REFACTORING DOCUMENT

In Build 3 of our SOEN 6441 project, we performed considerable reworking to improve the modularity, maintainability, and structure of our software. These refactoring efforts resulted in a more reliable and practical application by utilizing design patterns and strong principles. This paper includes a summary of the refactoring process, as well as information on possible goals, justifications, and descriptions of accurate refactoring processes.

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. **Adapter Pattern for save/load map:**

- **Reasoning :** When saving or loading a map, the application might interact with different external formats or data structures. Using an Adapter Pattern helps in managing these differences. For instance, if the map data structure within the application doesn't align directly with the desired format for saving/loading (e.g., JSON, XML, binary), an adapter can be implemented to translate between the internal map representation and the external format. This segregation ensures that the core map-handling logic remains unaffected by format changes or updates, enhancing maintainability.

- **Refactoring :** Map structures may change or new features may be added as software develops, causing compatibility problems with older versions of saved maps. It is possible to achieve backward or forward compatibility by employing the Adapter Pattern. In order to ensure that older map versions can be properly loaded into newer versions of the application and vice versa, adapters can be implemented to manage different versions of map data. This enhances robustness and guarantees a more seamless map handling experience by preventing the application from crashing when working with maps saved in different versions.

2. **Game Execution in the Switch situation :**

- **Reasoning :** Initially, using a switch statement for game execution with limited modes (e.g., tournament, solitary) may be sufficient. However, as new modes are added, the switch statement becomes cumbersome and difficult to maintain. By refactoring this into a strategy pattern, new game modes may be added without altering existing code, improving maintainability and scalability.

- **Refactoring :** Decouple the selection logic from the game execution by reorganizing the game execution using a strategy pattern that encapsulates each game mode (tournament, solo) into its own class. This removes the need to change current conditional statements for mode selection and makes it easier to add new game modes by defining new strategy classes.

3. **Lack of Sufficient Modularity :**

- **Reasoning :** The original code's lack of modularity hinders the ability to easily expand or modify specific functionalities.

- **Refactoring :** Establish modularity by dividing the code into distinct modules or folders that align with the architectural layout, enhancing its organization and flexibility.

4. **Enhancing Gameplay through Strategy Implementation:**

- **Reasoning :** Integrating diverse strategies that allow players to interact more dynamically with the computer player serves as a significant enhancement to the current gameplay experience.

- **Refactoring :** Implement adaptable strategies that adhere to Warzone regulations, allowing players to select these strategies in either tournament mode or single-player mode. This ensures compliance with gaming standards while offering strategic choices across different modes for an enriched gaming experience.

5. **Elimination of Redundant Code Blocks :**

- **Reasoning :** Managing redundant code demands excessive effort and resources, potentially leading to increased maintenance overhead and complexity within the codebase.

- **Refactoring :** The strategy involves the identification and subsequent removal of duplicate code segments scattered across various areas of the codebase. By consolidating similar functionalities, it streamlines maintenance efforts and reduces the chances of inconsistencies or errors arising from redundant code repetition.

6. **Enhanced Command Parsing for Adding Game Players :**

- **Reasoning :** The command parsing underwent an update to accommodate player strategies within the game setup process.

- **Refactoring :** The code modification broadened the command acceptance criteria to handle commands of lengths 3 or 4. Specifically, this alteration allows the incorporation of player strategies during the addition of players into the game using the command format: -add <playername> <strategy>. This adjustment ensures seamless integration of player strategies during game setup while maintaining command flexibility and clarity.

7. **Implementing the Strategy pattern for the Player's issueOrder() :**

- **Reasoning :** Implementing the Strategy pattern for the issueOrder() method allows for more modular and interchangeable behaviors within the Player class. This approach separates the algorithmic behavior from the Player class, facilitating easier addition or alteration of various strategies without modifying the core Player class. It enables dynamic behavior changes during runtime without impacting the Player class structure.

- **Refactoring :** Restructure the issueOrder() method within the Player class to utilize the Strategy pattern. Create separate strategy classes representing different behaviors (e.g., aggressive strategy, cheater strategy, etc). Each strategy class should implement a common interface defining the issueOrder() method.

8. **Incorporating different computer player behaviors:**

- **Reasoning :** Employing the Strategy pattern for different computer player behaviors caters to diverse gaming strategies. By encapsulating specific behaviors within separate strategy classes, it enables the Player class to exhibit varying behaviors during the execution of the issueOrder() method. This approach facilitates extensibility by adding new strategies and modifying existing ones without directly modifying the Player class, promoting code maintainability and scalability.

