



SOEN 6441 - Advanced Programming Practices

Submitted To: Prof. Joey Paquet

Submitted By: Group 22

Team Members	
Student ID	Name
40289562	Inderjeet Singh Chauhan
40292088	Saphal Ghimire
40256283	Apoorva Sharma
40238072	Vibhor Gulati
40279035	Mohammad Zaid Shaikh

Before moving from Build II to III, we noted 15 potential refactoring targets that would improve the performance and quality of the code. We also have considered reusability, implementation, single responsibility principles, simplification of complex functions, code duplication, proper use of object oriented concepts and many more. Also, we realized that we have not used exceptions for error handling, so we kept it as a potential refactoring target. We also saw the need of using interfaces more extensively to define consistent contracts for classes, improving code flexibility and promoting object oriented programming. We needed to implement single responsibility principles and remove overused codes like if else and switch statements. We were taking inputs from multiple places and we needed it to be refactored to be handled from the same place. We have refactored the load map and save map functionalities to adapt the adaptor pattern. Issue order method is refactored to adapt the strategies.

Potential Refactoring Targets

The potential refactoring targets are listed below:

1. Refactor Edit Continent
2. Refactor Edit Country
3. Improve Documentation
4. Extract the file I/O operations into a separate class to separate concerns and adhere to the Single Responsibility Principle in DominationMapIO class.
5. Overuse of if else and switch statements and change certain data types
6. Remove complexity and simplify function
7. Implement logger functionalities
8. Increase use of getter and setters
9. Increase null value check
10. Refactored player constructor to accept the strategy
11. Command Validator - New CPU command
12. Adaptor pattern for the loading different formats of maps
13. Refactored issue order method
14. Add more Exception handling
15. Execution functions can return boolean if successfully executed instead of void

Actual Refactoring Targets

1. Refactored player constructor to accept the strategy
2. Command Validator - New CPU command
3. Adaptor pattern for the loading different formats of maps
4. Refactored issue order method
5. Introduced model for GameController

Major Refactoring Targets:

1. Refactored player constructor to accept the strategy

In our Players class, which handles player assignments for the game, we've made a change to the constructor to incorporate strategies. This update boosts the class's versatility by enabling it to manage various player strategies during the game setup phase.

Previously, the constructor only took a GameEngine instance as input. However, to accommodate CPU players with specific strategies, we've expanded it to include a strategy parameter. This means that when developers create a new Players class instance, they can now specify both the GameEngine instance and the strategy to assign to CPU players.

This improvement offers developers more control over the game setup process, allowing them to dynamically configure player strategies based on their needs. It promotes code reusability and extensibility by separating the strategy assignment logic from other parts of the application, making maintenance and future enhancements easier.

Screenshot of the testcase

```

new *
public class PlayerTest {

    new *
    @Test
    public void testPlayerConstructorWithStrategy() {
        // A mock strategy for testing
        Strategy strategy = Strategy.Aggressive;

        // A Player object with the specified strategy
        Player player = new Player( p_name: "TestPlayer", strategy);

        // Check if the strategy is correctly set
        assertEquals(strategy, player.get_playerStrategyType());
    }
}

```

Screenshot of the code before

```

/**
 * Constructs a player with the given name.
 *
 * @param p_name The name of the player.
 */
GSaphal
public Player(String p_name) { this.setName(p_name); }

```

Screenshot of the code after

```

/**
 * Constructs a player with the given name.
 *
 * @param p_name The name of the player.
 */
GSaphal
public Player(String p_name) { this.setName(p_name); }

/**
 * Constructs a player with the given name.
 *
 * @param p_name The name of the player.
 */
Vibhor7398
public Player(String p_name, Strategy p_strategy) {
    this.setName(p_name);
    this.set_playerStrategyType(p_strategy);
}

```

2. Command validator - New CPU command

The `CommandValidator` class is a vital part of our game app. Its job is to make sure that the commands users type follow the rules and formats we've set. This is crucial for keeping the game running smoothly.

It checks every command a user gives to make sure it's valid and can be done by the game engine. It looks at different parts of the command, like the main command itself, any extra commands, and how many pieces of information are needed and what type they should be.

