Ahmad Hasan ——————————————————— CS (40294488)

Diego Palacios ——————————————————— SE (40270954)

Handika Harianto Ew Jong ——————————————— CS (40307322)

Anosh Kurian Vadakkeparampil ——————————— CS (40303184)

Ruqaiah Mohammed Abdo Hadwan ——————————— CS (40324800)

Potential Refactoring Targets

1. **Order.java Class**
    - The current implementation utilizes a switch-case structure to handle different order types, which violates the Open/Closed Principle and makes the system resistant to extension with new order types.
2. **Player.java Class**
    - Exposes internal mutable state through public fields and getters that return direct references to collections, compromising encapsulation and making the class vulnerable to external modifications.
3. **GameEngine.java Class**
    - Functions as an architectural hub with excessive responsibilities, managing game initialization, turn sequencing, order processing, and state transitions in a single monolithic class.
4. **MapReader.java Class**
    - Combines multiple concerns including file I/O operations, map parsing, and topological validation, resulting in a class that is difficult to maintain and test in isolation.
5. **Command.java Class**
    - Contains repetitive patterns for command flag parsing and argument processing that could be abstracted into reusable components to reduce code duplication.
6. **InputOutput.java Class**
    - Centralizes all user interface interactions, creating a bottleneck for modifications and making it challenging to implement alternative interfaces.
7. **MainMenu.java Class**
    - Embeds menu options and navigation logic directly in string literals, reducing flexibility for internationalization and dynamic menu generation.
8. **MapEditor.java Class**
    - Blurs the separation between map modification operations and validation logic, making it difficult to modify validation rules independently.
9. **Continent.java Class**
    - Provides unrestricted access to its country collection through a getter method, allowing external code to bypass intended business rules.
10. **Country.java Class**

    - Contains public setters for critical relationships (owner, neighbors) that should be managed through controlled domain operations.
11. **GameEngineTest.java Class**
    - Duplicates complex game logic calculations that exist in production code, creating maintenance overhead and potential inconsistency.
12. **MapTest.java Class**
    - Relies on concrete file system paths and specific map files, making tests brittle and environment-dependent.
13. **CommandTest.java Class**
    - Exhibits repetitive test structures and assertions that could be streamlined with parameterized testing approaches.
14. **MapWriter.java Class**
    - Hardcodes metadata and formatting details directly in the class, making it difficult to modify output formats without code changes.
15. **Country.java Class (Secondary Issue)**
    - Includes defensive null checks in the toString() method that suggest potential design issues with object initialization and lifecycle management.

Actual Refactoring Targets

- **Order.java Class**
  The previous Order class was structured to contain the logic of the execution of all orders (Deploy and Advance until now) which could've caused issues to the efficiency and performance of the program.

```
10    public class Order {
28        }
29
30        /**
31         * Deploy armies at d_countryName
32         */
33        public void execute(){
34            // TODO: Implement this method
35            switch (d_orderType.toLowerCase()) {
36                case "deploy":
37                    executeDeploy();
38                    break;
39                    case "advance":
40                        executeAdvance();
41                        break;
42                    // Add more cases here for future orders like "attack", "bomb", etc.
43                    default:
44                        System.out.println("Unknown order type: " + d_orderType);
45            }
46        }
47
```

  A switch case was being deployed in the execute() method to differentiate the nature of the orders and be able to identify which order the program needed to execute based on the type.

```java
public class Order {
    /**
     * executeDeploy method
     * Executes the deployment of orders issues by the players
     * Makes basic verifications if the player is able to execute the order based on how many reinforcements wants to se
     */
    private void executeDeploy() {
        System.out.println(d_player.getName() + " is deploying " + d_numArmy + " armies to country " + d_countryName);

        // Ensure the player owns the country before deploying
        if (!d_player.ownsCountry(d_countryName)) {
            System.out.println("Error: Player does not own country " + d_countryName);
            return;
        }

        // Ensure the player has enough reinforcements
        if (d_player.getReinforcements() < d_numArmy) {
            System.out.println("Error: Not enough armies in reinforcement pool.");
            return;
        }

        // Deploy armies
        d_player.removeReinforcement(d_numArmy);
        d_player.addArmiesToCountry(d_countryName, d_numArmy);
        System.out.println("Successfully deployed " + d_numArmy + " armies to country " + d_countryName);
    }

```

The executeDeploy() method contained the Deploy logic, everything inside the Order class.
We ensured to optimize the design and architecture of this class, the result of the refactor is the following:

```java
public abstract class Order {
    */
    public Order(String p_orderType, String p_countryName, int p_numArmy, Player p_player) {
        this.d_orderType = p_orderType;
        this.d_countryName = p_countryName;
        this.d_numArmy = p_numArmy;
        this.d_player = p_player;
    }

    /**
     * Deploy armies at d_countryName
     */
    public abstract void execute();