- **Refactoring :** During the main development phase, create distinct strategy classes for various computer player behaviors, such as aggressive, defensive, or random strategies. Implement these strategies using the same interface employed for the Player's behavior strategies. Integrate these strategies dynamically with the Player class, allowing the Player to execute diverse behaviors based on the selected strategy during gameplay.

9. **Enhanced User Experience and Persistence :**

- **Reasoning :** Implementing game save and load functionalities enriches the user experience by allowing players to save their progress and resume the game from where they left off. It provides a sense of continuity and convenience to the players.

Additionally, this feature ensures the persistence of game state, enabling users to reload the exact state of the game from a saved file, fostering flexibility and continuity in gameplay.

- **Refactoring :** Implement savegame and loadgame commands in the game interface/console, refactoring the game logic to serialize and store critical game data when using savegame. Similarly, loadgame should deserialize saved data from a file, restoring the game to the exact saved point, ensuring uninterrupted gameplay using file I/O or serialization mechanisms.

10. Varied Player Behaviors for Diverse Gameplay :

- **Reasoning :** Introducing different player behaviors like aggressive, benevolent, random, cheater, and human adds depth and variety to the game. Each behavior offers distinct strategies, allowing for varied gameplay experiences that involve both user-controlled decision-making and automated behaviors.

- **Refactoring :** Create separate classes for each player behavior (aggressive, benevolent, random, cheater). Refactor the game to instantiate the appropriate behavior classes for each player type, allowing them to exhibit their specific strategies during gameplay. Maintain a dedicated behavior class for the human player that interacts with user commands for decision-making and order creation. This segregation of behaviors enables distinct and engaging gameplay experiences tailored to different player types.

11. Elimination of dead code :

- **Reasoning :** The accumulation of unused and irrelevant code within the codebase can create clutter, potentially impeding code readability and maintainability.

- **Refactoring :** Enhancing code cleanliness involves actively searching for and removing dead code elements, such as obsolete variables, unnecessary imports, and unused methods. This refactoring practice not only declutters the codebase but also enhances its efficiency by reducing unnecessary processing and potential confusion for developers maintaining the code.

12. Handling Exceptions Effectively :

- **Reasoning :** Neglected exceptions can lead to application crashes and unsatisfactory user interactions.

- **Refactoring :** Improve by displaying informative error messages, preventing application crashes, and appropriately catching and managing exceptions at relevant levels to implement a more graceful and controlled exception handling mechanism.

13. Improving Unit Testing Coverage :

- **Reasoning :** Extensive unit tests are vital to ensure the reliability of the revised code.

- **Refactoring :** Enhance the existing unit test suite by including additional test cases that cover critical aspects of the code, such as order execution, reinforcement calculations, map validation, and other crucial functionalities.

14. Documenting Code Effectively:

- **Reasoning :** Thoroughly documented code facilitates easier comprehension and modifications.

- **Refactoring :** Guarantee that all classes and methods contain suitable Javadoc comments, elucidating parameters and return values for better understanding and maintainability.

15. Validating Commands:

- **Reasoning :** Ensuring a positive user experience relies on accurately validating user commands.

- **Refactoring :** Strengthen command validation mechanisms and offer clear indications to users regarding the reasons for command invalidity.

ACTUAL REFACTORING TARGETS:

1) **Adapter Pattern for save/load map:**
   Utilizing the Adapter Pattern during map save or load operations addresses potential discrepancies between the application's internal map structure and external data formats. For instance, when the internal map format differs from the desired save/load format (e.g., JSON, XML, binary), implementing an adapter facilitates seamless translation between these structures. This approach safeguards the core map-handling logic from the

impacts of format alterations, ensuring easier maintenance and preserving the application's stability.

2) **Game Execution in the Switch situation :**
Using a switch statement for game execution with limited modes (e.g., tournament, solo) may suffice at first. However, as new modes are added, the switch statement becomes bulky and difficult to maintain. Refactoring this into a strategy pattern enables for the seamless addition of new game modes without altering existing code, improving maintainability and scalability.

3) **Implementing the Strategy pattern for the Player's issueOrder() :**
Implementing the Strategy pattern for the issueOrder() method allows for more modular and interchangeable behaviors within the Player class. This approach separates the algorithmic behavior from the Player class, facilitating easier addition or alteration of various strategies without modifying the core Player class. It enables dynamic behavior changes during runtime without impacting the Player class structure.