The `CommandValidator` class makes sure that commands users type in a strategy game match the ones we've decided are okay. The main method it uses is called `validateCommand`. This method takes the command a user typed and either says it's okay by making a `Command` object, or it says it's not okay by throwing an `InvalidCommandException`. This method looks closely at every part of the command to make sure it's one of the commands we've approved, making sure everything's good before letting the game run.

One big improvement we've made is now you can add CPU players with specific strategies. This is done by adding a "-cpu" part under the main "gameplayer" command. With this, players can

not only add human players but also tell the game to make computer players with certain strategies.

Screenshot of the test case

```
}  
  
new *  
@Test  
public void cpuCommandValidator() {  
    CommandValidator l_cmd=new CommandValidator();  
    Command[] l_commands;  
    try {  
        l_commands = l_cmd.validateCommand( p_command: "gameplayer -cpu cpu1 Aggressive");  
    } catch (InvalidCommandException e) {  
        throw new RuntimeException(e);  
    }  
    assertTrue(l_commands[0] instanceof Command);  
}  
}
```

Screenshot of the code after the changes

```
        new String[]{"String"},  
        new ValidCommands( p_command: "gameplayer",  
                            p_subCommand: "-remove",  
                            p_numArgs: 1,  
                            new String[]{"playerName"},  
                            new String[]{"String"}),  
        new ValidCommands( p_command: "gameplayer",  
                            p_subCommand: "-cpu",  
                            p_numArgs: 2,  
                            new String[]{"playerName", "strategy"},  
                            new String[]{"String", "String"}),  
        new ValidCommands( p_command: "assigncountries",  
                            p_subCommand: "",  
                            p_numArgs: 0,  
                            new String[]{""},  
                            new String[]{""}),  
        new ValidCommands( p_command: "deploy",  
                            p_subCommand: "",
```

```

/**
 * Checks if the provided argument is a valid subcommand.
 *
 * @param p_subCommand The subcommand to validate.
 * @return true if the subcommand is valid, false otherwise.
 */
1 usage  1 GSaphal +1
private boolean isValidSubCommand(String p_subCommand) {
    return p_subCommand.equals("-add") || p_subCommand.equals("-remove") || p_subCommand.equals("-cpu");
}

    throw new InvalidCommandException(getValidCommand(l_baseCommand));
}
if (isValidSubCommand(l_cmd[1])) {
    l_validCommand = getValidCommandObject(l_baseCommand, l_cmd[1]);
    if (l_validCommand == null) {
        throw new InvalidCommandException(getValidCommand(l_baseCommand));
    }
    if (!l_cmd[1].equals("-add") && !l_cmd[1].equals("-remove") && !l_cmd[1].equals("-cpu")) {
        throw new InvalidCommandException(getValidCommand(l_baseCommand));
    }
    commandPointer++;
}
}
if (l_cmd.length > commandPointer) {

```

3. Refactored the loadmap and savemap functionalities to adopt the Adapter pattern

We made a strategic decision to refactor our map loading and saving functions using the Adapter pattern. This choice wasn't just about code restructuring; it was about ensuring our game's adaptability and maintainability for the long haul. By implementing this pattern, we've effectively separated the concerns of map handling from the core gameplay logic. Now, regardless of whether we're dealing with Conquest or Domination map formats, our game seamlessly integrates with both, thanks to the adapters we've crafted.

One of the key advantages we've gained is the ability to easily incorporate new map formats in the future. This modularity ensures that our codebase remains clean and comprehensible, adhering to the principle of Single Responsibility. Moreover, by abstracting away common file I/O operations into reusable adapter classes, we've minimized code duplication and promoted a more efficient, DRY coding approach.

But perhaps the most significant benefit of this adaptation lies in the enhanced flexibility it offers. We've future-proofed our game, enabling it to evolve alongside changing requirements without compromising its integrity. This means smoother development cycles, fewer headaches when implementing new features, and ultimately, a better gaming experience for our players.

