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:

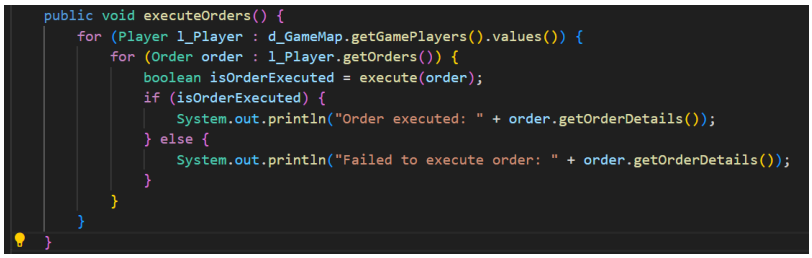
The potential refactoring targets have been found through careful analysis of the code base, keeping the new requirements for the build 2 as the primary concern. Also the change in requirements and the inconsistency faced in the code led to figure out the points mentioned below:

1. Add Next order function in Player Class.
2. Implement State Pattern in:
    i. Map editor Phase
    ii. Startup phase
    iii. Reinforcement Phase
    iv. Issue Order Phase
    v. Execute Order Phase
3. Implement command syntax validation
4. Implement Command pattern for processing of orders
5. Changed few filenames
6. Implement Exception handling
7. Implement additional test cases for existing logic
8. Add Javadoc for private data members
9. Implement Observer pattern for console log
10. Change the format of saving map as per domination map

Actual Refactoring Targets:

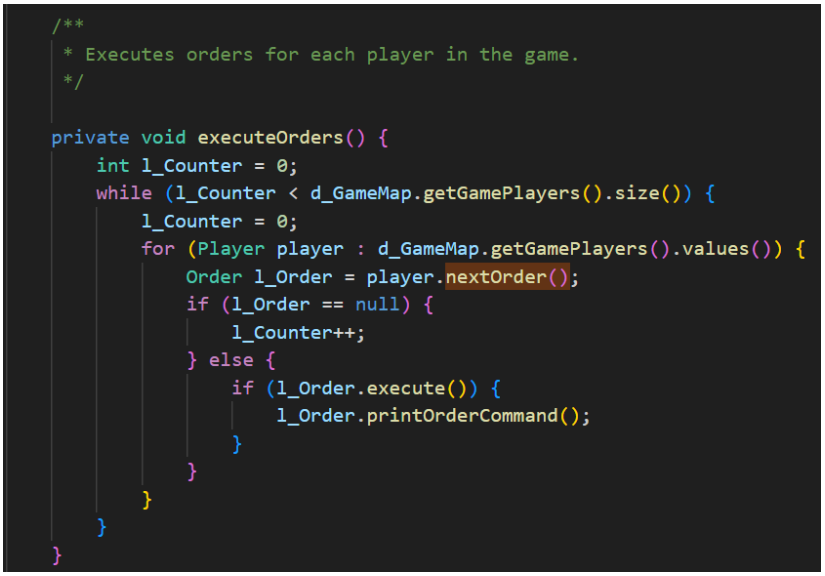
1. **Add next_order() function in Player Class, to** get the next order of the **player during the order execution phase:**

   In Build2, we included the nextOrder() method as part of our refactoring efforts since it wasn't initially integrated in the first build. In the order execution phase, the GameEngine prompts each Player for their subsequent order via the nextOrder() method, and then proceeds to carry out the order using the execute() method of the Order.



```
public void executeOrders() {
    for (Player l_Player : d_GameMap.getGamePlayers().values()) {
        for (Order order : l_Player.getOrders()) {
            boolean isOrderExecuted = execute(order);
            if (isOrderExecuted) {
                System.out.println("Order executed: " + order.getOrderDetails());
            } else {
                System.out.println("Failed to execute order: " + order.getOrderDetails());
            }
        }
    }
}
```

Before Refactoring



```
/**
 * Executes orders for each player in the game.
 */

private void executeOrders() {
    int l_Counter = 0;
    while (l_Counter < d_GameMap.getGamePlayers().size()) {
        l_Counter = 0;
        for (Player player : d_GameMap.getGamePlayers().values()) {
            Order l_Order = player.nextOrder();
            if (l_Order == null) {
                l_Counter++;
            } else {
                if (l_Order.execute()) {
                    l_Order.printOrderCommand();
                }
            }
        }
    }
}
```

