REFACTORING DOCUMENT

We carried out extensive refactoring in Build 2 of our SOEN 6441 project to enhance our codebase's modularity, maintainability, and structure. Using design patterns and solid principles, these refactoring efforts have produced a more reliable and practical application. An outline of the refactoring process is given in this report, together with information on possible goals, justifications, and descriptions of accurate refactoring procedures.

POTENTIAL REFACTORING TARGETS:

During our code analysis, we found potential areas where reworking could lead to improvements. Among the most critical possible refactoring targets included the following:

1. Extended Phase Switch Case:
    - **Reasoning:** The lengthy switch case used to control several game stages is against the Single Responsibility Principle (SRP) and reduces the code's maintainability.
    - **Refactoring:** Convert the switch case structure to a state pattern, with different state classes for each game phase.
2. Command Execution in the Switch situation:
    - **Reasoning:** Managing command execution in the switch situation involves giving a single code block many duties.
    - **Refactoring:** Use the Command pattern to divide commands into distinct classes, each in charge of a particular command.
3. Insufficient Modularity
    - **Reasoning:** It is difficult to expand or alter particular capabilities in the original code due to its lack of modularity.
    - **Refactoring:** Create modularity by dividing the code into clearly defined modules or folders that complement the architectural layout.
4. Observer Pattern for Logging:
    - **Reasoning:** By separating the duty of logging code operations, the code becomes more accessible to maintain.
    - **Refactoring:** To improve code organization and transparency, use the Observer pattern to log every action made during the game.

5. Presenting Random Cards for Players:
    - **Reasoning:** Including cards for players according to their gameplay is a valuable addition to the game.
    - **Refactoring:** By Warzone regulations, put in place the system that rewards players with random cards if they capture at least one nation during their turn.
6. Expanding Unit Testing:
    - **Reasoning:** Thorough unit tests are necessary to guarantee the dependability of the rewritten code.
    - **Refactoring:** Add more unit test cases to cover all the critical areas of the code, such as order execution, reinforcement calculations, map validation, and more.
7. Delineating Responsibilities:
    - **Reasoning:** Every class should have a single duty to uphold SOLID principles.
    - **Refactoring:** Whenever necessary, divide more complex classes into smaller ones so that each class in the codebase has a single, well-defined responsibility.
8. Command Validation:
    - **Reasoning:** The user experience depends on correctly validating user commands.
    - **Refactoring:** Provide strong command validation and indicate to users why a given command is invalid.
9. Redundant Code Blocks:
    - **Reasoning:** Maintaining redundant code might take much work.
    - **Refactoring:** Identify and eliminate redundant code blocks that perform similar tasks in different parts of the codebase.
10. Stated Documentation:
    - **Reasoning:** Well-documented Code is easier to read and update.
    - **Refactoring:** Ensure all methods and classes have appropriate Javadoc comments explaining parameters and return values.
11. Simplifying Error Handling:
    - **Reasoning:** Streamlining error handling can improve the readability and maintainability of the code.
    - **Refactoring:** Use streamlined and consistent error-handling techniques, like creating custom error-handling classes or using exceptions.

12. Code Formatting and Consistency:
    - **Reasoning:** Maintainability and readability of the code are enhanced by consistent code formatting.
    - **Refactoring:** Make sure the code follows a uniform layout throughout the project using an IDE auto-formatted.
13. Removal of Magic Numbers:
    - **Reasoning:** Adding named constants or variables in place of magic numbers makes the code easier to read.
    - **Refactoring:** Find all of the magic numbers in the code, then swap them out for useful variables or constants.
14. Code Duplication:
    - **Reasoning:** Duplicate code increases the likelihood of introducing bugs and complicates maintenance.
    - **Refactoring:** Identify and eliminate code duplication by extracting standard functionality into reusable methods or classes.
15. Consolidation of Constants:
    - **Reasoning:** Having several dispersed constants can make the code more difficult to update and less maintainable.
    - **Refactoring:** To make constants easier to manage and modify, group all the constants used in the code into a single Constants or Configuration class.
16. Reducing Cyclomatic Complexity:
    - **Reasoning:** Code with a high cyclomatic complexity may be challenging to read and update.
    - **Refactoring:** Dissect intricate procedures with a high cyclomatic complexity into smaller, easier-to-manage functions.
17. Getting Rid of Dead Code:
    - **Reasoning:** Unused and useless code can clog up the codebase.
    - **Refactoring:** To improve code cleanliness, find and eliminate any dead code, such as variables, imports, and unused methods.
18. Managing Exceptions with Grace:
    - **Reasoning:** Ignored exceptions can cause crashes and unsatisfactory user experiences.

