Refactoring Document

Refactoring Targets

No.	Refactoring Targets
1.	Executed State Pattern
2.	Implemented Observer Pattern
3.	Implemented Command Pattern
4.	Remove deadcode
5.	Move logic of setting the continent list from startup phase to assign reinforcement
6.	Introducing Country owner field and creating respective getter setter methods in Country Class
7.	Improving UI in showmap functionality and showCommands()
8.	IssueOrder() method signature change
9.	Order Class updated to Interface
10.	Redundant code in GameEngine removed
11.	Removed implementation of deploy from Order Class (now interface) to Deploy Class
12.	Made a separate method isValid() in Player Controller Class for command validation from issueOrder() loop
13.	Implemented a separate algorithm to ask player to re-enter the order command if its syntactically invalid
14.	Added fields in Player Class to implement new functionalities
15.	Implemented new logic to stop the game when a player wins

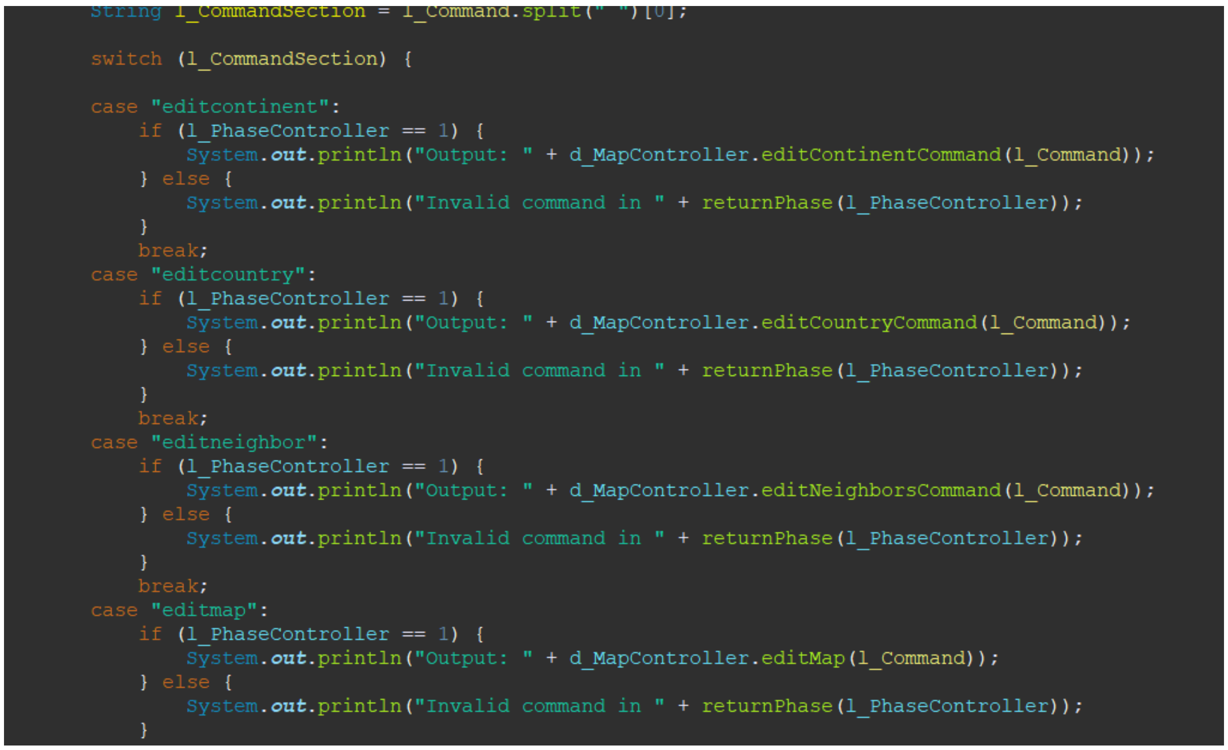
1. State Pattern

The State pattern was adopted for Phase Change in the game engine. Initially, the GameEngine contained all the logic for phase changes, which was consolidated into several large methods. This setup posed challenges for maintenance, enhancement, and testing of phase changes. The first build saw the implementation of the pattern, followed by a more specific refactoring towards the requirements in the second build. The application of the State pattern simplified the enhancement and testing of the phase change logic. This has enhanced code readability significantly.

Test Class

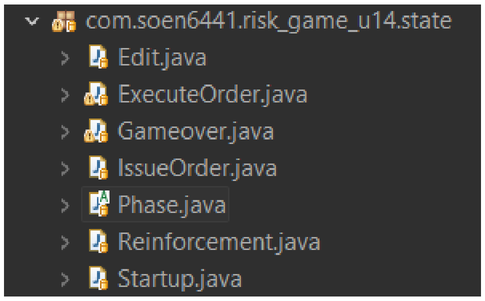
StateTestTestSuite

Before:

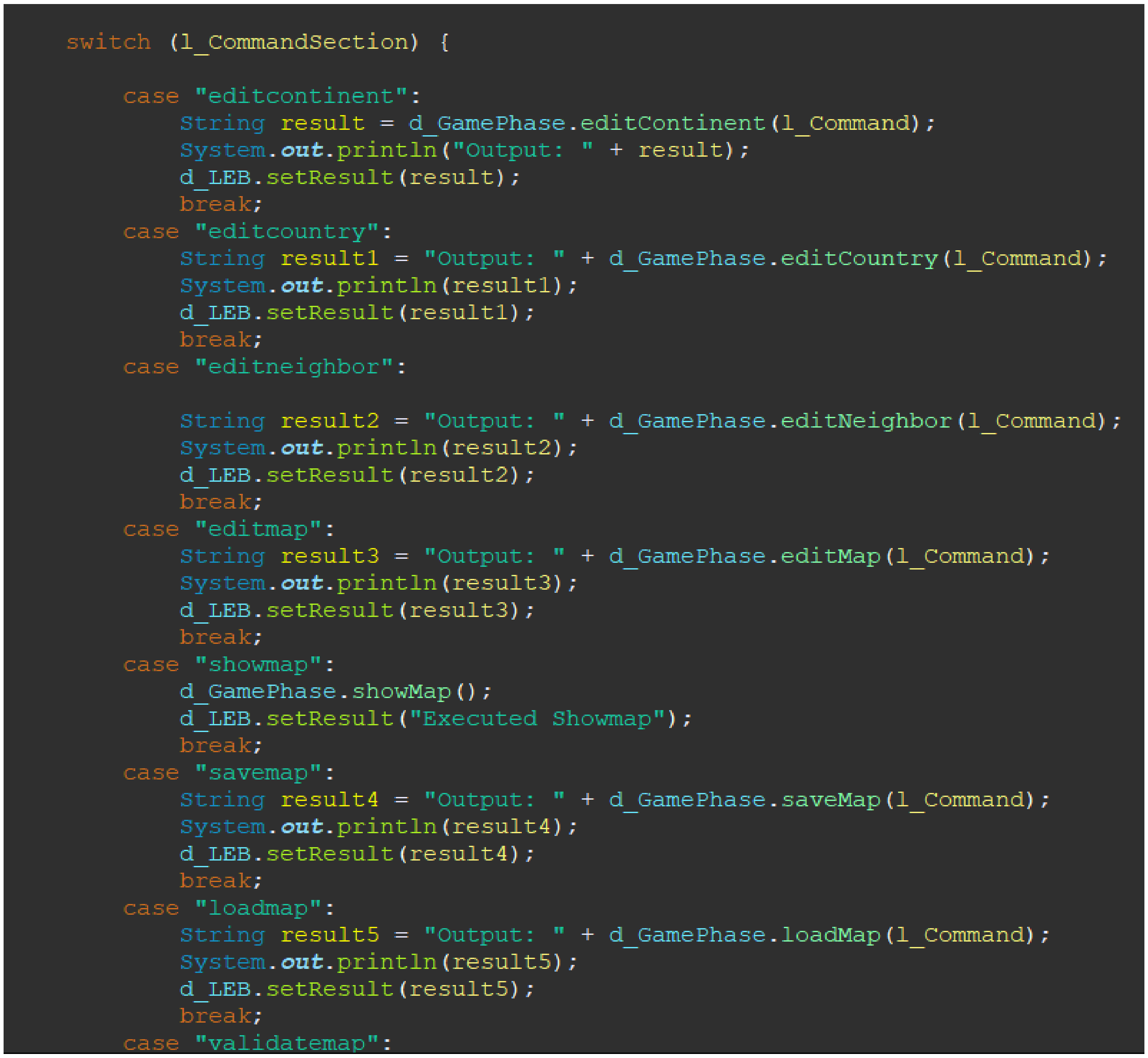


```
        String l_CommandSection = l_Command.split(" ")[0];

        switch (l_CommandSection) {

        case "editcontinent":
            if (l_PhaseController == 1) {
                System.out.println("Output: " + d_MapController.editContinentCommand(l_Command));
            } else {
                System.out.println("Invalid command in " + returnPhase(l_PhaseController));
            }
            break;
        case "editcountry":
            if (l_PhaseController == 1) {
                System.out.println("Output: " + d_MapController.editCountryCommand(l_Command));
            } else {
                System.out.println("Invalid command in " + returnPhase(l_PhaseController));
            }
            break;
        case "editneighbor":
            if (l_PhaseController == 1) {
                System.out.println("Output: " + d_MapController.editNeighborsCommand(l_Command));
            } else {
                System.out.println("Invalid command in " + returnPhase(l_PhaseController));
            }
            break;
        case "editmap":
            if (l_PhaseController == 1) {
                System.out.println("Output: " + d_MapController.editMap(l_Command));
            } else {
                System.out.println("Invalid command in " + returnPhase(l_PhaseController));
            }
```