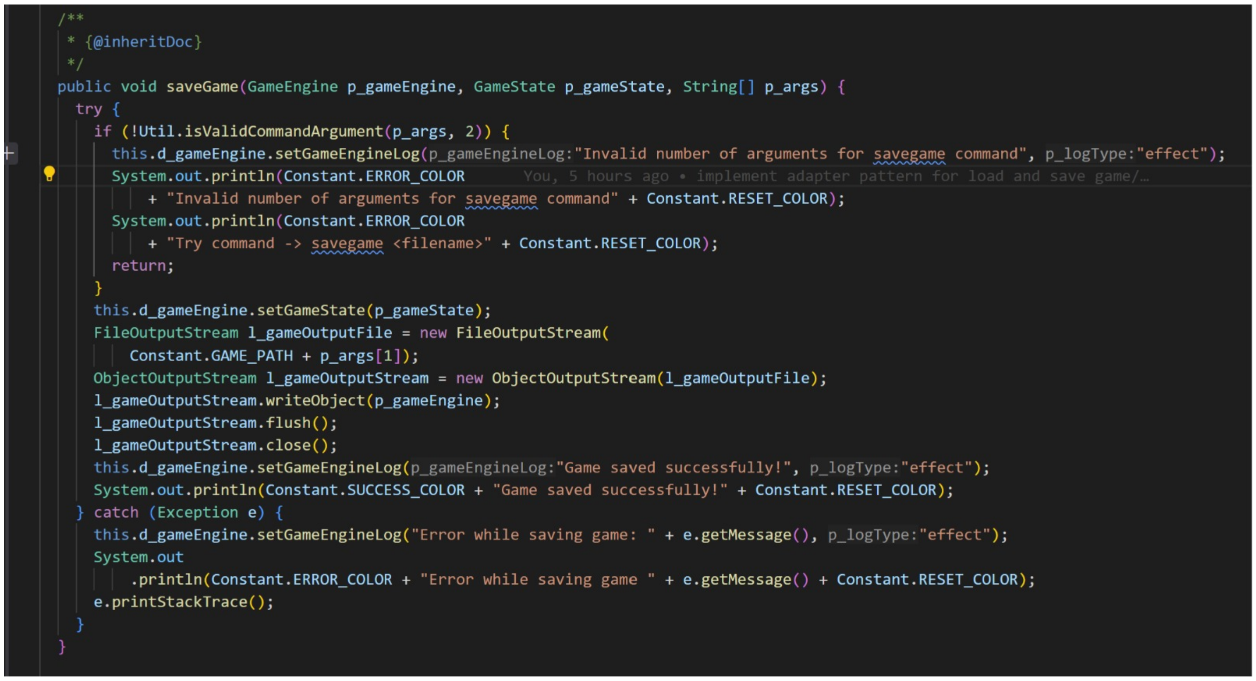
4) **Enhanced Command Parsing for Adding Game Players :**
The command parsing underwent an update to accommodate player strategies within the game setup process.

5) **Improving Unit Testing Coverage :**
Extensive unit tests are vital to ensure the reliability of the revised code.

REFACTORING OPERATIONS:

1) **Adapter Pattern for Map Save/Load :**
As software evolves, map structures may change or new features may be added, causing compatibility issues with older versions of saved maps. Backward and forward compatibility can be achieved by using the Adapter Pattern. Adapters can be used to manage different versions of map data, ensuring that older map versions can be loaded correctly into newer versions of the application and vice versa. This prevents the application from breaking when dealing with maps saved in different versions, improving robustness and ensuring a smoother user experience during map handling.