- **Refactoring:** Provide informative error messages, stop application crashes, and catch and handle exceptions at the proper levels to implement graceful exception handling.

ACTUAL REFACTORING TARGETS:

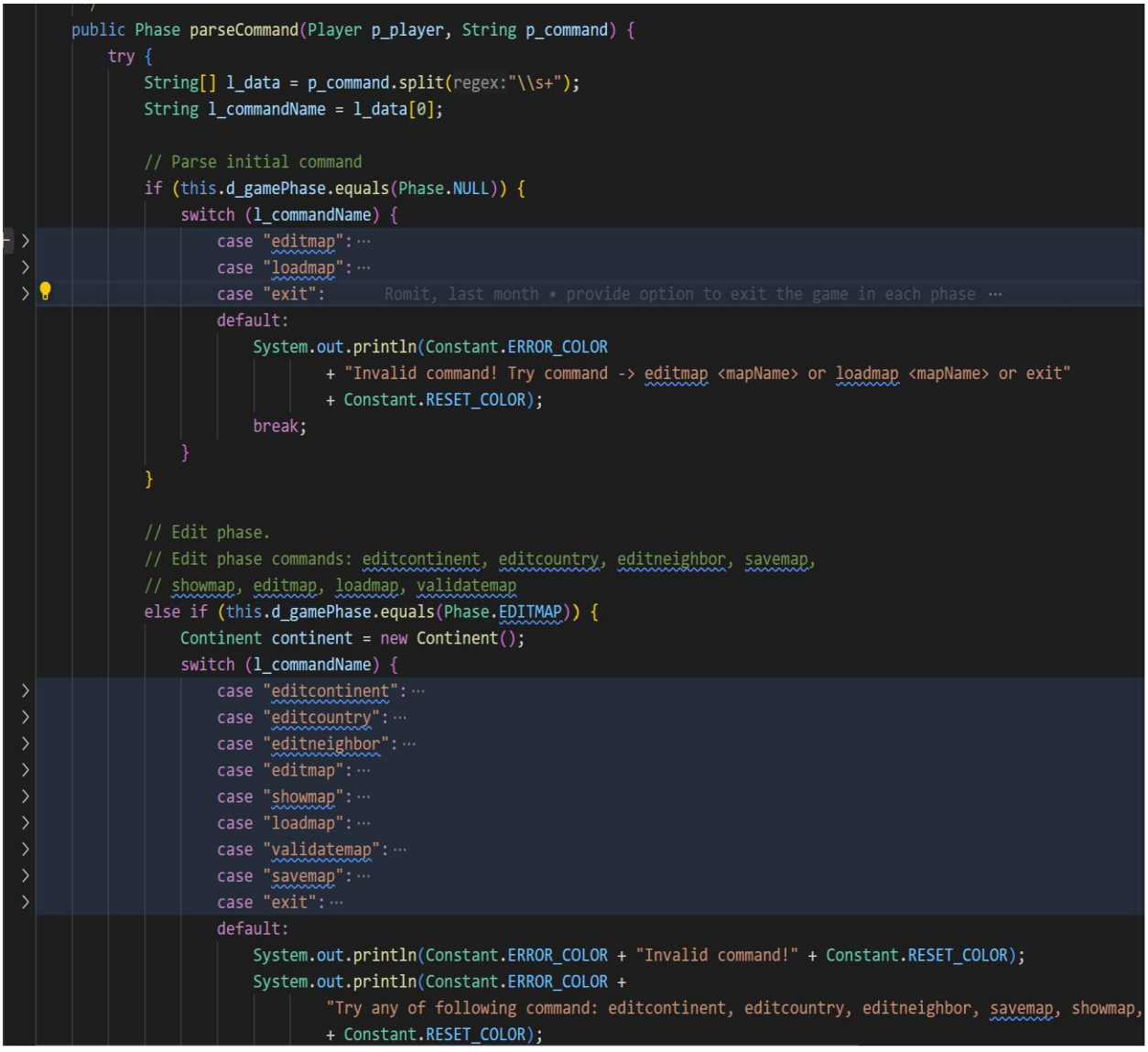
1. Implementing the State Pattern into Practice:
    - **Reasoning:** The Single Responsibility Principle (SRP) was broken, and the code became less maintainable due to the lengthy switch case structure used to control the game's various phases. We achieved better organization and concern separation by implementing the State pattern, which resulted in cleaner and more extensible code.
2. The Command Pattern's Introduction:
    - **Reasoning:** Managing command execution in the switch case required adding several duties to one code block. We were able to adhere to the SRP and make it simple to add new commands by separating the commands into distinct classes thanks to the introduction of the Command pattern.
3. Observer Pattern for Logging:
    - **Reasoning:** Recording code-related actions is a distinct duty handled with the Observer pattern's help. By recording every move made during the game, these modifications improved code organization and transparency.
4. Including Cards for Participants:
    - **Reasoning:** Adding the feature of players receiving random cards if they conquered at least one country in their turn is crucial for adhering to the game's rules and enhancing gameplay.
5. Extending Unit Testing:
    - **Reasoning:** We increased the scope of our unit testing to guarantee the dependability of the refactored code. This testing required writing extra test cases that addressed crucial code features like order execution, map validation, reinforcement computations, and more.

