

Group - ADYST: REFACTORING DOCUMENT

COURSE: SOEN 6441

TEAM MEMBERS:

- Akhilesh Kanbarkar (40301665)
- Disha Padsala (40324578)
- Yash Koladiya (40324106)
- Shrey Hingu (40305516)
- Taksh Rana (40303723)

Potential Refactoring Targets:

Following the completion of Build 1, several enhancements and code optimizations were essential to align with the evolving requirements of Build 2. The following refactoring actions were implemented to improve code maintainability, scalability, and readability:

- **Centralized Constant Management**
Replaced hard-coded string literals with named constants organized in a dedicated class to boost modularity and reuse.
- **JavaDoc Enhancement**
Documented private fields and methods using JavaDoc annotations.
- **Improved Naming Conventions**
Standardized variable and method naming conventions for better readability and easier collaboration.
- **Expanded Test Coverage**
Added new unit tests to validate the functionality of newly introduced and existing features.
- **User-Friendly Error Reporting**
Enhanced error-handling by displaying clear, actionable messages when users perform invalid operations.
- **Logger Integration for Event Tracking**
Developed a centralized Logger component to track and record key events throughout the game lifecycle.
- **Observer Pattern with Logger Updates**
Leveraged the Observer design pattern to notify all relevant views of log updates, ensuring synchronization across components.

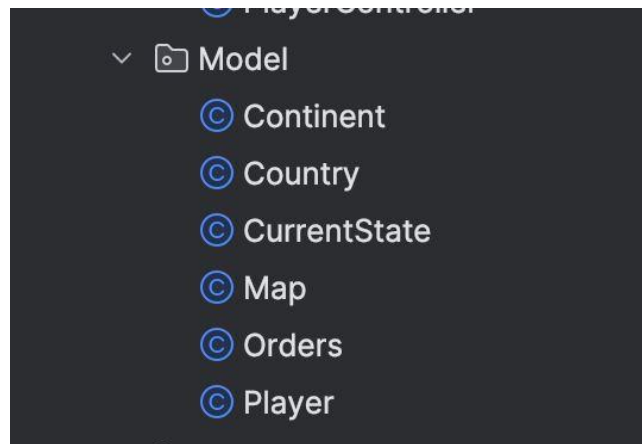
- **Application of State Design Pattern**
Introduced the State pattern to streamline transitions across game phases: Startup, Order Issuing, and Order Execution.
- **Command Pattern for Order Handling**
Employed the Command design pattern to manage and process different order commands in a more structured manner.
- **Code Cleanup and Import Optimization**
Removed unused and redundant imports to improve code cleanliness and compile-time efficiency.
- **Command Syntax Validation**
Implemented rigorous input validation to prevent syntactic errors in user-entered commands.
- **Mandatory Player Count Enforcement**
Enforced a rule to ensure a minimum of two players are present before starting a game.
- **Refactoring of Existing Test Cases**
Updated the existing test cases to match the new structure and logic introduced in Build 2.
- **Validation of Invalid Commands**
Built validation logic for all command operations, ensuring the system handles invalid inputs gracefully.
- **Comprehensive Test Suite Organization**
Developed a structured test suite that includes tests across all major modules and folders for streamlined quality assurance.

Actual Refactoring Targets:

1) Implementation of State Pattern for phases.

Before Refactoring (Build 1):

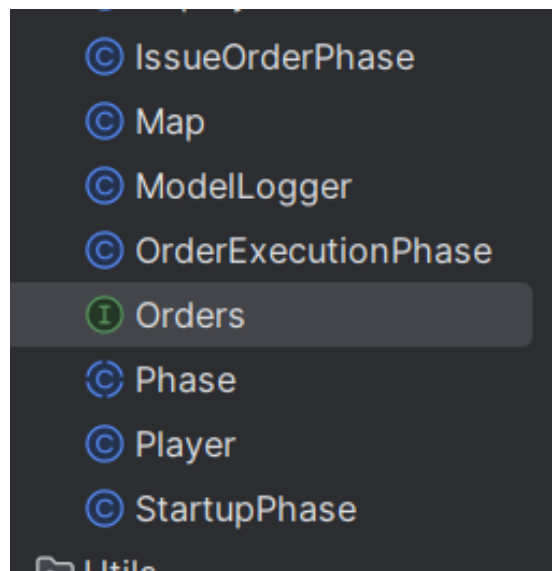
In Build 1, the game phases were managed directly within MainGameEngine without using the State Pattern. Command parsing and conditional checks (if-else) controlled the transitions between the Startup Phase, Issue Order Phase, and Order Execution Phase. This resulted in tightly coupled logic, making MainGameEngine overly complex. Any modification required updating multiple conditions, leading to reduced flexibility and maintainability.