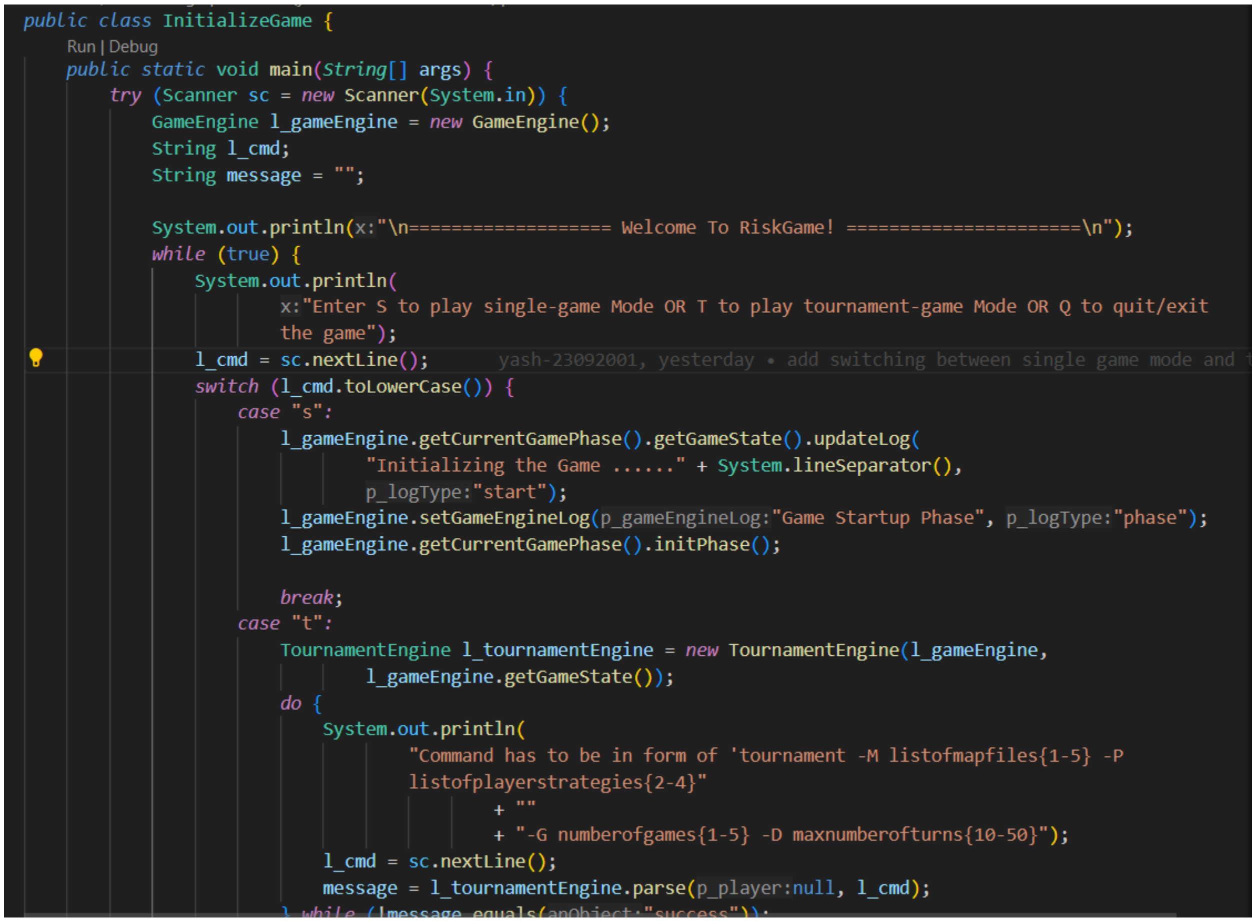
After Refactoring :



```
/**
 * {@inheritDoc}
 */
public void saveGame(GameEngine p_gameEngine, GameState p_gameState, String[] p_args) {
  try {
    if (!Util.isValidCommandArgument(p_args, 2)) {
      this.d_gameEngine.setGameEngineLog(p_gameEngineLog:"Invalid number of arguments for savegame command", p_logType:"effect");
      System.out.println(Constant.ERROR_COLOR       You, 5 hours ago • implement adapter pattern for load and save game/…
          + "Invalid number of arguments for savegame command" + Constant.RESET_COLOR);
      System.out.println(Constant.ERROR_COLOR
          + "Try command -> savegame <filename>" + Constant.RESET_COLOR);
      return;
    }
    this.d_gameEngine.setGameState(p_gameState);
    FileOutputStream l_gameOutputFile = new FileOutputStream(
        Constant.GAME_PATH + p_args[1]);
    ObjectOutputStream l_gameOutputStream = new ObjectOutputStream(l_gameOutputFile);
    l_gameOutputStream.writeObject(p_gameEngine);
    l_gameOutputStream.flush();
    l_gameOutputStream.close();
    this.d_gameEngine.setGameEngineLog(p_gameEngineLog:"Game saved successfully!", p_logType:"effect");
    System.out.println(Constant.SUCCESS_COLOR + "Game saved successfully!" + Constant.RESET_COLOR);
  } catch (Exception e) {
    this.d_gameEngine.setGameEngineLog("Error while saving game: " + e.getMessage(), p_logType:"effect");
    System.out
        .println(Constant.ERROR_COLOR + "Error while saving game " + e.getMessage() + Constant.RESET_COLOR);
    e.printStackTrace();
  }
}
```

2) **Game Execution in the Switch situation :**
Restructure game execution by implementing a strategy pattern in which each game mode (tournament, solo) is encapsulated in its own class, thus decoupling selection logic from game execution. This makes adding new game modes easier by creating new strategy classes and eliminates the need to modify existing conditional statements for mode selection.