REFACTORING OPERATION:

1. Pattern of State

The code became complex and less maintainable due to the lengthy switch case structure that managed game phases, which had multiple responsibilities and violated the Single Responsibility Principle (SRP). In order to achieve better organization and concern separation, refactoring with the State pattern was required, resulting in cleaner and more extensible code.

- **Before Refactoring:**



```
public Phase parseCommand(Player p_player, String p_command) {
    try {
        String[] l_data = p_command.split(regex:"\\s+");
        String l_commandName = l_data[0];

        // Parse initial command
        if (this.d_gamePhase.equals(Phase.NULL)) {
            switch (l_commandName) {
                case "editmap":···
                case "loadmap":···
                case "exit":      Romit, last month • provide option to exit the game in each phase ···
                default:
                    System.out.println(Constant.ERROR_COLOR
                            + "Invalid command! Try command -> editmap <mapName> or loadmap <mapName> or exit"
                            + Constant.RESET_COLOR);
                    break;
            }
        }

        // Edit phase.
        // Edit phase commands: editcontinent, editcountry, editneighbor, savemap,
        // showmap, editmap, loadmap, validatemap
        else if (this.d_gamePhase.equals(Phase.EDITMAP)) {
            Continent continent = new Continent();
            switch (l_commandName) {
                case "editcontinent":···
                case "editcountry":···
                case "editneighbor":···
                case "editmap":···
                case "showmap":···
                case "loadmap":···
                case "validatemap":···
                case "savemap":···
                case "exit":···
                default:
                    System.out.println(Constant.ERROR_COLOR + "Invalid command!" + Constant.RESET_COLOR);
                    System.out.println(Constant.ERROR_COLOR +
                            "Try any of following command: editcontinent, editcountry, editneighbor, savemap, showmap,
                            + Constant.RESET_COLOR);
```

After Refactoring:

```java
public class StartUpPhase extends Phase {

    /**
     * Constructor that initializes the GameEngine context in the Phase class.
     *
     * @param p_gameEngine The GameEngine context.
     * @param p_gameState  The current game state.
     */
    public StartUpPhase(GameEngine p_gameEngine, GameState p_gameState) {
        super(p_gameEngine, p_gameState);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    protected void deploy(GameState p_gameState, Player p_player, String[] p_args) {
        this.printInvalidCommandInState();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    protected void showMap(GameEngine p_gameEngine, GameState p_gameState, String[] p_args) {…

    /**
     * {@inheritDoc}
     */
    public void editMap(GameEngine p_gameEngine, GameState p_gameState, String[] p_args) {…

    /**
     * {@inheritDoc}
     */
    public void editContinent(GameEngine p_gameEngine, GameState p_gameState, String[] p_args) {…
```

2. Introduction of the Command Pattern

The SRP was broken when handling command execution inside a lengthy switch case, which brought several duties into one code block. By dividing commands into distinct classes and adhering to the SRP, the Command pattern was introduced, making it simple to add new commands.

Before refactoring:

```
// EXECUTE_ORDER phase commands : execute, showmap, exit
else if (this.d_gamePhase.equals(Phase.EXECUTE_ORDERS)) {
    switch (l_commandName) {
        case "execute":…
        case "showmap":
            if (!isValidCommandArgument(l_data, p_expectedArgsLength:1)) {
                System.out.println(Constant.ERROR_COLOR
                        + "Invalid number of arguments for showmap command"
                        + Constant.RESET_COLOR);
                break;
            }
            MapHelper l_gameMap = new MapHelper();
            l_gameMap.showMap(this.d_playerList, this.d_gameMap);
            break;
        case "exit":
            System.out.println(
                    Constant.SUCCESS_COLOR + "Finish!" + Constant.RESET_COLOR);
            System.exit(status:0);
            break;
        default:
            System.out.println(Constant.ERROR_COLOR +
                    "Try -> showmap or execute or exit"
                    + Constant.RESET_COLOR);
            break;
    }
}
```