```

Order class was converted into an abstract class to act as a skeleton for the structure of both Deploy and Advance.

```java
package com.gameplay;

public class Deploy extends Order{

    public Deploy(String p_orderType, String p_countryName, int p_numArmy, Player p_player){
        super(p_orderType, p_countryName, p_numArmy, p_player);
    }

    @Override
    public void execute(){
        System.out.println(d_player.getName() + " is deploying " + d_numArmy + " armies to country " + d_countryName);

        // Ensure the player owns the country before deploying
        if (!d_player.ownsCountry(d_countryName)) {
            System.out.println("Error: Player does not own country " + d_countryName);
            return;
        }

        // Ensure the player has enough reinforcements
        if (d_player.getReinforcements() < d_numArmy) {
            System.out.println("Error: Not enough armies in reinforcement pool.");
            return;
        }

        // Deploy armies
        //        d_player.removeReinforcement(d_numArmy);
        d_player.addArmiesToCountry(d_countryName, d_numArmy);
        System.out.println("Successfully deployed " + d_numArmy + " armies to country " + d_countryName);
        System.out.println("--------------------------------------------------------------------------------");
    }
```

```java
public class Advance extends Order{

    public Advance(String p_orderType, String p_countryFrom, String p_countryTo , int p_numArmy, Player p_player){
        super(p_orderType, p_countryFrom, p_numArmy, p_player);
        this.p_countryTo = p_countryTo;
    }

    @Override
    public void execute(){
        System.out.println(d_player.getName() + " is moving " + d_numArmy + " armies from country " + d_countryName + " to country " + p_countryTo);
        Country d_sourceCountry = d_player.getCountryByName(d_countryName);
        Country d_destiantionCountry = d_player.getCountryByName(p_countryTo);

        // Ensure the player owns the country before advancing
        if (!d_player.ownsCountry(d_countryName)) {
            System.out.println("Error: Player does not own country " + d_countryName);
            return;
        }

        // Ensure the country has enough reinforcements
        if (d_sourceCountry.getArmies() < d_numArmy) {
            System.out.println("Error: Not enough armies in available in the country.");
            return;
        }

        //Ensure both countries are adjacent
        if (!d_sourceCountry.isNeighbor(p_countryTo)){
            System.out.println("Error: Countries are not adjacent");
            return;
        }

        //Advance logic
        if (d_player.ownsCountry(p_countryTo)){
            d_sourceCountry.removeReinforcements(d_numArmy);
            d_destiantionCountry.addReinforcements(d_numArmy);
            d_destiantionCountry.setOwner(d_player);
            d_player.d_ownedCountries.add(d_destiantionCountry);
            System.out.println("Successfully moved " + d_numArmy + " armies from country " + d_countryName + " to country " + p_countryTo);
            System.out.println("-----------------------------------------------------------------------------");
        }else{
            //Battle Logic if the destination country is owned by another player
            System.out.println("Starting attack from country " + d_countryName + " to country " + p_countryTo);
            Country d_defenderCountry = d_sourceCountry.getNeighborByName(p_countryTo);
            Player p_defenderPlayer = d_defenderCountry.getOwner();

            Random random = new Random();
            int d_attackingArmies = d_numArmy;
            int d_defendingArmies = d_defenderCountry.getArmies();

            //Kill count to decide a winner after the attack
            int d_attackersKilled = 0;
            int d_defendersKilled = 0;
```

Both Deploy and Advance classes even if they are Order objects, they keep their own logic and functionalities inside each of them. This refactoring decision guarantees a better encapsulation performance and allows for a truly well implemented single responsibility principle creating different Order objects but each one with their own behavior.