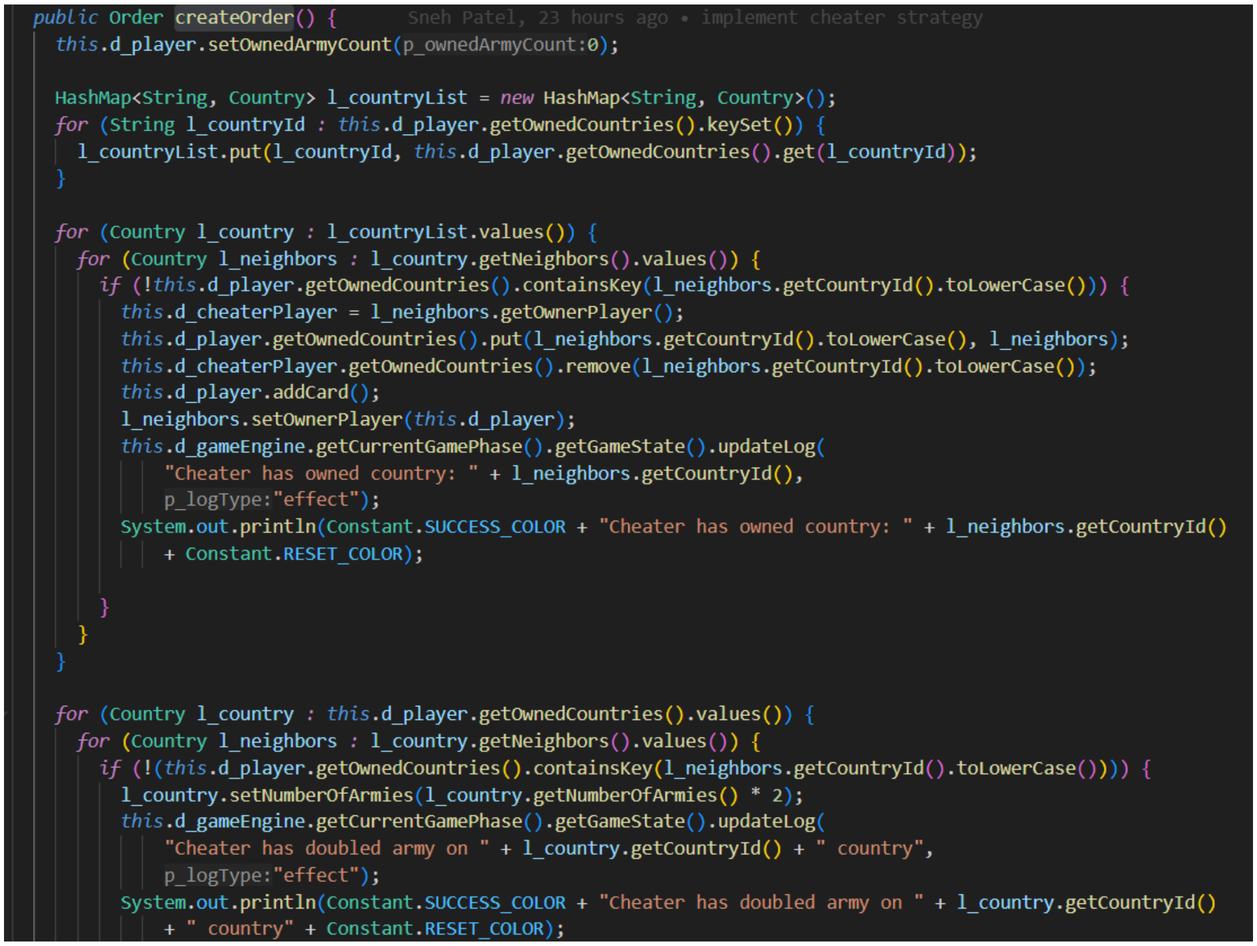
After Refactoring :



```
public class InitializeGame {
    Run | Debug
    public static void main(String[] args) {
        try (Scanner sc = new Scanner(System.in)) {
            GameEngine l_gameEngine = new GameEngine();
            String l_cmd;
            String message = "";

            System.out.println(x:"\n================== Welcome To RiskGame! =====================\n");
            while (true) {
                System.out.println(
                        x:"Enter S to play single-game Mode OR T to play tournament-game Mode OR Q to quit/exit
                        the game");
                l_cmd = sc.nextLine();        yash-23092001, yesterday • add switching between single game mode and t
                switch (l_cmd.toLowerCase()) {
                    case "s":
                        l_gameEngine.getCurrentGamePhase().getGameState().updateLog(
                                "Initializing the Game ......" + System.lineSeparator(),
                                p_logType:"start");
                        l_gameEngine.setGameEngineLog(p_gameEngineLog:"Game Startup Phase", p_logType:"phase");
                        l_gameEngine.getCurrentGamePhase().initPhase();

                        break;
                    case "t":
                        TournamentEngine l_tournamentEngine = new TournamentEngine(l_gameEngine,
                                l_gameEngine.getGameState());
                        do {
                            System.out.println(
                                    "Command has to be in form of 'tournament -M listofmapfiles{1-5} -P
                                    listofplayerstrategies{2-4}"
                                            + ""
                                            + "-G numberofgames{1-5} -D maxnumberofturns{10-50}");
                            l_cmd = sc.nextLine();
                            message = l_tournamentEngine.parse(p_player:null, l_cmd);
                        } while (!message.equals(anObject:"success"));
```