After Refactoring:

```java
@Override
  public void initPhase() {
    while (this.d_gameEngine.getCurrentGamePhase() instanceof OrderExecutionPhase) {
      try {
        int l_numOfOrders = this.d_gameState.getUnexecutedOrders().size();
        if (l_numOfOrders == 0) {
          System.out.println("Orders already executed!!");
          break;
        } else {
          System.out.println("Total orders : " + l_numOfOrders);

          while (!this.d_gameState.getUnexecutedOrders().isEmpty()) {
            Order l_order = this.d_gameState.getUnexecutedOrders().poll();
            boolean l_executed = l_order.execute();
            l_order.setGameState(this.d_gameState);
          }
          Iterator<Player> l_iterator = this.d_gameState.getPlayerList().listIterator();
          while (l_iterator.hasNext()) {
            Player l_player = l_iterator.next();
            l_player.showCards();
          }
          this.d_gameEngine.setIssueOrderPhase();
        }
      } catch (Exception e) {
        this.d_gameEngine.setGameEngineLog(e.getMessage(), "effect");
      }
    }
  }
```

3. Observer Pattern for Logging

Recording activities within the code is an independent duty. The Observer pattern had to be implemented for logging to improve code organization and transparency, which records each action made during the game.

```java
public class GameLogger implements Observer {

  /**
   * The `LogEntryBuffer` object that provides updated log entries.
   */
  LogEntryBuffer d_logEntryBuffer;

  /**
   * Writes log entries from the `LogEntryBuffer` object to the log file.
   *
   * @param p_observable The `LogEntryBuffer` object that is observed.
   * @param p_object     An object (not used in this method).
   */
  @Override
  public void update(Observable p_observable, Object p_object) {
    this.d_logEntryBuffer = (LogEntryBuffer) p_observable;
    // File l_logfile = new File("LogFile.txt");
    String l_logMessage = this.d_logEntryBuffer.getLogMessage();

    try {
      if (l_logMessage.equals("Initializing the Game ......" + System.lineSeparator() + System.lineSeparator())) {
        Files.newBufferedWriter(Paths.get(first:"LogFile.txt"), StandardOpenOption.TRUNCATE_EXISTING).write(str:" ");
      }
      Files.write(Paths.get(first:"LogFile.txt"), l_logMessage.getBytes(StandardCharsets.US_ASCII), StandardOpenOption.CREATE,
          StandardOpenOption.APPEND);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

4. Integration of Cards for Players

It is imperative to incorporate the function of randomly assigning cards to players in response to their actions to uphold the game's rules and improve gameplay.

```java
    /**...*/
    public GameState d_gameState;

    /**...*/
    public String getCardType() {
        return this.d_cardType;
    }

    /**...*/
    public Card() {
    }

    /**...*/
    public Card(String p_cardType) {
        this.d_cardType = p_cardType;
        this.d_gameState = new GameState();
    }

    /**...*/
    public void setGameState(GameState p_gameState) {
        this.d_gameState = p_gameState;
    }

    /**...*/
    public void createCard() {
        d_cardType = this.getRandomCard();
    }

    /**...*/
    public void createCard(String p_cardType) {
        d_cardType = p_cardType;
    }

    /**...*/
    public String getRandomCard() {
        Random randomGenerator = new Random();
        int index = randomGenerator.nextInt(this.d_cardsList.length);
        // return this.d_cardsList[index];
        return this.d_cardsList[1];      You, 1 hour ago • Uncommitted changes
    }
}
```

5. Extending Unit Testing

To guarantee that the refactored code is reliable, thorough unit tests are necessary. We ensured robust testing of the codebase by adding more test cases that covered important features of the code.

Following are test classes lists:

- TestSuite.java
- GameEngineTest.java
- AdvanceTest.java
- AdvanceTestM.java
- AirliftTest.java
- BlockadeTest.java
- BombTest.java
- CardTest.java
- ContinentTest.java
- CountryTest.java
- DeployTest.java
- DiplomacyTest.java
- GameMapTest.java
- GameStateTest.java
- LogEntryBufferTest.java
- MapHelperTest.java
- OrderExecutionPhaseTest.java
- PhaseTest.java
- PlayerTest.java
- StartUpPhaseTest.java
- MapValidatorTest.java
- UtilTest.java
- GameLoggerTest.java