After:



```
com.soen6441.risk_game_u14.state
    Edit.java
    ExecuteOrder.java
    Gameover.java
    IssueOrder.java
    Phase.java
    Reinforcement.java
    Startup.java
```



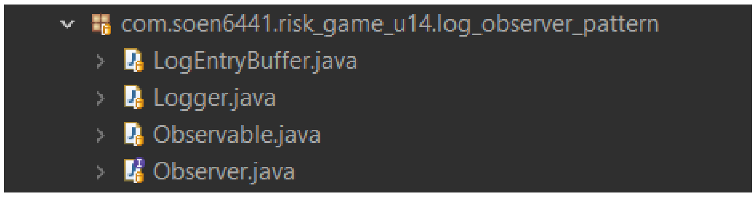
```
    switch (l_CommandSection) {

        case "editcontinent":
            String result = d_GamePhase.editContinent(l_Command);
            System.out.println("Output: " + result);
            d_LEB.setResult(result);
            break;
        case "editcountry":
            String result1 = "Output: " + d_GamePhase.editCountry(l_Command);
            System.out.println(result1);
            d_LEB.setResult(result1);
            break;
        case "editneighbor":

            String result2 = "Output: " + d_GamePhase.editNeighbor(l_Command);
            System.out.println(result2);
            d_LEB.setResult(result2);
            break;
        case "editmap":
            String result3 = "Output: " + d_GamePhase.editMap(l_Command);
            System.out.println(result3);
            d_LEB.setResult(result3);
            break;
        case "showmap":
            d_GamePhase.showMap();
            d_LEB.setResult("Executed Showmap");
            break;
        case "savemap":
            String result4 = "Output: " + d_GamePhase.saveMap(l_Command);
            System.out.println(result4);
            d_LEB.setResult(result4);
            break;
        case "loadmap":
            String result5 = "Output: " + d_GamePhase.loadMap(l_Command);
            System.out.println(result5);
            d_LEB.setResult(result5);
            break;
        case "validatemap":
```

2. Observer Pattern

Refactoring was needed due to the frequent alterations of values associated with generating a comprehensive log file containing phases, commands, and their effects throughout the game.

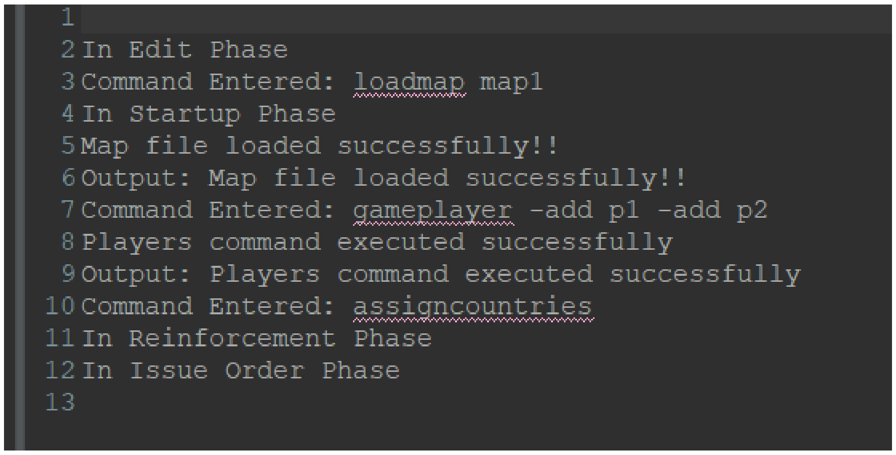
After the refactoring process, any time a change occurs in the state of a phase or command, all associated elements are promptly informed, and in this context, they are recorded in the log file alongside the relevant phase, command, and command effects.

Before:



```
com.soen6441.risk_game_u14.log_observer_pattern
    LogEntryBuffer.java
    Logger.java
    Observable.java
    Observer.java
```

After:



```

In Edit Phase
Command Entered: loadmap map1
In Startup Phase
Map file loaded successfully!!
Output: Map file loaded successfully!!
Command Entered: gameplayer -add p1 -add p2
Players command executed successfully
Output: Players command executed successfully
Command Entered: assigncountries
In Reinforcement Phase
In Issue Order Phase

```

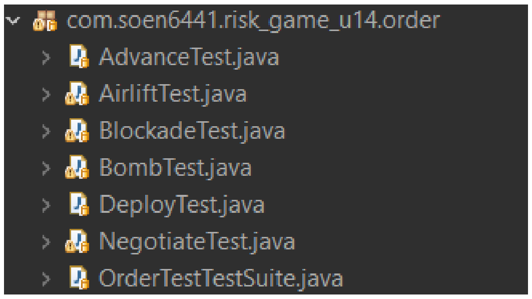
3. Command Pattern

The need for a refactoring operation arose from the requirement to handle various types of orders in contrast to the single order used in Build1.

Prior to the refactoring, a single order was stored in specific queues for each player, and the entire implementation was contained within the Order class.