3) **Implementing the Strategy pattern for the Player's issueOrder() :**
To use the Strategy pattern, restructure the issueOrder() method in the Player class. Create distinct strategy classes to represent various behaviors (e.g., aggressive strategy, cheater strategy, etc.). Each strategy class should implement the issueOrder() method of a common interface.

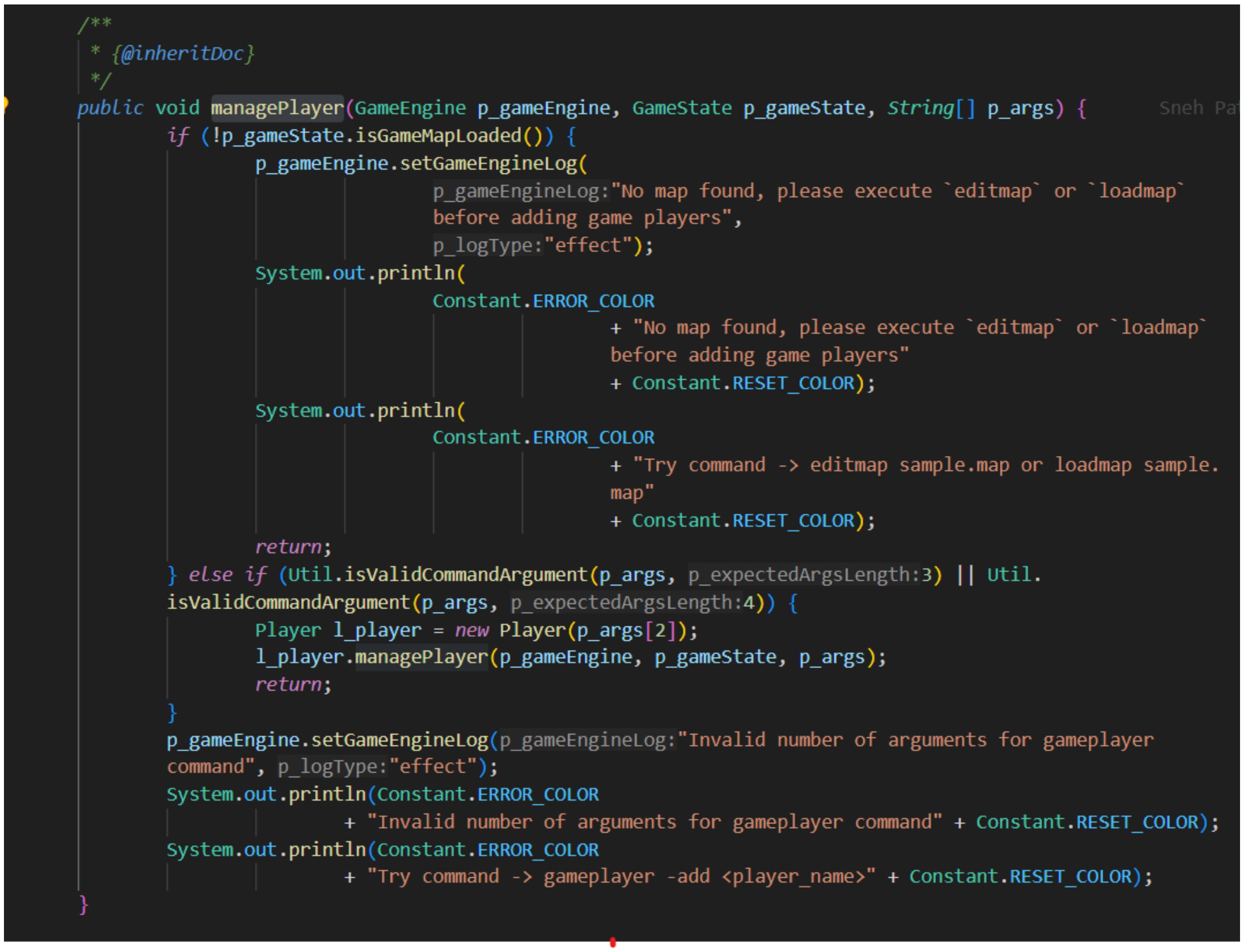
After Refactoring:



```
public Order createOrder() {        Sneh Patel, 23 hours ago • implement cheater strategy
    this.d_player.setOwnedArmyCount(p_ownedArmyCount:0);

    HashMap<String, Country> l_countryList = new HashMap<String, Country>();
    for (String l_countryId : this.d_player.getOwnedCountries().keySet()) {
      l_countryList.put(l_countryId, this.d_player.getOwnedCountries().get(l_countryId));
    }

    for (Country l_country : l_countryList.values()) {
      for (Country l_neighbors : l_country.getNeighbors().values()) {
        if (!this.d_player.getOwnedCountries().containsKey(l_neighbors.getCountryId().toLowerCase())) {
          this.d_cheaterPlayer = l_neighbors.getOwnerPlayer();
          this.d_player.getOwnedCountries().put(l_neighbors.getCountryId().toLowerCase(), l_neighbors);