- **Relevant Test Cases:**

```java
/**
 * Test a player does not own the source country
 */
@Test
public void playerDoesNotOwnSourceCountry() {
    System.out.println(x:"\nTEST : Player doesn't own source country");

    Country l_countryFrom = this.d_player2.getOwnedCountries().getFirst();
    Country l_countryTo = this.d_player2.getOwnedCountries().getLast();

    Advance l_advanceOrder = new Advance(this.d_gameEngine, this.d_player1, l_countryFrom.getName(), l_countryTo.getName(), p_numArmies:2);
    assertFalse(l_advanceOrder.isValid());
}

/**
 * Test the source and target country are the same
 */
@Test
public void sourceAndTargetCountryTheSame() {
    System.out.println(x:"\nTEST : Source an target country are the same");

    Country l_countryFrom = this.d_player1.getOwnedCountries().getFirst();
    Country l_countryTo = this.d_player1.getOwnedCountries().getFirst();

    Advance l_advanceOrder = new Advance(this.d_gameEngine, this.d_player1, l_countryFrom.getName(), l_countryTo.getName(), p_numArmies:2);
    assertFalse(l_advanceOrder.isValid());
}
```

```java
/**
 * Test source country does not have enough armies
 */
@Test
public void notEnoughArmies() {
    System.out.println(x:"\nTEST : Source country does not have enough armies");

    Country l_countryFrom = this.d_player1.getOwnedCountries().getFirst();

    // Get player 1's adjacent countries
    List<Country> l_adjacentCountries = new ArrayList<>();
    for (Country l_country : this.d_player1.getOwnedCountries()) {
        for (Country l_adjCountry : l_country.getNeighbors()) {
            if (!this.d_player1.ownsCountry(l_adjCountry.getName())) {
                l_adjacentCountries.add(l_adjCountry);
            }
        }
    }
    Country l_countryTo = l_adjacentCountries.getFirst();

    Advance l_advanceOrder = new Advance(this.d_gameEngine, this.d_player1, l_countryFrom.getName(), l_countryTo.getName(), p_numArmies:15);
    assertFalse(l_advanceOrder.isValid());
}

/**
 * Test source and target countries are not adjacent
 */
@Test
public void SourceAndTargetCountryAdjacentOrNot() {
    System.out.println(x:"\nTEST : Source and target country are not adjacent to each other");

    Country l_countryFrom = this.d_player1.getOwnedCountries().getFirst();
    l_countryFrom.setArmies(p_armies:10);

    Country l_countryTo = this.d_player2.getOwnedCountries().getFirst();
    l_countryTo.setArmies(p_armies:10);

    Advance l_advanceOrder = new Advance(this.d_gameEngine, this.d_player1, l_countryFrom.getName(), l_countryTo.getName(), p_numArmies:2);

    if (l_countryFrom.isNeighbor(l_countryTo.getName())) {
        System.out.println(x:"\nSource and target country are adjacent");
        assertTrue(l_advanceOrder.isValid());
    } else {
        assertFalse(l_advanceOrder.isValid());
    }
}
```

```java
/**
 * Test attacker winning and successfully occupies the target country
 */
@Test
public void AttackerWinning() {
    System.out.println(x:"\nTEST : Attacker successfully capture the defender country");

    Country l_countryFrom = this.d_player1.getOwnedCountries().getFirst();
    l_countryFrom.setArmies(p_armies:10);

    List<Country> l_adjacentCountries = new ArrayList<>();
    for (Country l_country : l_countryFrom.getNeighbors()) {
        if (!l_country.getOwner().getName().equals(this.d_player1.getName())) {
            l_adjacentCountries.add(l_country);
        }
    }

    if (l_adjacentCountries.isEmpty()) {
        System.out.println("\n" + l_countryFrom.getName() + " has no adjacent countries");
        return;
    }

    Country l_countryTo = l_adjacentCountries.getFirst();
    l_countryTo.setArmies(p_armies:5);

    Advance l_advanceOrder = new Advance(this.d_gameEngine, this.d_player1, l_countryFrom.getName(), l_countryTo.getName(), p_numArmies:10);
    l_advanceOrder.execute();

    assertEquals(0, l_countryFrom.getArmies());
    assertEquals(6, l_countryTo.getArmies());
}
```

```java
/**
 * Test defender successfully defends the country
 */
@Test
public void DefenderWinning() {
    System.out.println(x:"\nTEST : Defender successfully defends the country");

    Country l_countryFrom = this.d_player1.getOwnedCountries().getFirst();
    l_countryFrom.setArmies(p_armies:26);

    List<Country> l_adjacentCountries = new ArrayList<>();
    for (Country l_country : l_countryFrom.getNeighbors()) {
        if (!l_country.getOwner().getName().equals(this.d_player1.getName())) {
            l_adjacentCountries.add(l_country);
        }
    }

    if (l_adjacentCountries.isEmpty()) {
        System.out.println("\n" + l_countryFrom.getName() + " has no adjacent countries");
        return;
    }

    Country l_countryTo = l_adjacentCountries.getFirst();
    l_countryTo.setArmies(p_armies:20);

    Advance l_advanceOrder = new Advance(this.d_gameEngine, this.d_player1, l_countryFrom.getName(), l_countryTo.getName(), p_numArmies:25);
    l_advanceOrder.execute();

    assertEquals(12, l_countryFrom.getArmies());
    assertEquals(5, l_countryTo.getArmies());
}
```

```java
/**
 * Test a player does not have an airlift card.
 */
@Test
public void noAirliftCard() {
    Airlift l_airliftOrder = new Airlift(this.d_player1, p_sourceCountryName:"Peru", p_targetCountryName:"Peru", p_n…5);
    assertFalse(l_airliftOrder.isValid());
}

/**
 * Test a player performs airlift with the same source and target country.
 */
@Test
public void sameSourceAndTargetCountry() {
    System.out.println(x:"\nTEST : Player performs airlift with the same source and target country");

    this.d_player1.getCards().add(Card.AIRLIFT);

    Airlift l_airliftOrder = new Airlift(this.d_player1, p_sourceCountryName:"Peru", p_targetCountryName:"Peru", p_n…5);

    assertFalse(l_airliftOrder.isValid());
}

/**
 * Test a player does not own source country
 */
@Test
public void playerDoesNotOwnSourceCountry() {
    System.out.println(x:"\nTEST : Player performs airlift from the source country that is not owned by the player");

    this.d_player1.getCards().add(Card.AIRLIFT);

    Country l_sourceCountry = this.d_player2.getOwnedCountries().getFirst();
    Country l_targetCountry = this.d_player1.getOwnedCountries().getFirst();

    Airlift l_airliftOrder = new Airlift(this.d_player1, l_sourceCountry.getName(), l_targetCountry.getName(), p_numArmy:5);

    assertFalse(l_airliftOrder.isValid());
}
```

```java
/**
 * Test a player does not own target country
 */
@Test
public void insufficientArmies() {
    System.out.println(x:"\nTEST : Player performs airlift where the source country does not have sufficient armies");

    this.d_player1.getCards().add(Card.AIRLIFT);

    Country l_sourceCountry = this.d_player1.getOwnedCountries().getFirst();
    Country l_targetCountry = this.d_player1.getOwnedCountries().getLast();

    Airlift l_airliftOrder = new Airlift(this.d_player1, l_sourceCountry.getName(), l_targetCountry.getName(), p_numArmy:10);

    assertFalse(l_airliftOrder.isValid());
}

/**
 * Test a player performs airlift successfully
 */
@Test
public void successfulAirlift() {
    System.out.println(x:"\nTEST : Player performs airlift successfully");

    this.d_player1.getCards().add(Card.AIRLIFT);

    Country l_sourceCountry = this.d_player1.getOwnedCountries().getFirst();
    Country l_targetCountry = this.d_player1.getOwnedCountries().getLast();

    int l_sourceArmies = 10;
    int l_targetArmies = 3;
    int l_numArmies = 5;

    l_sourceCountry.setArmies(l_sourceArmies);
    l_targetCountry.setArmies(l_targetArmies);

    Airlift l_airliftOrder = new Airlift(this.d_player1, l_sourceCountry.getName(), l_targetCountry.getName(), l_numArmies);
    l_airliftOrder.execute();

    assertEquals(l_sourceArmies - l_numArmies, l_sourceCountry.getArmies());
    assertEquals(l_targetArmies + l_numArmies, l_targetCountry.getArmies());
}
```

- **Player.java Class**

  The previous Player class in build 1 was responsible for handling player-related data and order processing in a monolithic way. The class managed player orders through a simple queue system with limited order types (only deploy orders) and basic country management.

