# TEAM TSPPS: REFACTORING
# Course: SOEN 6441(WINTER 25)
# INSTRUCTOR- Prof. / Dr. Mohamed Taleb

**TEAM MEMBERS:**

**1.Taha Mirza**
**2.Shariq Anwar**
**3.Pruthvirajsinh Dodiya**
**4.Poorav Panchal**
**5.Sakshi Mulik**

## 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. Make file path as constant
13. Implement Observer pattern for console log
14. Change the format of saving map as per domination map.
15. Change the Main Game loop as more orders were introduced.

## 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

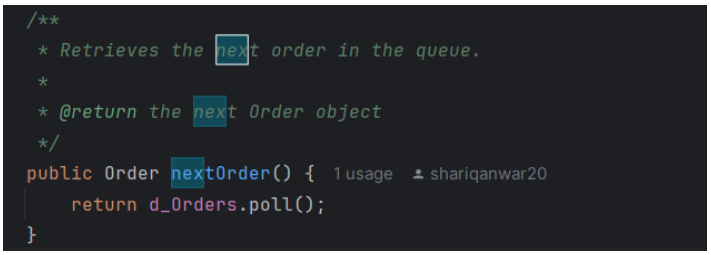
1. **Add Next order function in Player Class, to get the next order of the player during the order execution phase:**

   Before/After Refactoring:
   we added the nextOrder() method as a refactor in the Build2, as we did not implement it in the 1st 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.

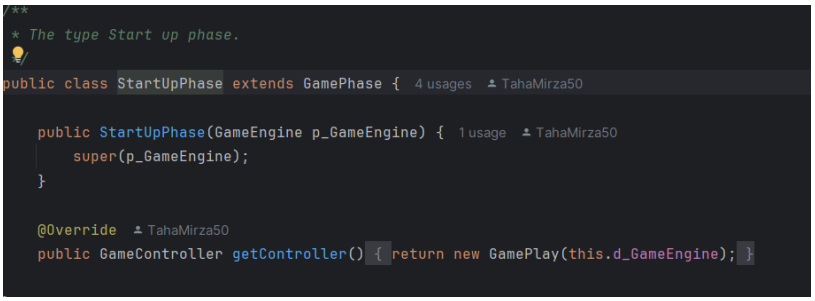

After refactoring: ExecuteOrder.java




After refactoring: 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 the phase changes. We implemented the pattern in the 1st build. In the 2nd 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 phase

```java
public class IssueOrderPhase extends GamePhase {  5 usages  TahaMirza50

    public IssueOrderPhase(GameEngine p_GameEngine) {  2 usages  TahaMirza50
        super(p_GameEngine);
    }

    @Override  TahaMirza50
    public GameController getController() { return new IssueOrder(this.d_GameEngine); }

}
```

IssuseOrder Phase

```java
public class ExecuteOrderPhase extends GamePhase {  3 usages  TahaMirza50

    public ExecuteOrderPhase(GameEngine p_GameEngine) { super(p_GameEngine); }

    @Override  TahaMirza50
    public GameController getController() { return new ExecuteOrder(this.d_GameEngine); }
}
```

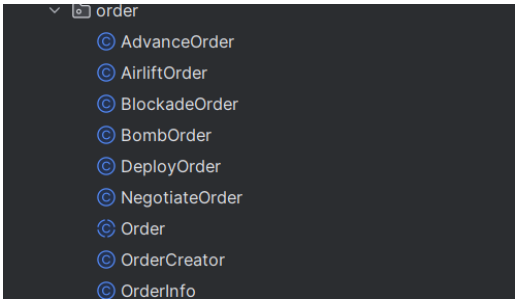
ExecuteOrder phase

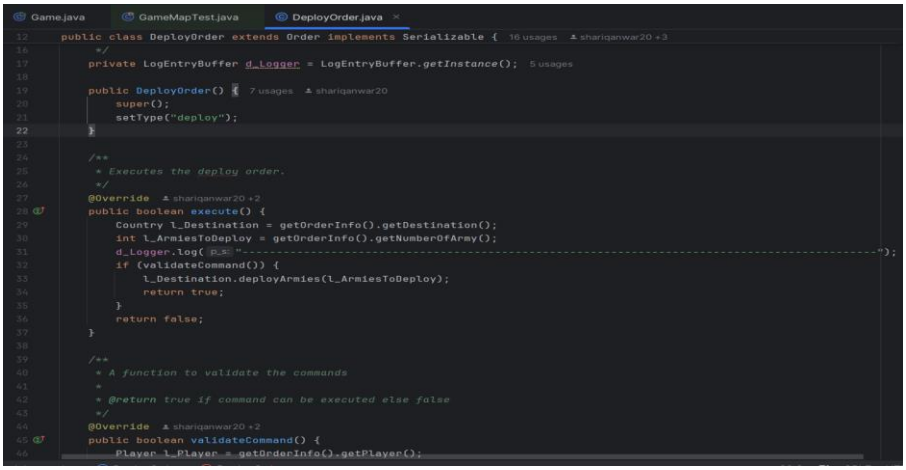
**3. Implement Command pattern for processing of order:**