          this.d_cheaterPlayer.getOwnedCountries().remove(l_neighbors.getCountryId().toLowerCase());
          this.d_player.addCard();
          l_neighbors.setOwnerPlayer(this.d_player);
          this.d_gameEngine.getCurrentGamePhase().getGameState().updateLog(
              "Cheater has owned country: " + l_neighbors.getCountryId(),
              p_logType:"effect");
          System.out.println(Constant.SUCCESS_COLOR + "Cheater has owned country: " + l_neighbors.getCountryId()
              + Constant.RESET_COLOR);

        }
      }
    }

    for (Country l_country : this.d_player.getOwnedCountries().values()) {
      for (Country l_neighbors : l_country.getNeighbors().values()) {
        if (!(this.d_player.getOwnedCountries().containsKey(l_neighbors.getCountryId().toLowerCase()))) {
          l_country.setNumberOfArmies(l_country.getNumberOfArmies() * 2);
          this.d_gameEngine.getCurrentGamePhase().getGameState().updateLog(
              "Cheater has doubled army on " + l_country.getCountryId() + " country",
              p_logType:"effect");
          System.out.println(Constant.SUCCESS_COLOR + "Cheater has doubled army on " + l_country.getCountryId()
              + " country" + Constant.RESET_COLOR);
```