  The original implementation had several limitations and was one of the classes that changed the most compared to its previous build:
  Only supported deploy orders through direct user input, Hardcoded support for only deploy orders, nested conditionals for validation, tight coupling with user input. showing monolithic input handling(before)

```java
    /**
     * Takes input from user in this format "deploy countryID num" and adds a command to playerOrders
     * Decreases the appropriate number of reinforcements from the numReinforcement
     */
    public void issue_order() {
        System.out.println(d_name + ", enter your order (deploy <countryID> <num>):");
        Command l_command = InputOutput.get_user_command();
        if (l_command == null) { System.out.println("Invalid order. Please try again.");
            return;
        }
        if (l_command.d_commandType.equals("deploy")) {
            int l_num = Integer.parseInt(l_command.d_argArr.get(1));
            String l_countryName = l_command.d_argArr.get(0);
            if (l_num <= d_numReinforcement && ownsCountry(l_countryName)) {
                Order l_newOrder = new Order("deploy", l_countryName , l_num, this);
                d_playerOrders.add(l_newOrder);
                d_numReinforcement -= l_num;
                System.out.println("Order added: Deploy " + l_num + " armies to country " + l_countryName);
            } else {
                System.out.println("Not enough reinforcements available or you don't own this country.");
            }
        } else {
            System.out.println("Invalid order. Please try again.");
        }
    }
```

  Delegates validation to `order` subclasses, extensible for new order types, cleaner separation of concerns, polymorphic Order Handling, new `issue_order()` delegating to specialized classes(after):

```java
    /**
     * Takes input from user in this format "deploy countryID num" and adds a command to playerOrders
     * Decreases the appropriate number of reinforcements from the numReinforcement
     */
    public void issue_order(GameEngine p_gameEngine,Parsing l_parsing) {
        ArrayList<String> l_arguments = l_parsing.getArgArr();

        switch (l_parsing.d_commandType) {
            case "deploy" -> {
                String l_countryName = l_arguments.get(0).replace('_', ' ');
                int l_num = Integer.parseInt(l_arguments.get(1));

                Order l_deployOrder = new Deploy(this, l_countryName, l_num);
                if (l_deployOrder.isValid()) {
                    this.d_playerOrders.add(l_deployOrder);
                }
            }
            case "advance" -> {
                String l_countryFrom = l_arguments.get(0);
                String l_countryTo = l_arguments.get(1);
                int l_numArmies = Integer.parseInt(l_arguments.get(2));

                Order l_advanceOrder = new Advance(p_gameEngine, this, l_countryFrom, l_countryTo, l_numArmies);
                if (l_advanceOrder.isValid()) {
                    this.d_playerOrders.add(l_advanceOrder);
                }
            }
            case "bomb" -> {
                String l_countryName = l_arguments.getFirst();

                Order l_bombOrder = new Bomb(this, l_countryName);
                if (l_bombOrder.isValid()) {
                    this.d_playerOrders.add(l_bombOrder);
                }
            }
```

```
157                 case "blockade" -> {
158                     String l_countryName = l_arguments.getFirst();
159 
160                     Order l_blockadeOrder = new Blockade(this, l_countryName);
161                     if (l_blockadeOrder.isValid()) {
162                         this.d_playerOrders.add(l_blockadeOrder);
163                     }
164                 }
165                 case "airlift" -> {
166                     String l_sourceCountryName = l_arguments.get(0);
167                     String l_targetCountryName = l_arguments.get(1);
168                     int l_numArmy = Integer.parseInt(l_arguments.get(2));
169 
170                     Order l_airliftOrder = new Airlift(this, l_sourceCountryName, l_targetCountryName, l_numArmy);
171                     if (l_airliftOrder.isValid()) {
172                         this.d_playerOrders.add(l_airliftOrder);
173                     }
174                 }
175                 case "negotiate" -> {
176                     String l_playerName = l_arguments.getFirst();
177 
178                     Order l_diplomacyOrder = new Diplomacy(p_gameEngine, this, l_playerName);
179                     if (l_diplomacyOrder.isValid()) {
180                         this.d_playerOrders.add(l_diplomacyOrder);
181                     }
182                 }
183             }
184         }
```

Had minimal player attributes just name, reinforcements, and countries(before):

```
13  public class Player {
14 
15      private Queue<Order> d_playerOrders;
16      private int d_numReinforcement;
17      public List<Country> d_ownedCountries;
18 
19      private String d_name;
20 
```