Screenshot of the code before the changes

```
public class GameController {

    /**
     * Loads a map from a file and validates it.
     * This method attempts to load a map from the specified file, validates it, and checks if the map is valid.
     * If the map is invalid, a message indicating the same is printed.
     *
     * @param p_filename The name of the file containing the map to be loaded.
     */
    public void executeLoadMap(String p_filename){
        try {
            d_Map.loadMap(AppConstants.MapsPath + p_filename);
            d_Map.validateMap();
            boolean l_isValid = d_Map.isMapValid();
            if(!l_isValid){
                System.out.println("Map is invalid!");
            }
        } catch (IOException l_e) {
            System.out.println("Load map failed. Check for map file. " + l_e.getMessage());
        }
    }

    /**
     * Displays the details of the current game map.
     * This method invokes the `showMap` method of the game map object to display the details of the map,
     * including continents, countries, and their respective neighbors.
     */
    public void executeShowMap(){
        d_Map.showMap();
    }
}
```

Screenshot of the code after the changes

```
0      * This method overrides the ConquestMapIO's loadMap method to adapt between the Domination and Conquest formats.
1      *
2      * @param p_gameMap The Maps object to populate with the loaded map data.
3      * @param p_fileName The name of the file from which to load the map.
4      */
5      GSaphal
6      @Override
7      public boolean loadMap(Maps p_gameMap, String p_fileName) {
8          return d_dominationMapIO.loadMap(p_gameMap, p_fileName);
9      }
10
11     /**
12     * Saves the current state of the game map, represented by the provided Maps object, to a file using the
13     * DominationMapIO's save method. This method overrides the ConquestMapIO's saveMap method to adapt between the
14     * Domination and Conquest formats.
15     *
16     * @param p_gameMap The Maps object containing the map data to be saved.
17     * @param p_fileName The name of the file to which the map data will be saved.
18     */
19     GSaphal
20     @Override
21     public boolean saveMap(Maps p_gameMap, String p_fileName){
22         return d_dominationMapIO.saveMap(p_gameMap, p_fileName);
23     }