Model classes in Build 1

After Refactoring (Build 2): Implementing the State Pattern

In Build 2, we implemented the State Pattern by introducing an abstract Phase class and three concrete phase classes (StartupPhase, IssueOrderPhase, OrderExecutionPhase). The MainGameEngine maintains a reference to the current phase and delegates phase-specific operations to it. Transitions between phases are now handled via methods like setIssueOrderPhase() and setOrderExecutionPhase(), making the game flow more structured. Each phase encapsulates its own behavior, improving code modularity, reducing complexity, and making future modifications easier and more scalable.



Added phase classes for State pattern

```

21 public class OrderExecutionPhaseTest {
22     public void setUp() {
23         d_map.getCountryByName( p_countryName: "Canada").setD_armies(0);
24         l_player1Countries.add(d_map.getCountryByName( p_countryName: "USA"));
25         l_player1Countries.add(d_map.getCountryByName( p_countryName: "Canada"));
26         l_player1Countries.add(d_map.getCountryByName( p_countryName: "France"));
27         l_player1Countries.add(d_map.getCountryByName( p_countryName: "UK"));
28         l_player1Countries.add(d_map.getCountryByName( p_countryName: "Germany"));
29         l_player1Countries.add(d_map.getCountryByName( p_countryName: "Argentina"));
30         l_player1Countries.add(d_map.getCountryByName( p_countryName: "Chile"));
31         l_player2Countries.add(d_map.getCountryByName( p_countryName: "Brazil"));
32         d_player1.setD_currentCountries(l_player1Countries);
33         d_player2.setD_currentCountries(l_player2Countries);
34
35         d_orderExecutionPhase = new OrderExecutionPhase(d_currentState, d_mainGameEngine);
36     }
37
38     /**
39      * Check if the game ends after executing an advance order.
40      * <p>
41      * This test verifies whether the end of game condition is triggered after a successful conquest
42      * and the acquisition of a card by Player 1.
43      * </p>
44      */
45     @Test
46     public void checkEndOfGame() {
47         assertFalse(d_orderExecutionPhase.checkEndOfGame(d_currentState));
48         Orders l_advanceOrder = new Advance( p_sourceCountry: "USA", p_targetCountry: "Brazil", p_noOfArmiesToPlace: 9, d_player1);
49         assertEquals( expected: 0, d_player1.getD_cardOwnedByPlayer().size());
50         l_advanceOrder.execute(d_currentState);
51         assertEquals( expected: 1, d_player1.getD_cardOwnedByPlayer().size());
52         assertTrue(d_orderExecutionPhase.checkEndOfGame(d_currentState));
53     }
54 }

```

Test cases implementation for OrderExecutionPhase

2) Implementation of Command Pattern for executing orders

Before Refactoring (Build 1):