After Refactoring: ExecuteOrder.java

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

After Refactoring: Player.java

2. Implement State pattern for Phase Change:

The state design pattern can be used to manage the different phases of the game within the GameEngineController interface. The state design pattern allows an object to alter its behavior when its internal state changes. In this case, the GamePhase represents the different states of the game, and the start method is responsible for transitioning between these states By employing the state design pattern, the GameEngineController can manage the flow of the game seamlessly, ensuring that the correct logic is executed based on the current state. This approach promotes better organization and maintainability of the game logic, making it easier to add or modify different phases of the game without impacting the overall structure.

```java
/**
 * @param p_GamePhaseID holding the ID of the current game phase
 * @throws InvalidCommandException
 */

public void controller(int p_GamePhaseID) throws InvalidCommandException {

    switch (p_GamePhaseID) {
        case 1:
            new MapEditor().mapEdit(p_GamePhaseID);
            break;
        case 2:
            new GamePlayBegins().runPhase(p_GamePhaseID);
            break;

        case 3:
            new Reinforcements().start(p_GamePhaseID);
            break;

        case 4:
            new OrderIssue().begin(p_GamePhaseID);
            break;

        case 5:
            new ExecuteOrder().startExecuteOrder(p_GamePhaseID);
            break;

        case 6:
            new ExitService().exitGame(p_GamePhaseID);
            break;

        default:
            System.exit(status:0);
    }
```

Before Refactoring

```
/**
 * MapEditor state handling map creation and validation operations
 */

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

    /**
     * Overrides getController() method which returns the controller
     * for map editor game phase.
     *
     * @return MapEditor Object
     */
    @Override
    public GameEngineController getController() {
        return new MapEditor();
    }
},
```

MapEditor Phase

```
/**
 * StartUp state handling load map, player creation and countries
 * allocation operations
 */
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 GameEngineController getController() {
        return new GamePlayBegins();
    }
},
```

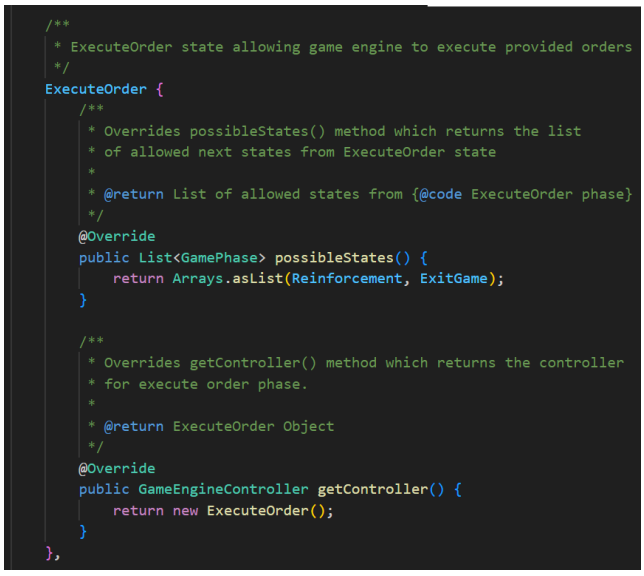
StartUp Phase

```
/**
 * Reinforcement state handling allocation of reinforcement
 * armies to each player after completing execute orders phase
 */
Reinforcement {
    /**
     * Overrides possibleStates() method which returns the list
     * of allowed next states from Reinforcement state
     *
     * @return List of allowed states from {@code Reinforcement phase}
     */
    @Override
    public List<GamePhase> possibleStates() {
        return Collections.singletonList(IssueOrder);
    }

    /**
     * Overrides getController() method which returns the controller
     * for reinforcement phase.
     *
     * @return Reinforcement Object
     */
    @Override
    public GameEngineController getController() {
        return new Reinforcements();
    }
},
```

Reinforcement Phase

```
/**
 * IssueOrder state allowing players to provide list of orders
 */
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 GameEngineController getController() {
        return new OrderIssue();
    }
},
```

Issue Order Phase

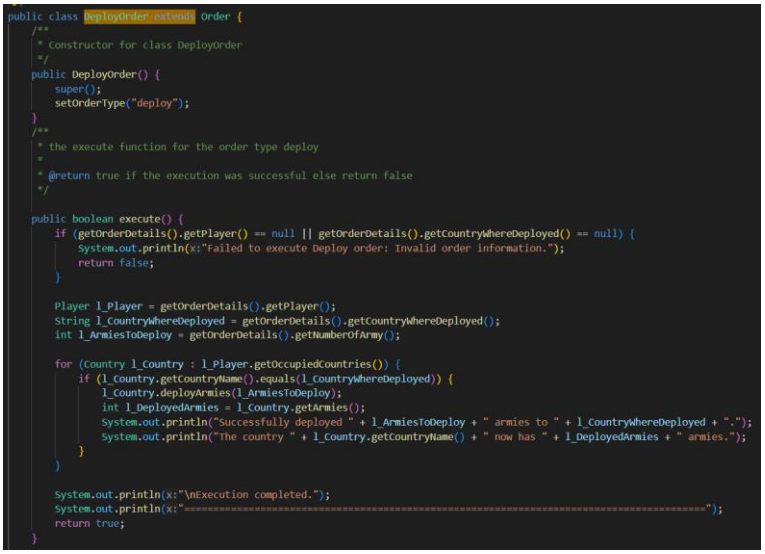


```
/**
 * ExecuteOrder state allowing game engine to execute provided orders
 */
ExecuteOrder {
    /**
     * Overrides possibleStates() method which returns the list
     * of allowed next states from ExecuteOrder state
     *
     * @return List of allowed states from {@code ExecuteOrder phase}
     */
    @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 GameEngineController getController() {
        return new ExecuteOrder();
    }
},
```