Enhanced Player State,(`d_cards`): Enables card-based gameplay mechanics, (`d_diplomacyPlayers`): Tracks diplomatic relationships, (after):

```
13  public class Player {
14 
15      private Queue<Order> d_playerOrders;
16      private int d_numReinforcement;
17      /**
18       * A list of countries owned by the player.
19       */
20      public List<Country> d_ownedCountries;
21 
22      private String d_name;
23      private List<Card> d_cards;
24      private List<Player> d_diplomacyPlayers;
25 
```

Basic Country Validation, manual iteration through `d_ownedCountries`, string-based comparison, no object return capability, (before):

```java
    /**
     * ownsCountry method
     * @param p_countryName String containing the country's name being analyzed
     * @return Boolean type validating if the country is owned by the player
     */
    public boolean ownsCountry(String p_countryName) {
        for (Country l_country : d_ownedCountries) {
            if (l_country.getName().equals(p_countryName)) {
                return true;
            }
        }
        return false;
    }
```

Enhanced Country Operations

- `getCountryByName()` returning full `Country` objects
- `removeCountry()` modifying the collection
- `addCountryToOwnedCountries()` for symmetric management, All methods using consistent iteration style,(after):

```java
     * getCountryByName method
     * @param p_countryName String containing the country's name being searched
     * @return Country object owned by the player
     */
    public Country getCountryByName(String p_countryName){
        for (Country l_country : d_ownedCountries){
            if (l_country.getName().equals(p_countryName)){
                return l_country;
            }
        }
        return null;
    }

```

```java
    public void removeCountry(String p_countryName) {
        for (Country l_country : d_ownedCountries) {
            if (l_country.getName().equals(p_countryName)) {
                d_ownedCountries.remove(l_country);
            }
        }
    }

    public void addCountryToOwnedCountries(Country p_country) {
        d_ownedCountries.add(p_country);
    }

}
```

- **Relevant Test Cases:**

```java
/**
 * Test case for removing a country from the map.
 */
@Test
void testRemoveCountry() {
    Preload l_preload = new Preload(this.d_gameEngine, this.mapReader);
    boolean l_isLoaded = l_preload.loadMap(p_filename:"Witcher_Map");

    Postload l_postload = new Postload(this.mapReader);
    l_postload.removeCountry(p_name:"Novigrad");

    assertNull(mapReader.getCountriesMap().get(key:"Novigrad"), "Novigrad should be removed");
}
```

```java
/**
 * Test a player does not own the target country
 */
@Test
public void playerNotOwnCountry() {
    System.out.println(x:"\nTEST : Player deploys armies where it is not their country");

    Country l_countryToDeploy = this.d_player2.getOwnedCountries().getFirst();
    Deploy l_deployOrder = new Deploy(this.d_player1, l_countryToDeploy.getName(), p_numArmy:5);
    assertFalse(l_deployOrder.isValid());
}
```

- **Command.java Class**
  The previous Command class was structured to handle both the parsing of user input and storage of command data within a single class. This design caused significant maintainability issues and made the system resistant to modifications when new command formats needed to be supported.

Renamed Class

```java
package com.gameplay;

import java.util.*;

/**
 * {@code Command} class manages the parsing of user commands.
 */
public class Command {
    Integer d_numArgs;
    String d_commandType;
    HashMap<String, List<String>> d_argsLabeled;
    ArrayList<String> d_argArr;
    String d_fullCommand;

```

Changed from Command.java to Parsing.java to better reflect its single responsibility of input transformation

```java
package com.gameplay;

import java.util.*;

/**
 * {@code Command} class manages the parsing of user commands.
 */
public class Parsing {
    Integer d_numArgs;
    public String d_commandType;
    public HashMap<String, List<String>> d_argsLabeled;
    public ArrayList<String> d_argArr;
    String d_fullCommand;