In Build 1, order processing did not follow the Command Pattern. A single function, execute, was responsible for handling all order logic, which is Deploy. This approach resulted in tightly coupled and monolithic code, making it difficult to modify or extend order behaviors without altering a large portion of the codebase.

```

78 /**
79  * Executes the order for the given player.
80  * <p>
81  * If the order command is "deploy", this method iterates through the player's current countries,
82  * and when it finds the country whose name matches the target name, it adds the specified number of armies.
83  * </p>
84  *
85  * @param p_eachPlayer the player executing the order
86  */
87 public void execute(Player p_eachPlayer) {
88     if(d_order.equals("deploy")){
89         for(Country l_eachCountry : p_eachPlayer.getD_currentCountries()){
90             if(l_eachCountry.getD_countryName().equals(this.d_targetName)){
91                 l_eachCountry.setD_armies(l_eachCountry.getD_armies() + this.d_noOfArmiesToMove);
92                 break;
93             }
94         }
95     }
96 }

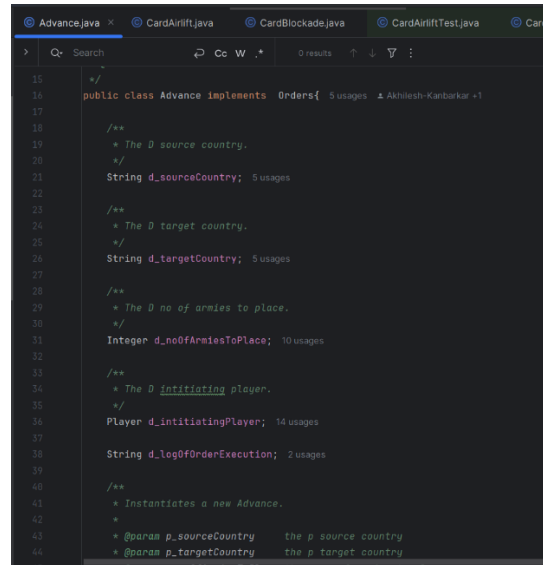
```

Execute method in Build 1

After Refactoring (Build 2): Implementing the Command Pattern

In Build 2, we implemented the Command Pattern for order processing. Each order type—Deploy, Advance, Bomb, Blockade, and Airlift—was encapsulated in its own command class, inheriting from a common interface. The execution logic is now handled by

individual command objects, improving modularity, separation of concerns, and maintainability. This change allows dynamic command execution and simplifies future enhancements without affecting existing implementations.

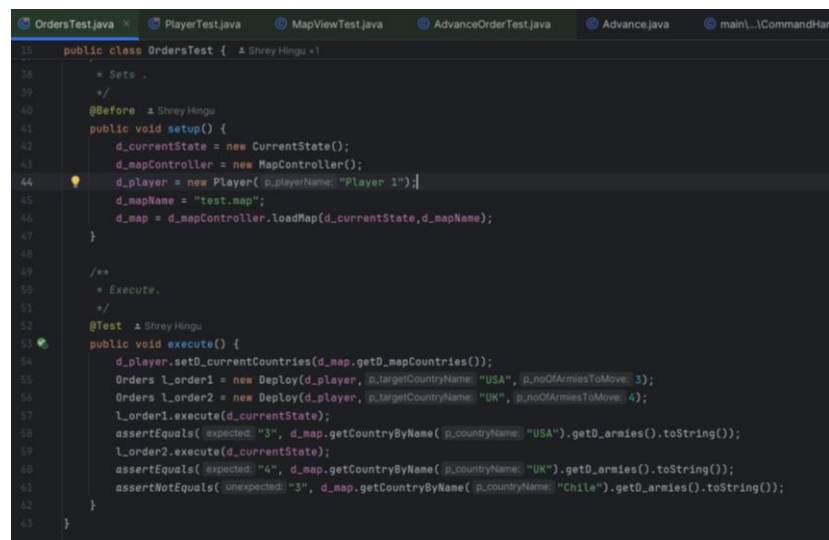


```

15  */
16  public class Advance implements Orders {
17      /**
18       * The d source country.
19       */
20      String d_sourceCountry;
21      /**
22       * The d target country.
23       */
24      String d_targetCountry;
25      /**
26       * The d no of armies to place.
27       */
28      Integer d_noOfArmiesToPlace;
29      /**
30       * The d initiating player.
31       */
32      Player d_initiatingPlayer;
33      /**
34       * The log of order execution.
35       */
36      String d_logOfOrderExecution;
37      /**
38       * Instantiates a new Advance.
39       *
40       * @param p_sourceCountry the p source country
41       * @param p_targetCountry the p target country

