecuteOrder;
```

After Refactoring: Controller -> IssueOrder.java

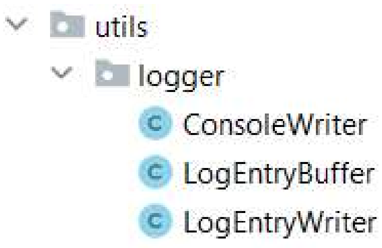
6. Implement Observer pattern for console log:

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.

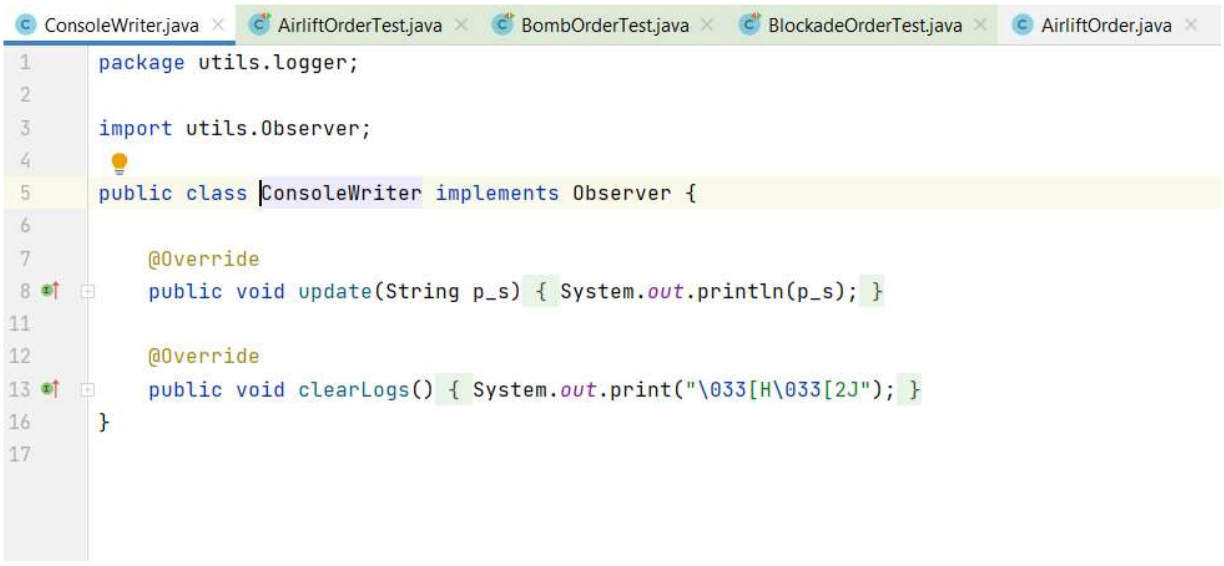
The refactoring was done to write the logfile in the console and as a text file (demo.log).

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.



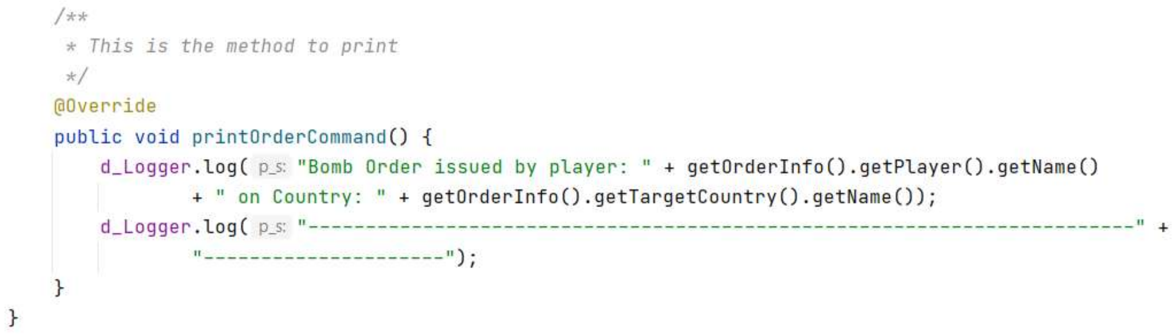
```
utils
  logger
    ConsoleWriter
    LogEntryBuffer
    LogEntryWriter
```

ConsoleWriter class



```
package utils.logger;

import utils.Observer;

public class ConsoleWriter implements Observer {

    @Override
    public void update(String p_s) { System.out.println(p_s); }

    @Override
    public void clearLogs() { System.out.print("\033[H\033[2J"); }
}
```

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