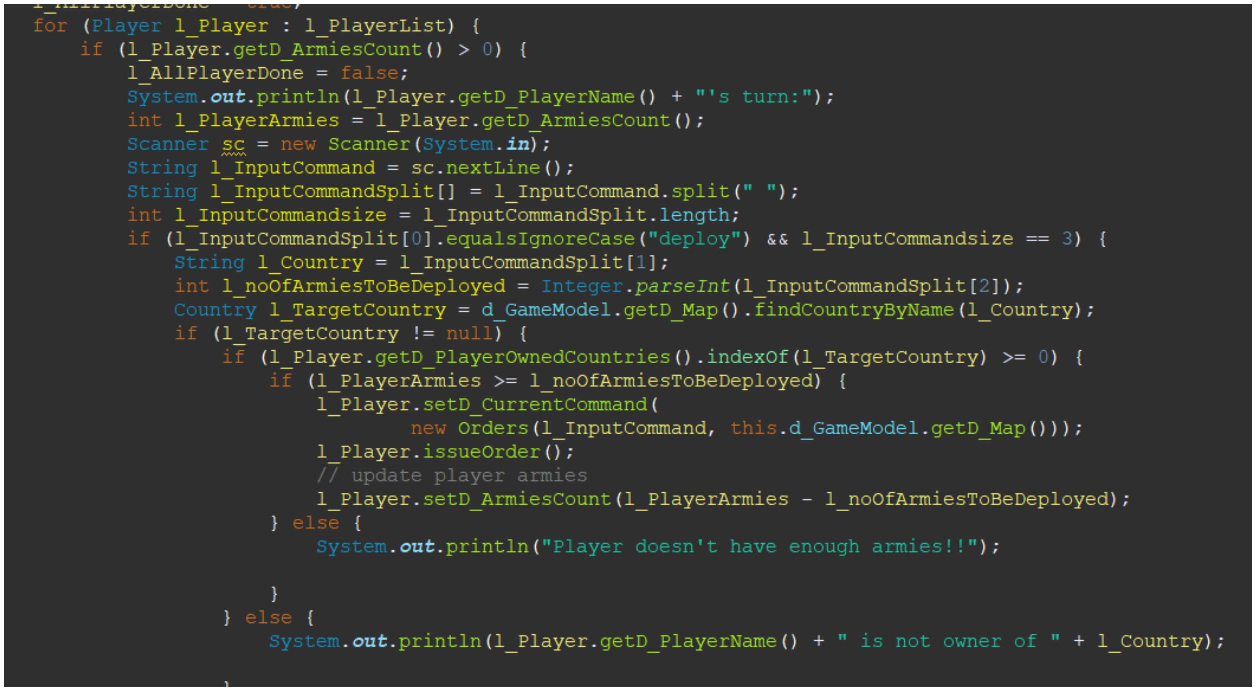
Following the refactoring, distinct classes were created based on the Order class, and specific execution methods were customized in each specialized Order class.

Test Classes:

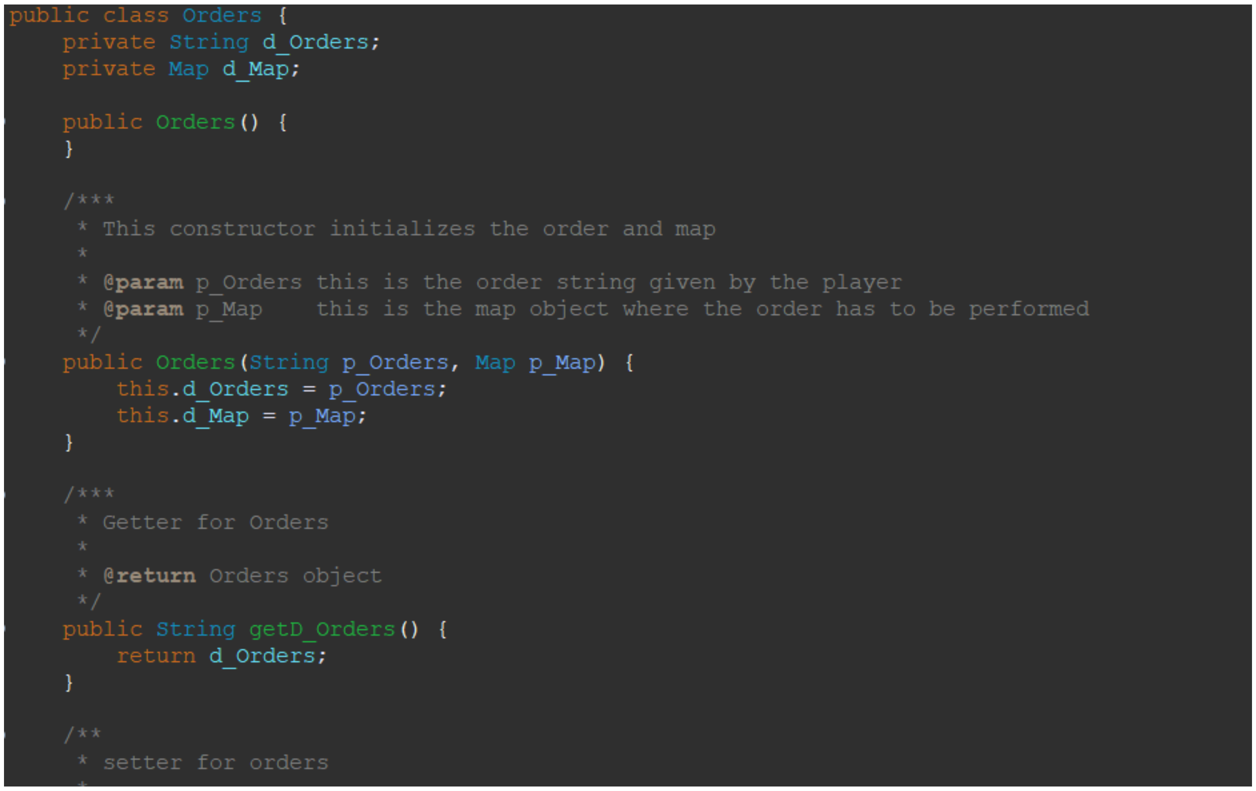


```
com.soen6441.risk_game_u14.order
    AdvanceTest.java
    AirliftTest.java
    BlockadeTest.java
    BombTest.java
    DeployTest.java
    NegotiateTest.java
    OrderTestTestSuite.java
```

Before:

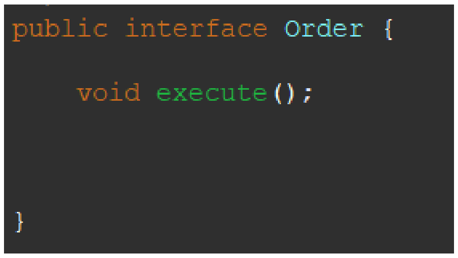


```
for (Player l_Player : l_PlayerList) {
    if (l_Player.getD_ArmiesCount() > 0) {
        l_AllPlayerDone = false;
        System.out.println(l_Player.getD_PlayerName() + "'s turn:");
        int l_PlayerArmies = l_Player.getD_ArmiesCount();
        Scanner sc = new Scanner(System.in);
        String l_InputCommand = sc.nextLine();
        String l_InputCommandSplit[] = l_InputCommand.split(" ");
        int l_InputCommandsize = l_InputCommandSplit.length;
        if (l_InputCommandSplit[0].equalsIgnoreCase("deploy") && l_InputCommandsize == 3) {
            String l_Country = l_InputCommandSplit[1];
            int l_noOfArmiesToBeDeployed = Integer.parseInt(l_InputCommandSplit[2]);
            Country l_TargetCountry = d_GameModel.getD_Map().findCountryByName(l_Country);
            if (l_TargetCountry != null) {
                if (l_Player.getD_PlayerOwnedCountries().indexOf(l_TargetCountry) >= 0) {
                    if (l_PlayerArmies >= l_noOfArmiesToBeDeployed) {
                        l_Player.setD_CurrentCommand(
                                new Orders(l_InputCommand, this.d_GameModel.getD_Map()));
                        l_Player.issueOrder();
                        // update player armies
                        l_Player.setD_ArmiesCount(l_PlayerArmies - l_noOfArmiesToBeDeployed);
                    } else {
                        System.out.println("Player doesn't have enough armies!!");

                    }
                } else {
                    System.out.println(l_Player.getD_PlayerName() + " is not owner of " + l_Country);
```