Execute Order Phase

3. Implement Command pattern for processing of orders:

In Build1, the deployment order processing lacked adherence to the command pattern, with all the logic contained within a single execute function. However, in Build2, each of the Commands, namely Deploy, Bomb, Advance, Blockade, Airlift, and Bomb, now conforms to the command pattern. In addition to the execute method, we've also introduced the validate and print command methods.



```java
public class DeployOrder extends Order {
    /**
     * Constructor for class DeployOrder
     */
    public DeployOrder() {
        super();
        setOrderType("deploy");
    }
    /**
     * the execute function for the order type deploy
     *
     * @return true if the execution was successful else return false
     */

    public boolean execute() {
        if (getOrderDetails().getPlayer() == null || getOrderDetails().getCountryWhereDeployed() == null) {
            System.out.println(x:"Failed to execute Deploy order: Invalid order information.");
            return false;
        }

        Player l_Player = getOrderDetails().getPlayer();
        String l_CountryWhereDeployed = getOrderDetails().getCountryWhereDeployed();
        int l_ArmiesToDeploy = getOrderDetails().getNumberOfArmy();

        for (Country l_Country : l_Player.getOccupiedCountries()) {
            if (l_Country.getCountryName().equals(l_CountryWhereDeployed)) {
                l_Country.deployArmies(l_ArmiesToDeploy);
                int l_DeployedArmies = l_Country.getArmies();
                System.out.println("Successfully deployed " + l_ArmiesToDeploy + " armies to " + l_CountryWhereDeployed + ".");
                System.out.println("The country " + l_Country.getCountryName() + " now has " + l_DeployedArmies + " armies.");
            }
        }

        System.out.println(x:"\nExecution completed.");
        System.out.println(x:"================================================================================");
        return true;
    }
```

Before Refactoring- DeployOrder.java

```java
public class DeployOrder extends Order {
    LogEntryBuffer d_LogEntryBuffer = new LogEntryBuffer();

    /**
     * Constructor for class DeployOrder
     */
    public DeployOrder() {
        super();
        setOrderType("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 = getOrderDetails().getCountryWhereDeployed();
        int l_ArmiesToDeploy = getOrderDetails().getAmountOfArmy();
        System.out.println(
                x:"--------------------------------------------------------------------------------");
        System.out.println(
                "The order: " + getOrderType() + " " + getOrderDetails().getCountryWhereDeployed().getCountryName()
                        + " " + getOrderDetails().getAmountOfArmy());
        if (validateCommand()) {
            l_Destination.deployArmies(l_ArmiesToDeploy);
            return true;
        }
        return false;
    }
```

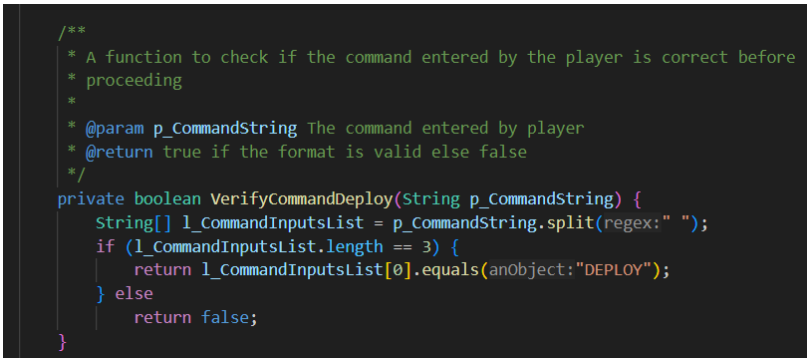
```java
    /**
     * A function to validate the commands
     *
     * @return true if command can be executed else false
     */
    public boolean validateCommand() {
        Player l_Player = getOrderDetails().getPlayer();
        Country l_Destination = getOrderDetails().getCountryWhereDeployed();
        int l_Reinforcements = getOrderDetails().getAmountOfArmy();
        if (l_Player == null || l_Destination == null) {
            System.out.println(x:"Invalid order information.");
            return false;
        }
        if (!l_Player.isCaptured(l_Destination)) {
            System.out.println(x:"The country does not belong to you");
            return false;
        }
        if (!l_Player.stationAdditionalArmiesFromPlayer(l_Reinforcements)) {
            System.out.println(x:"You do not have enough Reinforcement Armies to deploy.");
            return false;
        }
        return true;
    }
```

```
/**
 * A function to print the order on completion
 */
public void printOrderCommand() {
    System.out.println("Deployed " + getOrderDetails().getAmountOfArmy() + " armies to "
            + getOrderDetails().getCountryWhereDeployed().getCountryName() + ".");
    System.out.println(
            x:"-----------------------------------------------------------------------------------------");
    d_LogEntryBuffer.logAction("Deployed " + getOrderDetails().getAmountOfArmy() + " armies to "
            + getOrderDetails().getCountryWhereDeployed().getCountryName() + ".");

}
```