24 }
```



```

51  /**
52   * Loads a map from the specified file and populates the provided game map.
53   * The file should be in the Domination map format. It reads the continents,
54   * countries, and their borders from the file and initializes the game map accordingly.
55   *
56   * @param p_gameMap the game map to populate
57   * @param p_fileName the name of the file containing the map
58   * @return {@code true} if the map is loaded successfully, {@code false} otherwise
59   */
60
61  GSaphal +1
62  @ public boolean loadMap(Maps p_gameMap, String p_fileName) {
63      try {
64          p_gameMap.resetMap();
65          String l_content = Files.readString(Paths.get(p_fileName));
66          String[] l_lines = l_content.split( regex: "\n");
67          boolean l_readingContinents = false, l_readingCountries = false, l_readingBorders = false;
68          for (String l_line : l_lines) {
69              l_line = l_line.trim();
70              switch (l_line) {
71                  case "[Continents]" -> {
72                      l_readingContinents = true;
73                      l_readingCountries = false;
74                      continue;
75                  }
76              }
77          }
78      }
79  }
80
81  /**
82   * Loads a game map from a file into the provided Maps object.
83   *
84   * @param p_gameMap The Maps object to load the game map into.
85   * @param p_fileName The file name of the map file to load.
86   * @return true if the map is loaded successfully, false otherwise.
87   */
88  1 override GSaphal +1
89  @ public boolean loadMap(Maps p_gameMap, String p_fileName) {...}
90
91  /**
92   * Saves the game map to a file.
93   *
94   * @param p_gameMap The Maps object representing the game map.
95   * @param p_fileName The name of the file to save the map to.
96   * @return true if the map is successfully saved, false otherwise.
97   */
98  1 override GSaphal +1
99  @ public boolean saveMap(Maps p_gameMap, String p_fileName) {
100      try {
101          File l_file = new File(p_fileName);
102          if (!l_file.exists()) {
103              System.out.println("The file doesn't exist, creating a new file.");
104              if (!l_file.createNewFile()) {
105                  // ...
106              }
107          }
108      }
109  }

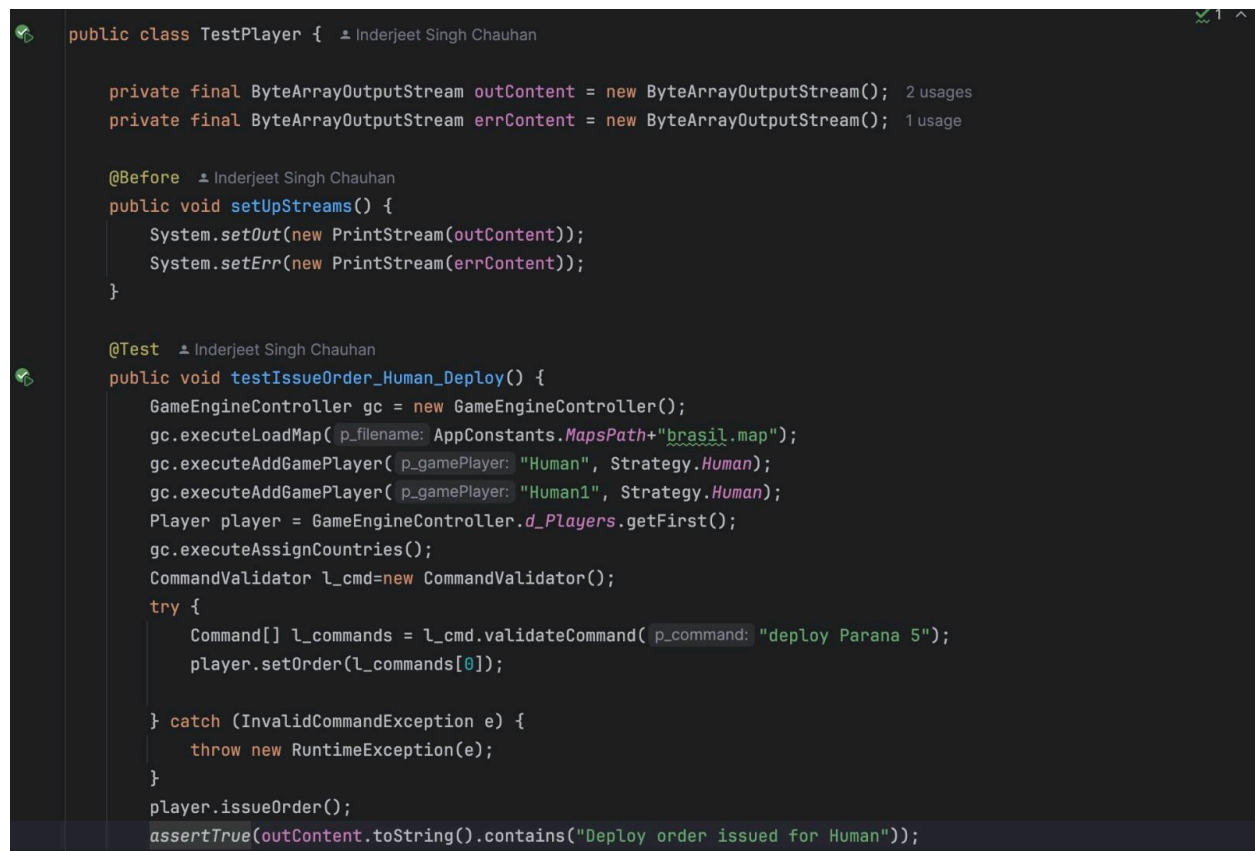
```