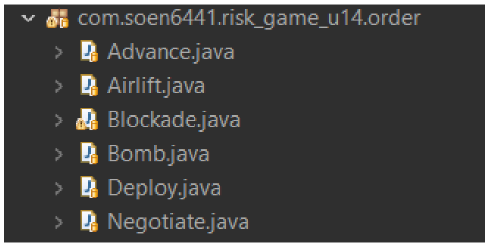


```
public class Orders {
    private String d_Orders;
    private Map d_Map;

    public Orders() {
    }

    /***
     * This constructor initializes the order and map
     *
     * @param p_Orders this is the order string given by the player
     * @param p_Map    this is the map object where the order has to be performed
     */
    public Orders(String p_Orders, Map p_Map) {
        this.d_Orders = p_Orders;
        this.d_Map = p_Map;
    }

    /***
     * Getter for Orders
     *
     * @return Orders object
     */
    public String getD_Orders() {
        return d_Orders;
    }

    /**
     * setter for orders
```

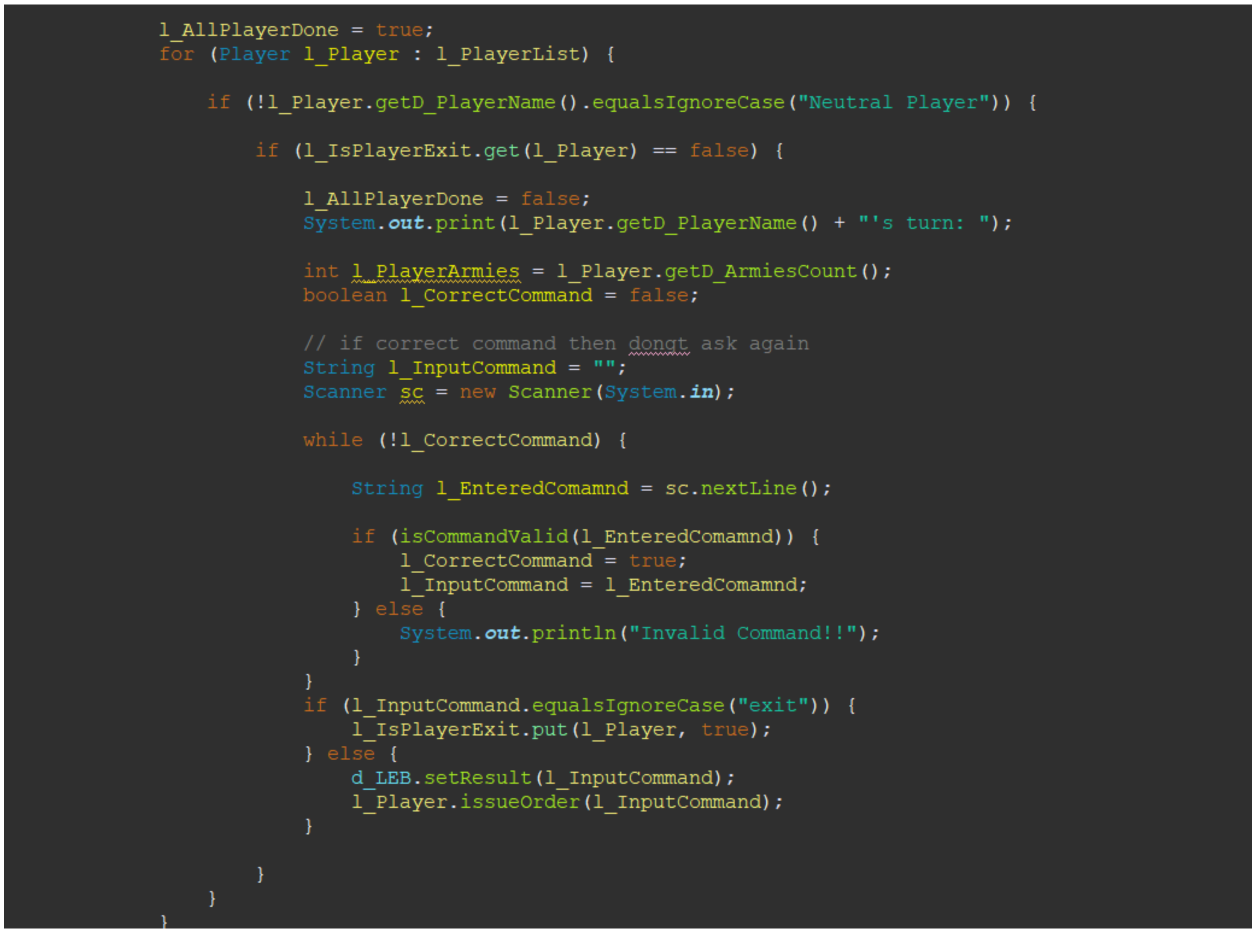
After:



```
public interface Order {

    void execute();


}
```



```
com.soen6441.risk_game_u14.order
    Advance.java
    Airlift.java
    Blockade.java
    Bomb.java
    Deploy.java
    Negotiate.java
```



```
        l_AllPlayerDone = true;
        for (Player l_Player : l_PlayerList) {

            if (!l_Player.getD_PlayerName().equalsIgnoreCase("Neutral Player")) {

                if (l_IsPlayerExit.get(l_Player) == false) {

                    l_AllPlayerDone = false;
                    System.out.print(l_Player.getD_PlayerName() + "'s turn: ");

                    int l_PlayerArmies = l_Player.getD_ArmiesCount();
                    boolean l_CorrectCommand = false;

                    // if correct command then dongt ask again
                    String l_InputCommand = "";
                    Scanner sc = new Scanner(System.in);

                    while (!l_CorrectCommand) {

                        String l_EnteredComamnd = sc.nextLine();

                        if (isCommandValid(l_EnteredComamnd)) {
                            l_CorrectCommand = true;
                            l_InputCommand = l_EnteredComamnd;
                        } else {
                            System.out.println("Invalid Command!!");
                        }
                    }
                    if (l_InputCommand.equalsIgnoreCase("exit")) {
                        l_IsPlayerExit.put(l_Player, true);
                    } else {
                        d_LEB.setResult(l_InputCommand);
                        l_Player.issueOrder(l_InputCommand);
                    }

                }
            }
        }
```

4. Made a separate method isValid() in Player Controller Class for command validation from issueOrder() loop

A new method named `isValid()` was introduced to assess the validity of commands issued by players. In the initial version, this functionality was integrated within the `issueOrder()` method, as there was only one type of order to be implemented (i.e., "deploy") in Build 1. This refactoring effort resulted in improved code readability and modularity, enabling us to verify the syntax of various order commands