```

- **Relevant Test Cases:**

```java
/**
 * Parse edit continent arguments.
 */
@Test
public void parseEditContinentArguments() {
    System.out.println(x:"\nTEST : Parse arguments from 'editcontinent' command");

    Parsing l_parsing = new Parsing(p_input:"editcontinent -add continentID continentValue -remove continentID");
    System.out.println("\nRunning: " + l_parsing.getFullCommand());

    System.out.println("\nArguments parsed from '-add' flag: " + l_parsing.getArgsLabeled().get(key:"-add"));
    assertEquals(new ArrayList<>(Arrays.asList(...a:"continentID", "continentValue")), l_parsing.getArgsLabeled().get(key:"-add"));

    System.out.println("Arguments parsed from '-remove' flag: " + l_parsing.getArgsLabeled().get(key:"-remove"));
    assertEquals(new ArrayList<>(Collections.singletonList(o:"continentID")), l_parsing.getArgsLabeled().get(key:"-remove"));
}
```

```java
/**
 * Parse edit country arguments.
 */
@Test
public void parseEditCountryArguments() {
    System.out.println(x:"\nTEST : Parse arguments from 'editcountry' command");

    Parsing l_parsing = new Parsing(p_input:"editcountry -add countryID continentID -remove countryID");
    System.out.println("\nRunning: " + l_parsing.getFullCommand());

    System.out.println("-> Arguments parsed from '-add' flag: " + l_parsing.getArgsLabeled().get(key:"-add"));
    assertEquals(new ArrayList<>(Arrays.asList(...a:"countryID", "continentID")), l_parsing.getArgsLabeled().get(key:"-add"));

    System.out.println("-> Arguments parsed from '-remove' flag: " + l_parsing.getArgsLabeled().get(key:"-remove"));
    assertEquals(new ArrayList<>(Collections.singletonList(o:"countryID")), l_parsing.getArgsLabeled().get(key:"-remove"));
}
```

```java
/**
 * Parse edit neighbor arguments.
 */
@Test
public void parseEditNeighborArguments() {
    System.out.println(x:"\nTEST : Parse arguments from 'editneighbor' command");

    Parsing l_parsing = new Parsing(p_input:"editcountry -add countryID neighborCountryID -remove countryID neighborCountryID");
    System.out.println("\nRunning: " + l_parsing.getFullCommand());

    System.out.println("-> Arguments parsed from '-add' flag: " + l_parsing.getArgsLabeled().get(key:"-add"));
    assertEquals(new ArrayList<>(Arrays.asList(...a:"countryID", "neighborCountryID")), l_parsing.getArgsLabeled().get(key:"-add"));

    System.out.println("-> Arguments parsed from '-remove' flag: " + l_parsing.getArgsLabeled().get(key:"-remove"));
    assertEquals(new ArrayList<>(Arrays.asList(...a:"countryID", "neighborCountryID")), l_parsing.getArgsLabeled().get(key:"-remove"));
}

/**
 * Parse savemap arguments.
 */
@Test
public void parseSavemapArguments() {
    System.out.println(x:"\nTEST : Parse argument from 'savemap' command");

    Parsing l_parsing = new Parsing(p_input:"savemap testFile");
    System.out.println("\nRunning: " + l_parsing.getFullCommand());

    System.out.println("-> Arguments parsed from 'savemap' command: " + l_parsing.getArgArr().getFirst());
    assertEquals("testFile", l_parsing.getArgArr().getFirst());
}

/**
 * Parse editmap arguments.
 */
@Test
public void parseEditmapArguments() {
    System.out.println(x:"\nTEST : Parse argument from 'editmap' command");

    Parsing l_parsing = new Parsing(p_input:"editmap testFile");
    System.out.println("\nRunning: " + l_parsing.getFullCommand());

    System.out.println("-> Arguments parsed from 'editmap' command: " + l_parsing.getArgArr().getFirst());
    assertEquals("testFile", l_parsing.getArgArr().getFirst());
}
```

```java
/**
 * Parse loadmap arguments.
 */
@Test
public void parseLoadmapArguments() {
    System.out.println(x:"\nTEST : Parse argument from 'loadmap' command");

    Parsing l_parsing = new Parsing(p_input:"loadmap testFile");
    System.out.println("\nRunning: " + l_parsing.getFullCommand());

    System.out.println("-> Arguments parsed from 'loadmap' command: " + l_parsing.getArgArr().getFirst());
    assertEquals("testFile", l_parsing.getArgArr().getFirst());
}
```

```java
/**
 * Parse gameplayer arguments.
 */
@Test
public void parseGameplayerArguments() {
    System.out.println(x:"\nTEST : Parse arguments from 'gameplayer' command");

    // Add and remove players at the same time
    Parsing l_parsing = new Parsing(p_input:"gameplayer -add player1 -remove player2");
    System.out.println("\nRunning: " + l_parsing.getFullCommand());

    System.out.println("-> Arguments parsed from '-add' flag: " + l_parsing.getArgsLabeled().get(key:"-add"));
    assertEquals(new ArrayList<>(Collections.singletonList(o:"player1")), l_parsing.getArgsLabeled().get(key:"-add"));

    System.out.println("-> Arguments parsed from '-remove' flag: " + l_parsing.getArgsLabeled().get(key:"-remove"));
    assertEquals(new ArrayList<>(Collections.singletonList(o:"player2")), l_parsing.getArgsLabeled().get(key:"-remove"));

    // Add players
    l_parsing = new Parsing(p_input:"gameplayer -add player1 player2");
    System.out.println("\nRunning: " + l_parsing.getFullCommand());

    System.out.println("-> Arguments parsed from '-add' flag: " + l_parsing.getArgsLabeled().get(key:"-add"));
    assertEquals(new ArrayList<>(Arrays.asList(...a:"player1", "player2")), l_parsing.getArgsLabeled().get(key:"-add"));

    // Add players
    l_parsing = new Parsing(p_input:"gameplayer -remove player3 player4");
    System.out.println("\nRunning: " + l_parsing.getFullCommand());

    System.out.println("-> Arguments parsed from '-remove' flag: " + l_parsing.getArgsLabeled().get(key:"-remove"));
    assertEquals(new ArrayList<>(Arrays.asList(...a:"player3", "player4")), l_parsing.getArgsLabeled().get(key:"-remove"));
}

/**
 * Parsedeploy arguments.
 */
@Test
public void parsedeployArguments() {
    System.out.println(x:"\nTEST : Parse argument from 'deploy' command");

    Parsing l_parsing = new Parsing(p_input:"deploy countryID 4");
    System.out.println("\nRunning: " + l_parsing.getFullCommand());

    System.out.println("-> Arguments parsed from 'deploy' command: " + l_parsing.getArgArr());
    assertEquals(new ArrayList<>(Arrays.asList(...a:"countryID", "4")), l_parsing.getArgArr());
}
```