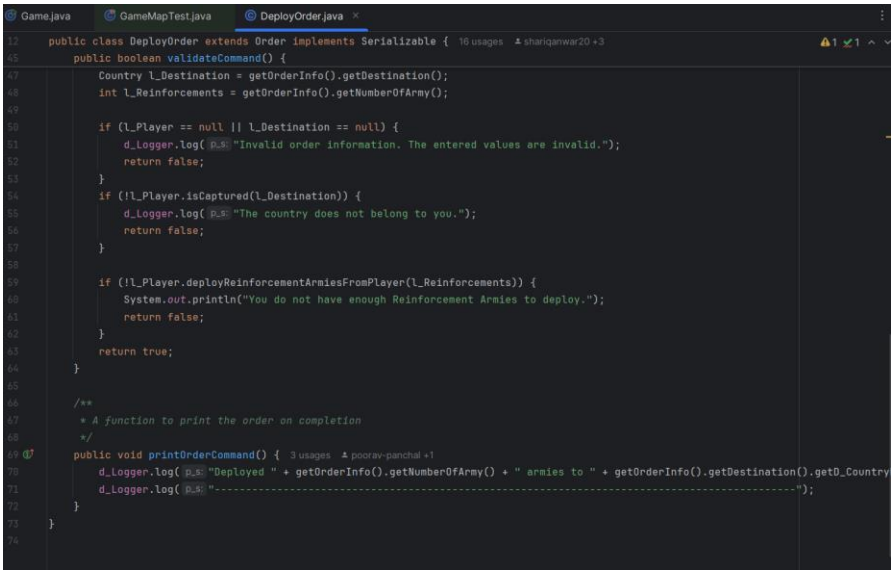
In build 1, the processing of deploy order did not follow command pattern, there was one function execute that had the logic. In build 2 all of the commands- Deploy, Bomb, Advance, Blockade, Airlift, Bomb are following the command pattern. In addition to execute method,

```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




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.

Before Refactoring : IssuseOrder.java

```
    /**
     * 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;
        }

    }
```

```
public boolean validateCommand(String p_CommandArr, Player p_Player) {  shariqanwar20 +1
    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")) {
        p_Player.setD_TurnCompleted(true);
        return false;
    }
    if (!l_Commands.contains(l_CommandArr[0].toLowerCase())) {
        d_Logger.log( p_s: "The command syntax is invalid." + p_CommandArr);
        return false;
    }
    if (!checkCommandLength(l_CommandArr[0], l_CommandArr.length)) {
        d_Logger.log( p_s: "The command syntax is invalid." + p_CommandArr);
        return false;
    }
    switch (l_CommandArr[0].toLowerCase()) {
        case "deploy":
            try {
                Integer.parseInt(l_CommandArr[2]);
            } catch (NumberFormatException l_Exception) {
                d_Logger.log( p_s: "The number format is invalid");
                return false;
            }
            if(Integer.parseInt(l_CommandArr[2]) < 0){
                d_Logger.log( p_s: "The number format is invalid");
                return false;
            }
```

```
        switch (l_CommandArr[0].toLowerCase()) {
            case "deploy":
                try {
                    Integer.parseInt(l_CommandArr[2]);
                } catch (NumberFormatException l_Exception) {
                    d_Logger.log( p_s: "The number format is invalid");
                    return false;
                }
                if(Integer.parseInt(l_CommandArr[2]) < 0){
                    d_Logger.log( p_s: "The number format is invalid");
                    return false;
                }
                break;
            case "advance":
                try {
                    Integer.parseInt(l_CommandArr[3]);
                } catch (NumberFormatException l_Exception) {
                    d_Logger.log( p_s: "The number format is invalid");
                    return false;
                }
                break;
            default:
                break;

        }
        return true;
    }
```

After Refactoring : IssuseOrder.java

**5. Removed logic from Model and added to Controller in IssuseOrder:**

Before/After Refactoring:

In Build 1, the IssuseOrder 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.