Test Class:

AdvanceTest

DeployTest

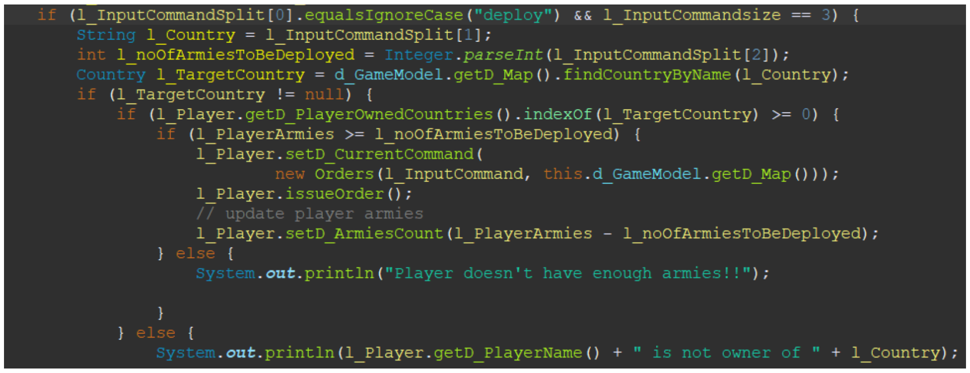
BombTest

BlockadeTest

NegotiateTest

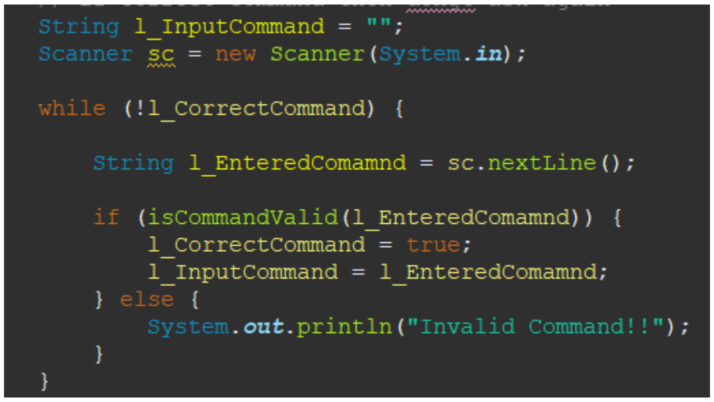
AirliftTest

Before:

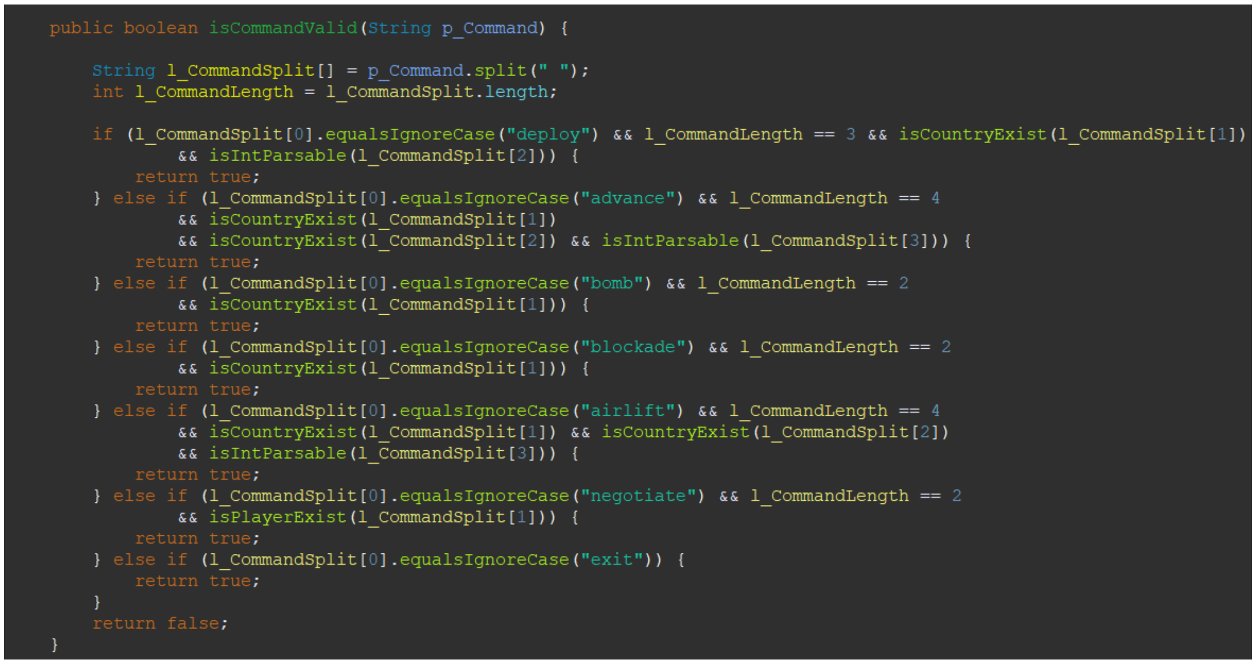


```
if (l_InputCommandSplit[0].equalsIgnoreCase("deploy") && l_InputCommandsize == 3) {
    String l_Country = l_InputCommandSplit[1];
    int l_noOfArmiesToBeDeployed = Integer.parseInt(l_InputCommandSplit[2]);
    Country l_TargetCountry = d_GameModel.getD_Map().findCountryByName(l_Country);
    if (l_TargetCountry != null) {
        if (l_Player.getD_PlayerOwnedCountries().indexOf(l_TargetCountry) >= 0) {
            if (l_PlayerArmies >= l_noOfArmiesToBeDeployed) {
                l_Player.setD_CurrentCommand(
                        new Orders(l_InputCommand, this.d_GameModel.getD_Map()));
                l_Player.issueOrder();
                // update player armies
                l_Player.setD_ArmiesCount(l_PlayerArmies - l_noOfArmiesToBeDeployed);
            } else {
                System.out.println("Player doesn't have enough armies!!");

            }
        } else {
            System.out.println(l_Player.getD_PlayerName() + " is not owner of " + l_Country);
```

After :



```
String l_InputCommand = "";
Scanner sc = new Scanner(System.in);

while (!l_CorrectCommand) {

    String l_EnteredComamnd = sc.nextLine();

    if (isCommandValid(l_EnteredComamnd)) {
        l_CorrectCommand = true;
        l_InputCommand = l_EnteredComamnd;
    } else {
        System.out.println("Invalid Command!!");
    }
}
```



```java
public boolean isCommandValid(String p_Command) {

    String l_CommandSplit[] = p_Command.split(" ");
    int l_CommandLength = l_CommandSplit.length;

    if (l_CommandSplit[0].equalsIgnoreCase("deploy") && l_CommandLength == 3 && isCountryExist(l_CommandSplit[1])
            && isIntParsable(l_CommandSplit[2])) {
        return true;
    } else if (l_CommandSplit[0].equalsIgnoreCase("advance") && l_CommandLength == 4
            && isCountryExist(l_CommandSplit[1])
            && isCountryExist(l_CommandSplit[2]) && isIntParsable(l_CommandSplit[3])) {
        return true;
    } else if (l_CommandSplit[0].equalsIgnoreCase("bomb") && l_CommandLength == 2
            && isCountryExist(l_CommandSplit[1])) {
        return true;
    } else if (l_CommandSplit[0].equalsIgnoreCase("blockade") && l_CommandLength == 2
            && isCountryExist(l_CommandSplit[1])) {
        return true;
    } else if (l_CommandSplit[0].equalsIgnoreCase("airlift") && l_CommandLength == 4
            && isCountryExist(l_CommandSplit[1]) && isCountryExist(l_CommandSplit[2])
            && isIntParsable(l_CommandSplit[3])) {
        return true;
    } else if (l_CommandSplit[0].equalsIgnoreCase("negotiate") && l_CommandLength == 2
            && isPlayerExist(l_CommandSplit[1])) {
        return true;
    } else if (l_CommandSplit[0].equalsIgnoreCase("exit")) {
        return true;
    }
    return false;
}
```