- **GameEngine.java Class**

  The previous GameEngine class created for build 1 was in charge of handling the different states of the game all at once inside the very same class.

```java
public class GameEngine {
    // Game starter
    public void startup() {
        System.out.println("----------------------------------------------------------------------------");
        System.out.println("Game setup started.");
        System.out.println("----------------------------------------------------------------------------");
        System.out.println("Add players using 'gameplayer -add <playername>'.");
        System.out.println("Load map using 'loadmap <MapName>'.");
        System.out.println("Display map using 'showmap'.");
        System.out.println("Assign countries and start game with 'assigncountries'.");
        System.out.println("----------------------------------------------------------------------------");
        while (true) {
            Command l_command = null;
            while (l_command == null) {
                l_command = InputOutput.get_user_command();
            }
            if (l_command.d_commandType.equals("gameplayer")) {
                if (l_command.d_argsLabeled.containsKey("-add")) {
                    String l_playername = l_command.d_argsLabeled.get("-add").getFirst();
                    d_playersList.add(new Player(l_playername));
                    System.out.println("Player added: " + l_playername);
                } else if (l_command.d_argsLabeled.containsKey("-remove")) {
                    String l_playerName = l_command.d_argsLabeled.get("-remove").getFirst();
                    d_playersList.removeIf(p -> p.getName().equals(l_playerName));
                    System.out.println("Player removed: " + l_playerName);
                }
```

```java
// Issuing Orders Phase
boolean l_ordersRemaining = true;
while (l_ordersRemaining) {
    l_ordersRemaining = false;
    for (Player l_player : d_playersList) {
        if (l_player.getReinforcements() > 0) {
            l_player.issue_order();
            l_ordersRemaining = true;
        }
    }
}
```

```java
// Order Execution Phase
boolean l_executingOrders = true;
while (l_executingOrders) {
    l_executingOrders = false;
    for (Player l_player : d_playersList) {
        Order l_pendingOrder = l_player.next_order();
        if (l_pendingOrder != null) {
            l_pendingOrder.execute();
            l_executingOrders = true;
        }
    }
}
```

This previous form of the GameEngine class brought a lot of conditions to check every state of the game, making the code larger, less readable and harder to

maintain. The refactored version of this class allowed the implementation of different other classes containing the behavior of every state to be called separately.

```
18    public class GameEngine {
60        public void gameLoop() {
61            while (true) {
62                Parsing l_parsing = null;
63                while (l_parsing == null) {
64                    l_parsing = InputOutput.get_user_command();
65                }
66                if (l_parsing.d_commandType.equals("gameplayer")) {
67                    l_phase.addGamePlayer(l_parsing);
68                } else if (l_parsing.d_commandType.equals("loadmap")) {
69                    l_phase.loadMap(l_parsing);
70                } else if (l_parsing.d_commandType.equals("showmap")) {
71                    l_phase.displayMap();
72                } else if (l_parsing.d_commandType.equals("assigncountries")) {
73                    l_phase.assignCountries();
74                    l_phase = new IssueOrder(this);
75                    l_phase.assignReinforcements();
76                } else if (checkIssuable(l_parsing)) {
77                    if (l_phase.createOrder(l_parsing)) {
78                        l_phase = new ExecuteOrder(this);
79                        while (true) {
80                            if (l_phase.executeOrder()) {
81                                l_phase = new IssueOrder(this);
82                                break;
83                            }
84                        }
85                    }
```

Conditions are filled and the different states are treated as objects depending on the current situation of the game, each with their own functionalities and methods to be accessed.

```
6   public interface Phase {
7       default void addGamePlayer(Parsing l_parsing) {
8           System.out.println("DEFAULTBODY");
9       }
10
11      default void loadMap(Parsing l_parsing) {
12          System.out.println("DEFAULTBODY");
13
14      }
15
16      default void displayMap() {
17          System.out.println("DEFAULTBODY");
18      }
19
20      default void assignCountries() {
21          System.out.println("DEFAULTBODY");
22      }
23
24      default void assignReinforcements() {
25          System.out.println("DEFAULTBODY");
26      }
27
```

The concept of the game state becomes an interface being implemented by every game state class as follows:

```java
public class IssueOrder implements Phase {

    GameEngine engine;
    public ArrayList<String> possibleOrders = new ArrayList<>(List.of("deploy","advance"));


    public IssueOrder(GameEngine engine) {
        if (engine.d_playersList.isEmpty()) {
            System.out.println("No players available. Exiting game loop.");
            engine.l_phase = new Startup(engine);
            return;
        }
        this.engine = engine;
    }
```

Even though the refactored version still contains a considerable amount of conditionals, it drastically reduces the amount from the previous build making the class less complex. Alongside this benefit, this form of refactoring ensures a better implementation of encapsulation and a more dynamic behavior letting the game change phases without altering the functionality of the main driver being the GameEngine class.