4. Refactored Issue order method

The issue order method checks if the player is of type "Human". If so, it allows the player to issue various types of orders such as deploy, advance, airlift, bomb, blockade, negotiate, or end their turn. Each type of order triggers specific actions within the game. For example, a "deploy" order might involve deploying troops to a specific country on the game map.

If the player is not human, the method determines the player's strategy type, which could be aggressive, benevolent, cheater, or random. Depending on the strategy, the method generates an order accordingly. For instance, an aggressive player might generate orders to attack neighboring countries aggressively, while a benevolent player might focus on strengthening alliances or defenses.

Screenshot of the test case

A screenshot of a code editor showing a Java test class named TestPlayer. The class has two private final ByteArrayOutputStream objects, outContent and errContent. It has a @Before method setUpStreams() that sets the system's out and err streams to these objects. It also has a @Test method testIssueOrder_Human_Deploy() that tests the issueOrder() method of a Human player. The test method sets up a GameEngineController, loads a map, adds two human players, and then issues a 'deploy' command to the first player. It asserts that the output contains the message 'Deploy order issued for Human'.

```
public class TestPlayer {  Inderjeet Singh Chauhan

    private final ByteArrayOutputStream outContent = new ByteArrayOutputStream(); 2 usages
    private final ByteArrayOutputStream errContent = new ByteArrayOutputStream(); 1 usage

    @Before  Inderjeet Singh Chauhan
    public void setUpStreams() {
        System.setOut(new PrintStream(outContent));
        System.setErr(new PrintStream(errContent));
    }

    @Test  Inderjeet Singh Chauhan
    public void testIssueOrder_Human_Deploy() {
        GameEngineController gc = new GameEngineController();
        gc.executeLoadMap( p_filename: AppConstants.MapsPath+"brasil.map");
        gc.executeAddGamePlayer( p_gamePlayer: "Human", Strategy.Human);
        gc.executeAddGamePlayer( p_gamePlayer: "Human1", Strategy.Human);
        Player player = GameEngineController.d_Players.getFirst();
        gc.executeAssignCountries();
        CommandValidator l_cmd=new CommandValidator();
        try {
            Command[] l_commands = l_cmd.validateCommand( p_command: "deploy Parana 5");
            player.setOrder(l_commands[0]);
        } catch (InvalidCommandException e) {
            throw new RuntimeException(e);
        }
        player.issueOrder();
        assertTrue(outContent.toString().contains("Deploy order issued for Human"));
    }
}
```

Screenshot of the changes before

Screenshot of the changes after

```

402         break;
403     }
404     } else {
405         switch (this.get_playerStrategyType().name()) {
406             case "Aggressive":
407                 set_playerStrategy(new AggressiveStrategy( p_player: this, new ArrayList<>(MapsController.getD_countri
408                 setD_currentOrder(get_playerStrategy().createOrder());
409                 d_orderList.add(getD_currentOrder());
410                 break;
411
412             case "Benevolent":
413                 set_playerStrategy(new BenevolentStrategy( p_player: this, new ArrayList<>(MapsController.getD_countri
414                 setD_currentOrder(get_playerStrategy().createOrder());
415                 d_orderList.add(getD_currentOrder());
416                 break;
417
418             case "Cheater":
419                 set_playerStrategy(new CheaterStrategy( p_player: this, new ArrayList<>(MapsController.getD_countries(
420                 setD_currentOrder(get_playerStrategy().createOrder());
421                 d_orderList.add(getD_currentOrder());
422                 break;
423
424             case "Random":
425                 set_playerStrategy(new RandomStrategy( p_player: this, new ArrayList<>(MapsController.getD_countries())
426                 setD_currentOrder(get_playerStrategy().createOrder());
427                 d_orderList.add(getD_currentOrder());
428                 break;

```

5. Introduced model for GameController

The `GameModel` class in Java represents the state of the game at any given point. It holds information about the players, the game map, the current player's turn, the number of completed turns, and the cards owned by players. This class is responsible for maintaining the state of the game and providing access to its various attributes. It is also designed to be serializable, allowing instances of `GameModel` to be serialized and deserialized, which is useful for saving and loading game states.

Screenshot of the code after the changes

```

    */
    public class GameModel implements Serializable { 8 usages  ± Mohammad Zaid Shaikh

        /** The list of players participating in the game. */
        private ArrayList<Player> d_Players; 2 usages

        /** The controller for managing game maps. */
        private MapsController d_Map; 2 usages

        /** The index of the current player in the list of players. */
        private int d_currentPlayer; 2 usages

        /** The number of completed turns in the game. */
        private int d_completedTurns; 2 usages

        /** The list of cards owned by players in the game. */
        private ArrayList<Player> d_cardsOwnedByPlayer; 2 usages

        /**
         * Constructs a new GameModel object with initial values.
         * Initializes the list of players, the game map, the list of cards owned by players, current player index, and completed turns.
         */
        public GameModel() { 2 usages  ± Mohammad Zaid Shaikh
            d_Players = new ArrayList<>();
            d_Map = new MapsController();
            d_cardsOwnedByPlayer = new ArrayList<>();
            d_currentPlayer = 0;
            d_completedTurns = 0;
        }

        /**

```