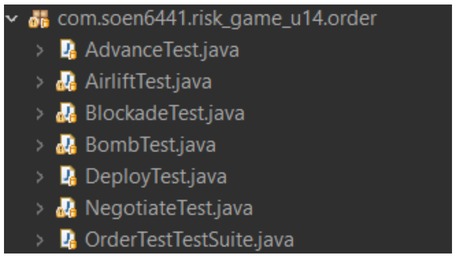
5. Added fields in Player Class to implement new functionalities

New fields were introduced as part of the refactoring process, including:

a. d_SkipCommands: This field was incorporated to enable the skipping of player commands in the event of logical errors.
b. d_GameModel: This field was added to access country and continent objects within the player class, facilitating the interaction with the map.
c. d_NegotiatedPlayers: A new field was introduced to maintain a list of players involved in negotiations, contributing to the implementation of the Negotiate card feature.
d. d_AtLeastOneBattleWon: This field was included to implement a logic that involves randomly obtaining a card when a player captures a country.
e. d_Cards: This field was introduced to manage and keep track of the cards owned by the player.

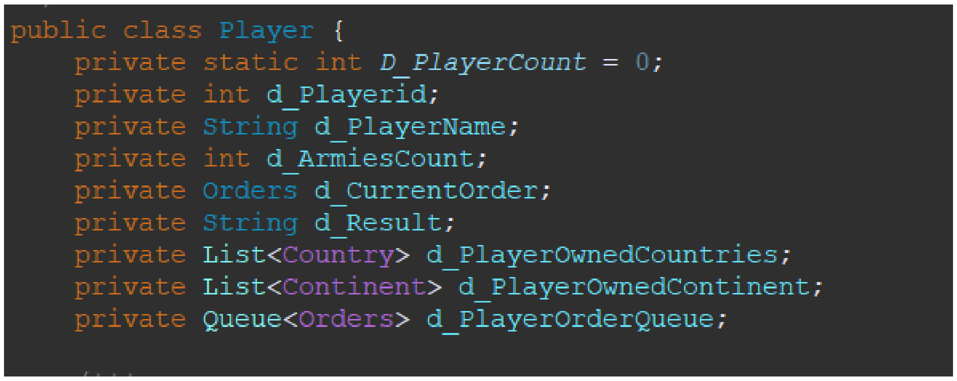
These fields were not necessary in Build 1 as there were no special orders to be implemented at that time. The introduction of these fields in the refactoring process enhances the flexibility and functionality of the codebase.

Test Class:



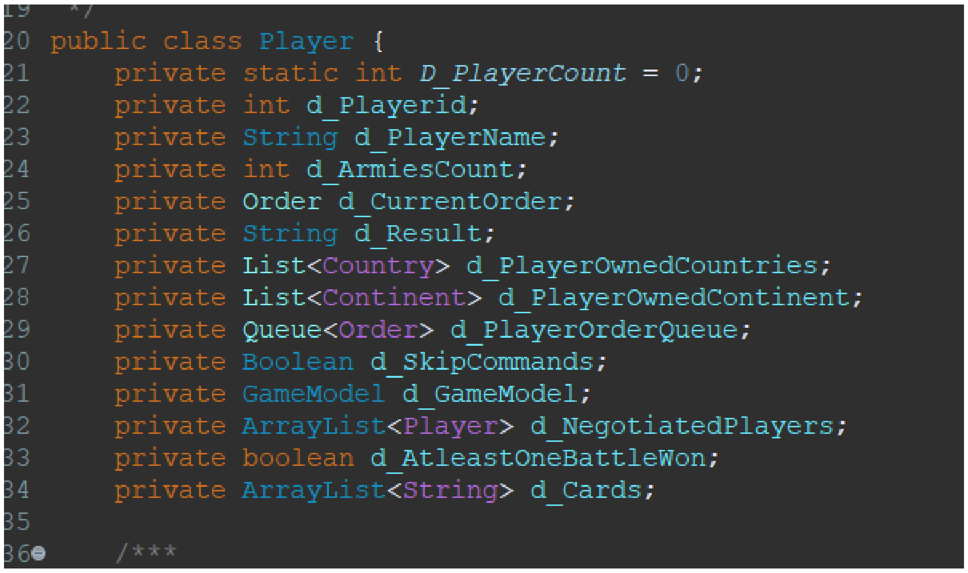
```
com.soen6441.risk_game_u14.order
  AdvanceTest.java
  AirliftTest.java
  BlockadeTest.java
  BombTest.java
  DeployTest.java
  NegotiateTest.java
  OrderTestTestSuite.java
```

Before:



```java
public class Player {
    private static int D_PlayerCount = 0;
    private int d_Playerid;
    private String d_PlayerName;
    private int d_ArmiesCount;
    private Orders d_CurrentOrder;
    private String d_Result;
    private List<Country> d_PlayerOwnedCountries;
    private List<Continent> d_PlayerOwnedContinent;
    private Queue<Orders> d_PlayerOrderQueue;
```

After:

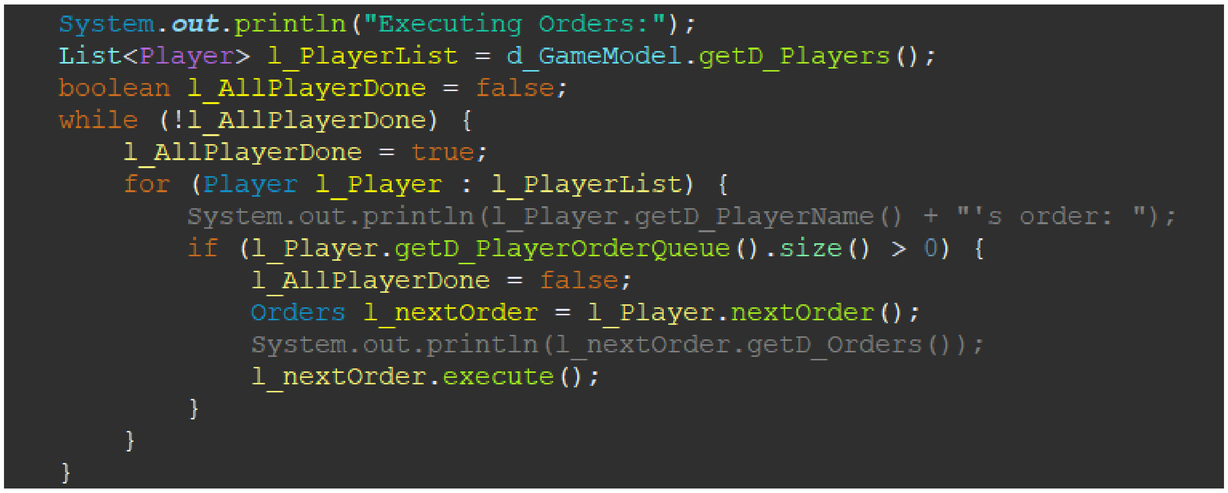


```java
 */
public class Player {
    private static int D_PlayerCount = 0;
    private int d_Playerid;
    private String d_PlayerName;
    private int d_ArmiesCount;
    private Order d_CurrentOrder;
    private String d_Result;
    private List<Country> d_PlayerOwnedCountries;
    private List<Continent> d_PlayerOwnedContinent;
    private Queue<Order> d_PlayerOrderQueue;
    private Boolean d_SkipCommands;
    private GameModel d_GameModel;
    private ArrayList<Player> d_NegotiatedPlayers;
    private boolean d_AtleastOneBattleWon;
    private ArrayList<String> d_Cards;

    /***
```