- **Relevant Test Cases:**

```java
 * Calculate reinforcement armies.
 */
@Test
public void calculateReinforcementArmies() {
    System.out.println(x:"\nTEST : Calculate the number of reinforcement armies\n");
    System.out.println(this.d_gameEngine.getPlayersList());

    // Change game phase to Issue Order
    this.d_gamePhase = new IssueOrder(this.d_gameEngine);
    this.d_gamePhase.assignReinforcements();

    // Calculating the number of reinforcement armies for each player
    HashMap<String, Integer> players = new HashMap<>();
    for (Player l_player: this.d_gameEngine.getPlayersList()) {
        int l_armies = 5;
        HashSet<Country> l_processedCountries = new HashSet<>();
        for (Country l_country: l_player.d_ownedCountries) {
            if (l_processedCountries.contains(l_country)) {
                continue;
            }
            Continent l_checkingContinent = l_country.getContinent();
            boolean l_givebonus = true;
            for (Country otherCountry : l_checkingContinent.getCountries()) {
                l_processedCountries.add(otherCountry);
                if (otherCountry.getOwner() != (l_player)) {
                    l_givebonus = false;
                    break;
                }
            }
            if (l_givebonus) {
                players.put(l_player.getName(), l_armies + l_checkingContinent.getBonus());
            }
        }
        if (!players.containsKey(l_player.getName())) {
            players.put(l_player.getName(), l_armies);
        }
    }

    System.out.println("\n-> Total number of reinforcement armies for " + this.d_player1.getName() + " : " + this.d_player1.getReinforcements());
    System.out.println("-> Total number of reinforcement armies for " + this.d_player2.getName() + " : " + this.d_player2.getReinforcements());

    assertEquals(players.get(key:"TestPlayer1"), this.d_player1.getReinforcements());
    assertEquals(players.get(key:"TestPlayer2"), this.d_player2.getReinforcements());
}
```

```java
@Test
public void verifyGamePhase() {
    System.out.println(x:"\nTEST : Verifying game phases\n");

    System.out.println(x:"Set game phase to 'Startup'");
    this.d_gamePhase = new Menu(this.d_gameEngine);

    System.out.println("Current game phase: " + this.d_gamePhase.currentPhase());
    assertEquals("Menu", this.d_gamePhase.currentPhase());

    System.out.println(x:"Set game phase to 'IssueOrder'");
    this.d_gamePhase = new IssueOrder(this.d_gameEngine);

    System.out.println("Current game phase: " + this.d_gamePhase.currentPhase());
    assertEquals("IssueOrder", this.d_gamePhase.currentPhase());

    System.out.println(x:"Set game phase to 'ExecuteOrder'");
    this.d_gamePhase = new ExecuteOrder(this.d_gameEngine);

    System.out.println("Current game phase: " + this.d_gamePhase.currentPhase());
    assertEquals("ExecuteOrder", this.d_gamePhase.currentPhase());
}
```

- **MainMenu.java Class**
  On the previous build, our main menu stage was handling the different actions of the game in order to know what to show the user on the command terminal, like the structured messages, and commands and opened processes like the following:

```java
public class MainMenu {
    public void run_main_menu() {
        do {
            System.out.println("\n**********************");
            System.out.println("* Welcome to Warzone *");
            System.out.println("**********************");

            Scanner l_scanner = new Scanner(System.in);

            System.out.println("\ncom.Main Menu:");
            System.out.println("1. Map Editor");
            System.out.println("2. Play Warzone Game");
            System.out.println("3. Exit\n");
            System.out.println("Enter your choice: ");

            // Check whether user input is an integer or not
            if (l_scanner.hasNextInt()) {
                l_input = l_scanner.nextInt();

                // Perform actions based on user input
                switch (l_input) {
                    // Runs map editor mode
                    case 1:
                        InputOutput l_inputOutput = new InputOutput();
                        l_inputOutput.run_map_editor();
                        break;
                    // Starts the Warzone Game
                    case 2:
                        GameEngine gameEngine = new GameEngine();
                        gameEngine.startup();
                        break;
                    // Finishes the program, prompting the user an exit message
                    case 3:
                        System.out.println("\nSuccessfully exit the game.");
                        break;
                }
            } else {
                System.out.println("\nInvalid input. Please try again.\n");
            }
        } while(l_input != 3);  // Keep looping if user doesn't choose exit
    }
}
```

This structure gives the class the authority to create the main GameEngine object used to run the whole game logic, it's a big responsibility for a class whose sole purpose is to visually represent the interactions between the user and the program.

The refactored version becomes cleaner with less lines of code and illustrates the Single Responsibility principle as follows:

```java
package com.States;

import com.gameplay.GameEngine;
import com.maps.MapReader;

public class Menu implements Phase {
    GameEngine engine;
    public Menu(GameEngine engine){
        this.engine = engine;
        System.out.println("""
                --------------------------------------------------------------------
                                                MAIN MENU
                --------------------------------------------------------------------
                Commands:

                startgame
                editor
                quit
                --------------------------------------------------------------------
                """
        );
    }
    public void editor(){
        engine.d_phase = new Preload(engine,new MapReader());
    }
    public void startGame(){
        engine.d_phase = new Startup(engine);
    }
    public String currentPhase() {
        return "Menu";
    }
}
```

The new Menu class, with its name refactored as well, creates different game phases depending on the option selected by the user from the menu but each phase has its own logic managing the stages of the game and its triggers, instead of creating the whole game logic within the menu.

- **Relevant Test Cases:**

```java
@Test
public void verifyGamePhase() {
    System.out.println(x:"\nTEST : Verifying game phases\n");

    System.out.println(x:"Set game phase to 'Startup'");
    this.d_gamePhase = new Menu(this.d_gameEngine);

    System.out.println("Current game phase: " + this.d_gamePhase.currentPhase());
    assertEquals("Menu", this.d_gamePhase.currentPhase());

    System.out.println(x:"Set game phase to 'IssueOrder'");
    this.d_gamePhase = new IssueOrder(this.d_gameEngine);

    System.out.println("Current game phase: " + this.d_gamePhase.currentPhase());
    assertEquals("IssueOrder", this.d_gamePhase.currentPhase());

    System.out.println(x:"Set game phase to 'ExecuteOrder'");
    this.d_gamePhase = new ExecuteOrder(this.d_gameEngine);

    System.out.println("Current game phase: " + this.d_gamePhase.currentPhase());
    assertEquals("ExecuteOrder", this.d_gamePhase.currentPhase());
}
```