```
 */
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

```
import ...

/**
 * The type Issue order.
 */
public class IssueOrder extends GameController {  16 usages   shariqanwar20 +2 *

    private final HashMap<String, Player> d_Players;  6 usages

    /**
     * Static variable to hold commands
     */
    public static String Commands = null;  17 usages

    /**
     * Logger instance
     */
    private LogEntryBuffer d_Logger = LogEntryBuffer.getInstance();  26 usages

    /**
     * Instantiates a new Issue order.
     *
     * @param p_GameEngine the p game engine
     */
    public IssueOrder(GameEngine p_GameEngine) {  3 usages   TahaMirza50
        super(p_GameEngine);
        d_GameMap = GameMap.getInstance();
```

After Refactoring :controller->IssueOrder.java

**6. Implement Observer pattern for console log:**

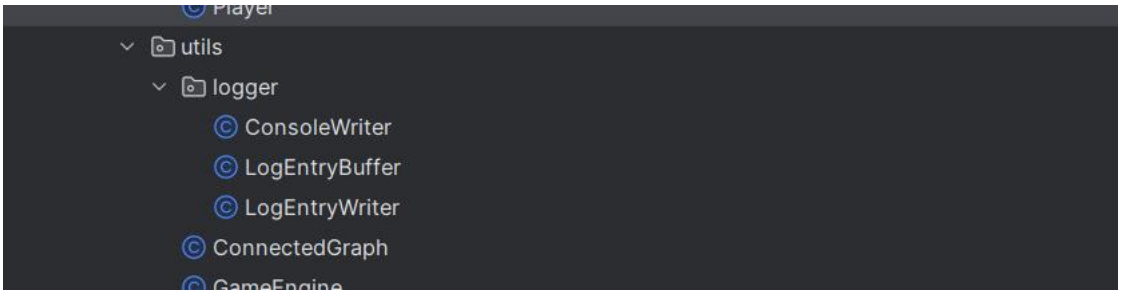
Before/after Refactoring:

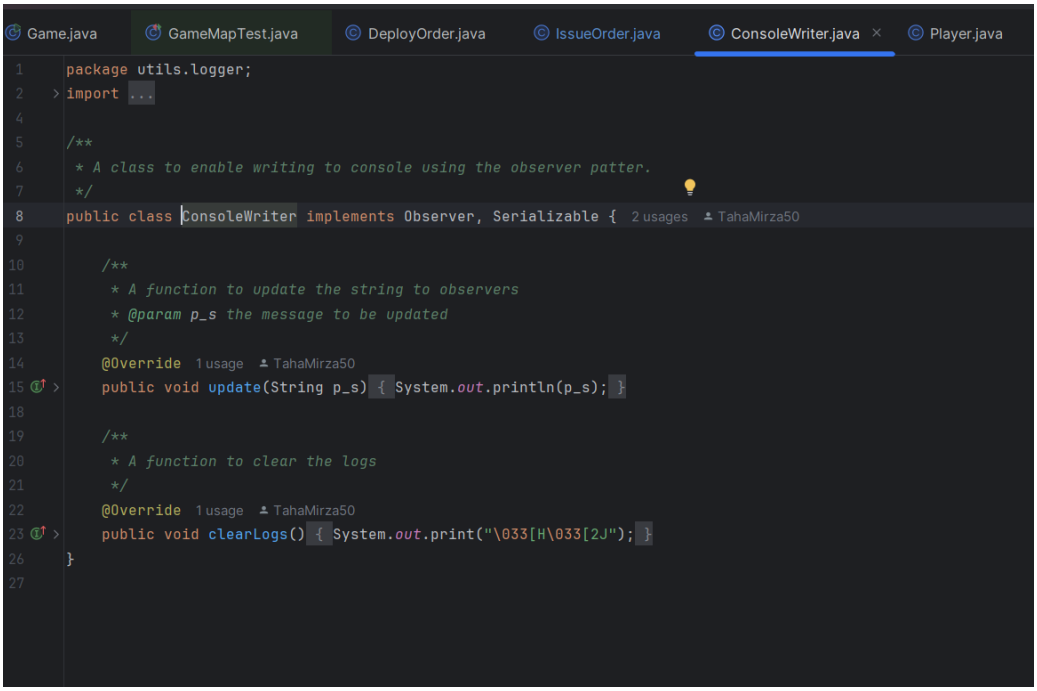
This refactoring was chosen not only because it is a requirement in build 2, but it makes the application easier to maintain, enhance and test.

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

The main refactoring was to make the observer write the console using the console writer

implementing the observer. Before refactoring, the observer wrote only to the log file.




ConsoleWriter implements Observer, to write in the console