After Refactoring:DeployOrder.java

4. Implement Command syntax validation:

In Build 1, the VerifyCommandDeploy function focuses solely on verifying the format of the "DEPLOY" command.In Build 2, the CommandValidation method expands the validation scope to cover multiple commands, including "deploy," "advance," "bomb," "blockade," "airlift," and "negotiate." It splits the provided command string and performs checks for different aspects, such as command type, length, and specific argument validations for "deploy" and "advance" commands. Additionally, it includes a check for the "pass" command and handles it accordingly.



```
/**
 * A function to check if the command entered by the player is correct before
 * proceeding
 *
 * @param p_CommandString The command entered by player
 * @return true if the format is valid else false
 */
private boolean VerifyCommandDeploy(String p_CommandString) {
    String[] l_CommandInputsList = p_CommandString.split(regex:" ");
    if (l_CommandInputsList.length == 3) {
        return l_CommandInputsList[0].equals(anObject:"DEPLOY");
    } else
        return false;
}
```

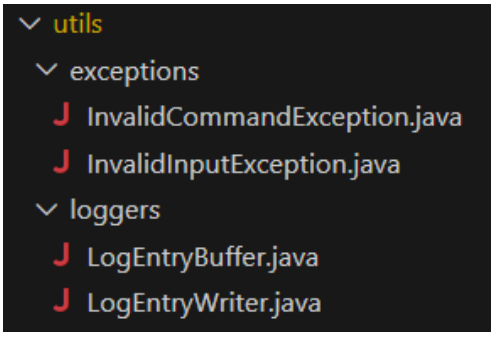
Before Refactoring

```
public static boolean CommandValidation(String p_CommandArr, Player p_Player) {
    List<String> l_Commands = Arrays.asList(...a:"deploy", "advance", "bomb", "blockade", "airlift", "negotiate");
    String[] l_CommandArr = p_CommandArr.trim().split(regex:"\\s+");

    String l_Command = l_CommandArr[0].toLowerCase();
    if ("pass".equals(l_Command)) {
        AddToSetOfPlayers(p_Player);
        return false;
    }

    if (!l_Commands.contains(l_Command) || !CheckLengthOfCommand(l_Command, l_CommandArr.length)) {
        System.out.println(x:"The command syntax is invalid.");
        return false;
    }

    if ("deploy".equals(l_Command) || "advance".equals(l_Command)) {
        return validateNumericArgument(l_CommandArr, l_Command.equals(anObject:"deploy") ? 2 : 3);
    }

    // No specific validation needed for other commands at this stage
    return true;
}
```

After Refactoring

5. Implement Observer pattern for console log:

The decision to refactor was motivated not only by the requirements in Build 2 but also by the aim to enhance the maintainability and testability of the application.The primary focus of the refactoring was to enable logging both in the console and as a text file (demo.log). This involved implementing the Observer pattern, where the observer was modified to utilize the ConsoleWriter in addition to writing to the log file. Prior to the refactoring, the observer only wrote to the log file.



ConsoleWriter class

```java
/**
 * Observes LogEntryBuffer and records the actions to a log file.
 */
public class LogEntryWriter implements Observer {

    /**
     * Receives updates from the observed subject and logs the information.
     *
     * @param p_message the information to be logged.
     */
    public void update(String p_message) {
        logMessage(p_message);
    }
```

LogEntryWriter implements Observer, to write in the console

```java
    /**
     * Outputs the details of the bomb order.
     */
    @Override
    public void printOrderCommand() {        JayRBhatt, 3 days ago • Order Card Implementations
        String l_message = "Bombing order by " + getOrderDetails().getPlayer().getPlayerName() +
                " targeting " + getOrderDetails().getTargetCountry().getCountryName();
        System.out.println(l_message);
        System.out.println(x:"-----------------------------------------------------------------");
        d_logEntryBuffer.logAction(l_message);
    }
```

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