```

Example of Advance command class



```

35  public class OrdersTest {
36      /**
37       * Sets up the test environment.
38       */
39      @Before
40      public void setup() {
41          d_currentState = new CurrentState();
42          d_mapController = new MapController();
43          d_player = new Player(p_playerName, "Player 1");
44          d_mapName = "test.map";
45          d_map = d_mapController.loadMap(d_currentState, d_mapName);
46      }
47      /**
48       * Executes the test cases.
49       */
50      @Test
51      public void execute() {
52          d_player.setD_currentCountries(d_map.get_mapCountries());
53          Orders l_order1 = new Deploy(d_player, p_targetCountryName: "USA", p_noOfArmiesToMove: 3);
54          Orders l_order2 = new Deploy(d_player, p_targetCountryName: "UK", p_noOfArmiesToMove: 4);
55          l_order1.execute(d_currentState);
56          assertEquals("expected: 3", d_map.getCountryByName(p_countryName: "USA").getD_armies().toString());
57          l_order2.execute(d_currentState);
58          assertEquals("expected: 4", d_map.getCountryByName(p_countryName: "UK").getD_armies().toString());
59          assertEquals("unexpected: 3", d_map.getCountryByName(p_countryName: "Chile").getD_armies().toString());
60      }
61  }

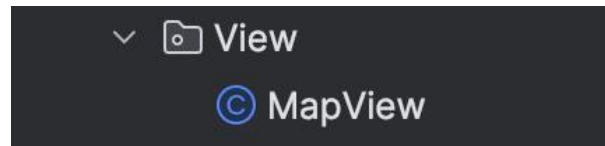
```

Test cases for Deploy Order

3) Implementation of Observer Pattern for Game log file creation

Before Refactoring (Build 1):

In Build 1, there was no centralized logging system for tracking game events. The game simply displayed outputs to the user through the console, making it difficult to trace past events and debug issues effectively. No record of executed commands or their effects was maintained.



Only Mapview file in Build 1

After Refactoring (Build 2): Implementing the Observer Pattern

In Build 2, we introduced the Observer Pattern to implement a logging system for tracking game events. A class was created to observe game actions, logging every executed command and its effects into a structured log file. This approach provides better traceability, debugging support, and allows players to review the history of game actions. By decoupling logging from core game logic, we improved maintainability and scalability while ensuring that logs persist beyond the session.

```
java  GameLogger.java  CardAirLift.java  CardBlockade.java  CardAirLiftTest.java  CardB
1  package View;
2
3  import Model.ModelLogger;
4  import java.io.BufferedWriter;
5  import java.io.File;
6  import java.io.FileWriter;
7  import java.io.IOException;
8  import java.util.Observable;
9  import java.util.Observer;
10
11  /**
12   * The GameLogger class listens for changes to the message in ModelLogger
13   * and writes the updated message to a log file.
14   * @author Akhilesh Kanbarkar
15   */
16  public class GameLogger implements Observer { 3 usages  ± Akhilesh-Kanbarkar +2
17
18      /**
19       * The default file path for storing game log output.
20       * Used by the logging system to write execution details, game events, and errors.
21       */
22      private static final String LOG_FILE_PATH = "src/main/resources/GameLogs.txt"; 1 usage
23
24      /**
25       * The D model logger.
26       */
27      ModelLogger d_modelLogger; 2 usages
28
29      /**
30       * Instantiates a new Game logger.
31       */
32      public GameLogger() { 1 usage  ± Akhilesh-Kanbarkar
```

Gamelogger class which extends Observer

```
GameSession.java  GameLog.txt  MapViewTest.java  AdvanceOrderTest.java

Game Session Started

=====Startup Phase of the Game=====

Command Entered: loadmap

Command Entered: gameplayer

Command Entered: assigncountries

=====Issue Order Phase=====

Order Issued: Player : shrey has entered command : deploy China 3

Command Entered: deploy

Log: Order is added to queue for execution.

Order Issued: Player : xyz has entered command : deploy India 3
```

Game logger file

4) Creation of Constants class to replace hardcoded messages

Before Refactoring (Build 1):