6. Implemented new logic for game termination

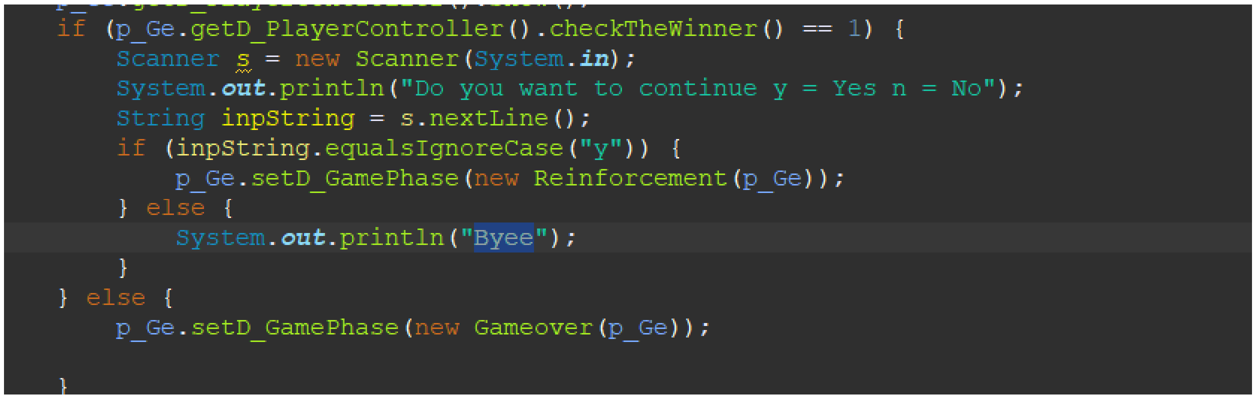
In the prior build, the game's termination condition stipulated that all players must exhaust their reinforcement pools and have zero armies remaining. However, in this latest build, this condition was modified to declare the player who conquers all the countries on the map as the winner. To accommodate this change, a new algorithm was developed to verify this condition and facilitate the transition to the "game over" phase.

Before:

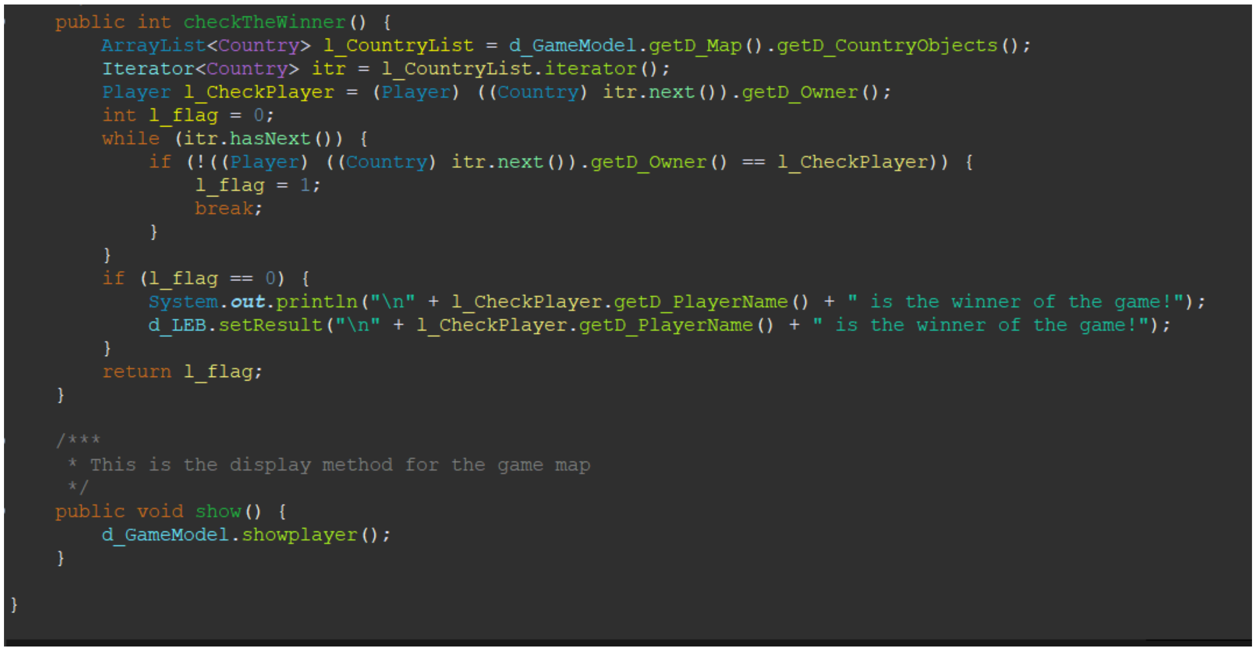


```java
System.out.println("Executing Orders:");
List<Player> l_PlayerList = d_GameModel.getD_Players();
boolean l_AllPlayerDone = false;
while (!l_AllPlayerDone) {
    l_AllPlayerDone = true;
    for (Player l_Player : l_PlayerList) {
        System.out.println(l_Player.getD_PlayerName() + "'s order: ");
        if (l_Player.getD_PlayerOrderQueue().size() > 0) {
            l_AllPlayerDone = false;
            Orders l_nextOrder = l_Player.nextOrder();
            System.out.println(l_nextOrder.getD_Orders());
            l_nextOrder.execute();
        }
    }
}
```

After:



```java
if (p_Ge.getD_PlayerController().checkTheWinner() == 1) {
    Scanner s = new Scanner(System.in);
    System.out.println("Do you want to continue y = Yes n = No");
    String inpString = s.nextLine();
    if (inpString.equalsIgnoreCase("y")) {
        p_Ge.setD_GamePhase(new Reinforcement(p_Ge));
    } else {
        System.out.println("Byee");
    }
} else {
    p_Ge.setD_GamePhase(new Gameover(p_Ge));

}
```

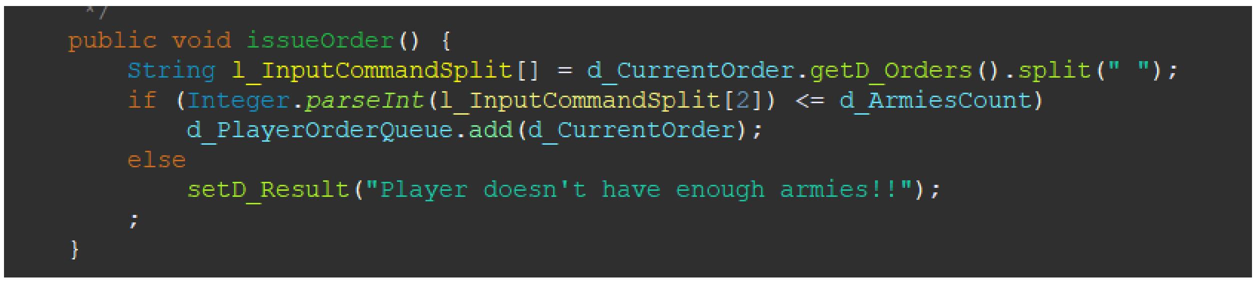


```java
    public int checkTheWinner() {
        ArrayList<Country> l_CountryList = d_GameModel.getD_Map().getD_CountryObjects();
        Iterator<Country> itr = l_CountryList.iterator();
        Player l_CheckPlayer = (Player) ((Country) itr.next()).getD_Owner();
        int l_flag = 0;
        while (itr.hasNext()) {
            if (!((Player) ((Country) itr.next()).getD_Owner() == l_CheckPlayer)) {
                l_flag = 1;
                break;
            }
        }
        if (l_flag == 0) {
            System.out.println("\n" + l_CheckPlayer.getD_PlayerName() + " is the winner of the game!");
            d_LEB.setResult("\n" + l_CheckPlayer.getD_PlayerName() + " is the winner of the game!");
        }
        return l_flag;
    }

    /***
     * This is the display method for the game map
     */
    public void show() {
        d_GameModel.showplayer();
    }

}
```

7. IssueOrder() method signature change

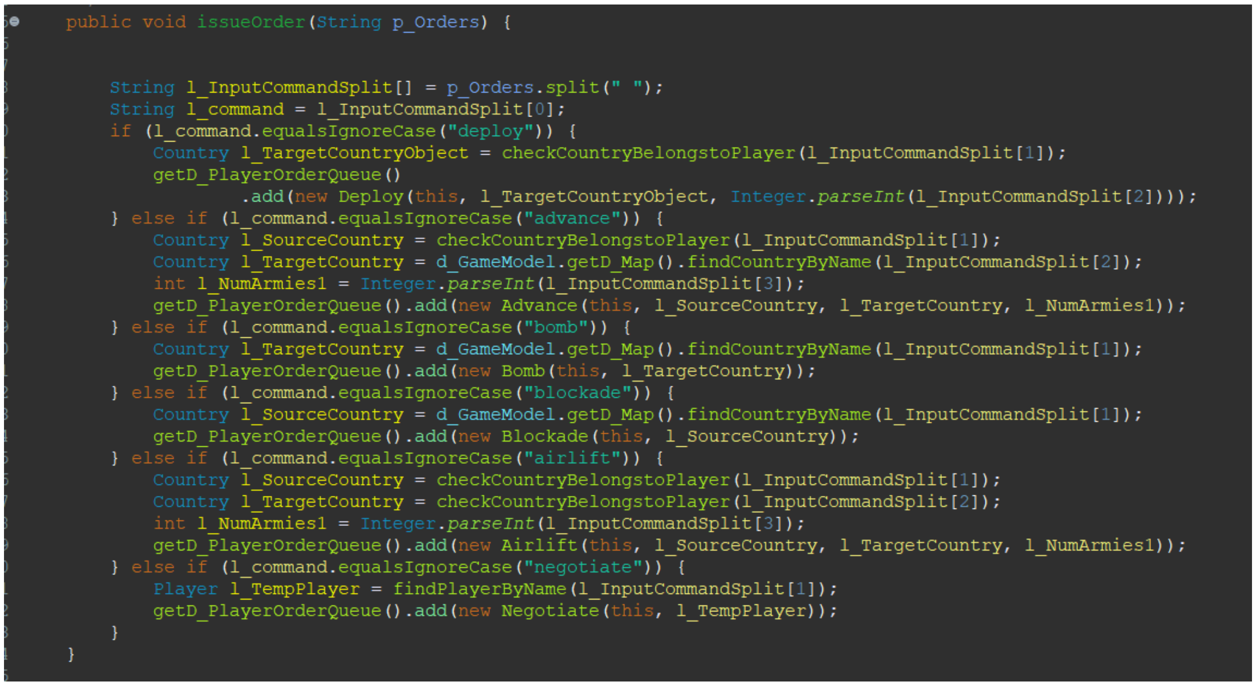
In the Player class, the `issueOrder` method originally lacked any arguments to accommodate various types of orders. However, with the introduction of multiple order types and the requirement to create order objects based on their type in the player queue, it became imperative to refactor the `issueOrder` method.

Before:



```java
     */
    public void issueOrder() {
        String l_InputCommandSplit[] = d_CurrentOrder.getD_Orders().split(" ");
        if (Integer.parseInt(l_InputCommandSplit[2]) <= d_ArmiesCount)
            d_PlayerOrderQueue.add(d_CurrentOrder);
        else
            setD_Result("Player doesn't have enough armies!!");
        ;
    }
```

After:



```
    public void issueOrder(String p_Orders) {


        String l_InputCommandSplit[] = p_Orders.split(" ");
        String l_command = l_InputCommandSplit[0];
        if (l_command.equalsIgnoreCase("deploy")) {
            Country l_TargetCountryObject = checkCountryBelongstoPlayer(l_InputCommandSplit[1]);
            getD_PlayerOrderQueue()
                    .add(new Deploy(this, l_TargetCountryObject, Integer.parseInt(l_InputCommandSplit[2])));
        } else if (l_command.equalsIgnoreCase("advance")) {
            Country l_SourceCountry = checkCountryBelongstoPlayer(l_InputCommandSplit[1]);
            Country l_TargetCountry = d_GameModel.getD_Map().findCountryByName(l_InputCommandSplit[2]);
            int l_NumArmies1 = Integer.parseInt(l_InputCommandSplit[3]);
            getD_PlayerOrderQueue().add(new Advance(this, l_SourceCountry, l_TargetCountry, l_NumArmies1));
        } else if (l_command.equalsIgnoreCase("bomb")) {
            Country l_TargetCountry = d_GameModel.getD_Map().findCountryByName(l_InputCommandSplit[1]);
            getD_PlayerOrderQueue().add(new Bomb(this, l_TargetCountry));
        } else if (l_command.equalsIgnoreCase("blockade")) {
            Country l_SourceCountry = d_GameModel.getD_Map().findCountryByName(l_InputCommandSplit[1]);
            getD_PlayerOrderQueue().add(new Blockade(this, l_SourceCountry));
        } else if (l_command.equalsIgnoreCase("airlift")) {
            Country l_SourceCountry = checkCountryBelongstoPlayer(l_InputCommandSplit[1]);
            Country l_TargetCountry = checkCountryBelongstoPlayer(l_InputCommandSplit[2]);
            int l_NumArmies1 = Integer.parseInt(l_InputCommandSplit[3]);
            getD_PlayerOrderQueue().add(new Airlift(this, l_SourceCountry, l_TargetCountry, l_NumArmies1));
        } else if (l_command.equalsIgnoreCase("negotiate")) {
            Player l_TempPlayer = findPlayerByName(l_InputCommandSplit[1]);
            getD_PlayerOrderQueue().add(new Negotiate(this, l_TempPlayer));
        }
    }
```