In Build 1, all messages displayed to the user were hardcoded directly within the game logic. This led to code duplication and inconsistencies when updating messages across different parts of the application.

```
390 @ private Map addRemoveCountry(Map p_mapToUpdate, String p_operation, String p_arguments) { 1 usage 1 Shrey Hingu
391     String[] splitArgs = p_arguments.split(regex: " ");
392
393     if (p_operation.equals("add")) {
394         if (splitArgs.length == 2) {
395             String countryName = splitArgs[0];
396             String continentName = splitArgs[1];
397
398             p_mapToUpdate.addCountry(countryName, continentName);
399         } else {
400             System.out.println("Error: Invalid format. Use 'add <country> <continent>'.");
401         }
402     } else if (p_operation.equals("remove")) {
403         if (splitArgs.length == 1) {
404             String countryName = splitArgs[0];
405
406             p_mapToUpdate.removeCountry(countryName);
407             System.out.println("Country " + countryName + " removed successfully!");
408         } else {
409             System.out.println("Error: Invalid format. Use 'remove <country>'.");
410         }
411     } else {
412         System.out.println("Error: Invalid operation. Use 'add' or 'remove'.");
413     }
414     return p_mapToUpdate;
415 }
```

Example method with hardcoded messages

After Refactoring (Build 2): Implementing a Constants Class

In Build 2, we introduced a Constants class to store all predefined messages. Instead of hardcoding messages throughout the code, we now reference constants from this centralized class. This change improves maintainability, consistency, and allows for easier modifications, such as localization or customization of messages, without modifying multiple parts of the codebase.

```
public class ProjectConstants { 57 usages 1 dishapadsala

    /**
     * Private constructor to prevent instantiation of this utility class.
     */
    public ProjectConstants() { no usages 1 dishapadsala
    }

    /** Error message when the map is not available. */
    public static final String MAP_NOT_AVAILABLE = "Map is not available. Use 'loadmap <filename>' or 'editmap <filename>' first.";

    /** Error message for an invalid command. */
    public static final String INVALID_COMMAND = "Invalid command. Please provide a valid command."; 1usage

    /** Error message for incorrect 'savemap' command usage. */
    public static final String INVALID_SAVEMAP_COMMAND = "Save map command is not correct. Use 'savemap <filename>' command."; 2us

    /** Error message for incorrect 'validatemap' command usage. */
    public static final String INVALID_VALIDATEMAP_COMMAND = "Validate map command is not correct. Use 'validatemap' command."; 1u

    /** Success message when a map is valid. */
    public static final String VALID_MAP = "Map is valid."; 3 usages

    /** Error message when a map is invalid. */
```

Constants class

```
private Map addRemoveCountry(Map p_mapToUpdate, String p_operation, String p_arguments) { 1usage 1 ShreyHingu +1
    String[] splitArgs = p_arguments.split(regex: " ");

    if (p_operation.equals("add")) {
        if (splitArgs.length == 2) {
            String countryName = splitArgs[0];
            String continentName = splitArgs[1];

            p_mapToUpdate.addCountry(countryName, continentName);
        } else {
            System.out.println(ProjectConstants.INVALID_ADD_COUNTRY_COMMAND);
        }
    } else if (p_operation.equals("remove")) {
        if (splitArgs.length == 1) {
            String countryName = splitArgs[0];

            p_mapToUpdate.removeCountry(countryName);
            System.out.println("Country " + countryName + " removed successfully!");
        } else {
            System.out.println(ProjectConstants.INVALID_REMOVE_COUNTRY_COMMAND);
        }
    } else {
        System.out.println(ProjectConstants.INVALID_OPERATION);
    }

    return p_mapToUpdate;
}
```

Method with predefined messages

5) Command Syntax validation

Before Refactoring (Build 1):

In Build 1, commands were not thoroughly validated before execution. This led to potential errors and inconsistencies during gameplay, as invalid or incomplete commands could still be processed.

After Refactoring (Build 2): Implementing ValidateCommand

In Build 2, we introduced a `ValidateCommand` class to ensure that all commands are properly checked before execution. This validation step prevents errors from propagating through the system, improving reliability and user experience. By enforcing command validation, we minimize unexpected behaviors and enhance the robustness of the game logic.

```
//
public boolean checkRequiredKey(String l_arguments, Map<String, String> l_singleOperation) { // usage: 1 Yash
    if(l_singleOperation.containsKey(l_arguments) && l_singleOperation.get(l_arguments) != null && !l_singleOperation.get(l_arguments).isEmpty()){
        return true;
    }
    return false;
}
```

Validation method