4) **Enhanced Command Parsing for Adding Game Players :**
The code change made it possible to accept commands with lengths of three or four. In particular, this modification permits the addition of player strategies when players are added to the game with the following command format: -add <playername> <strategy>. This modification preserves the flexibility and clarity of the command while guaranteeing the smooth integration of player strategies during game setup.

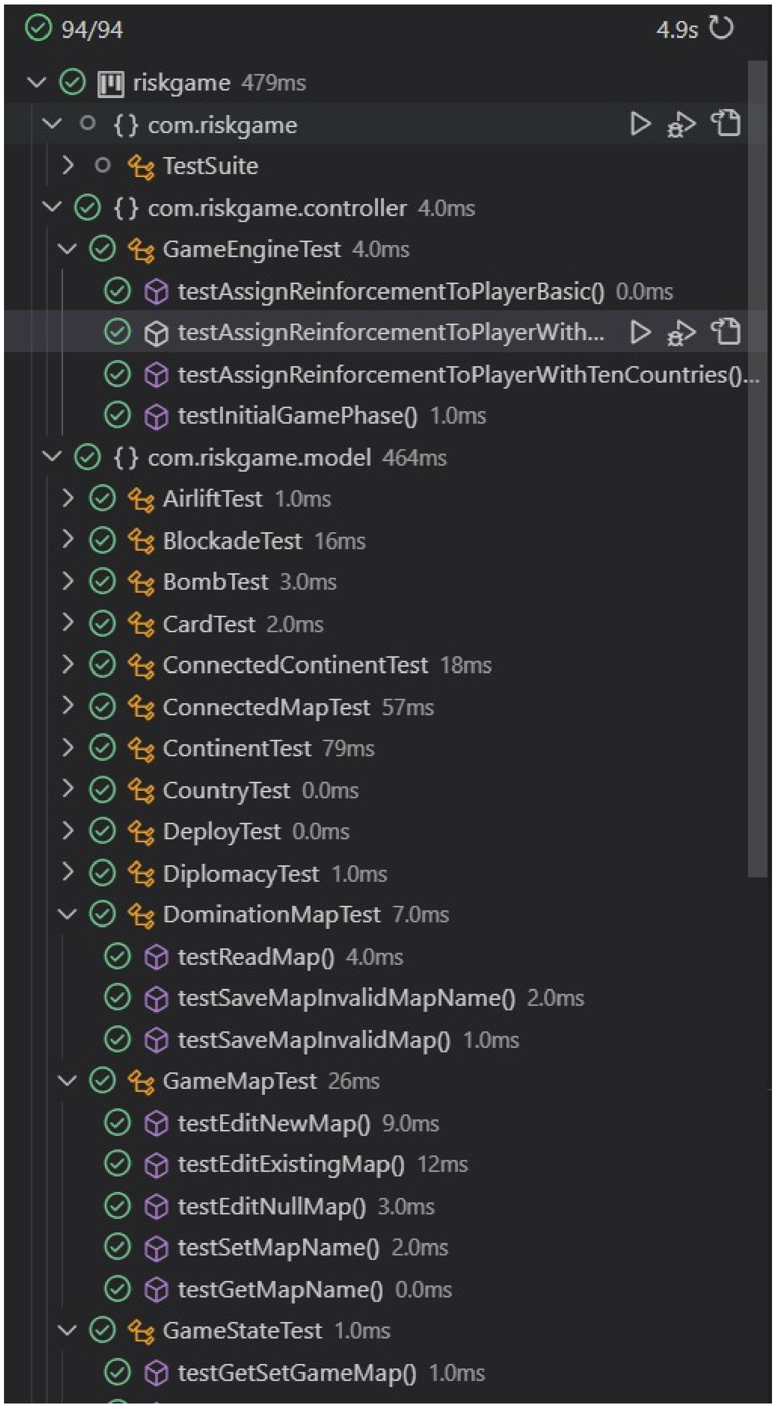
After Refactoring:



```
/**
 * {@inheritDoc}
 */
public void managePlayer(GameEngine p_gameEngine, GameState p_gameState, String[] p_args) {        Sneh Pa
        if (!p_gameState.isGameMapLoaded()) {
                p_gameEngine.setGameEngineLog(
                                p_gameEngineLog:"No map found, please execute `editmap` or `loadmap`
                                before adding game players",
                                p_logType:"effect");
                System.out.println(
                                Constant.ERROR_COLOR
                                                + "No map found, please execute `editmap` or `loadmap`
                                                before adding game players"
                                                + Constant.RESET_COLOR);
                System.out.println(
                                Constant.ERROR_COLOR
                                                + "Try command -> editmap sample.map or loadmap sample.
                                                map"
                                                + Constant.RESET_COLOR);
                return;
        } else if (Util.isValidCommandArgument(p_args, p_expectedArgsLength:3) || Util.
        isValidCommandArgument(p_args, p_expectedArgsLength:4)) {
                Player l_player = new Player(p_args[2]);
                l_player.managePlayer(p_gameEngine, p_gameState, p_args);
                return;
        }
        p_gameEngine.setGameEngineLog(p_gameEngineLog:"Invalid number of arguments for gameplayer
        command", p_logType:"effect");
        System.out.println(Constant.ERROR_COLOR
                        + "Invalid number of arguments for gameplayer command" + Constant.RESET_COLOR);
        System.out.println(Constant.ERROR_COLOR
                        + "Try command -> gameplayer -add <player_name>" + Constant.RESET_COLOR);
}
```

5) **Improving Unit Testing Coverage :**
Expand the current unit test suite with new test cases that cover important code features like map validation, order execution, reinforcement calculations, and other important functionalities.

After Refactoring:



```
94/94                                                          4.9s
riskgame 479ms
  { } com.riskgame
    TestSuite
  { } com.riskgame.controller 4.0ms
    GameEngineTest 4.0ms
      testAssignReinforcementToPlayerBasic() 0.0ms
      testAssignReinforcementToPlayerWith...
      testAssignReinforcementToPlayerWithTenCountries()...
      testInitialGamePhase() 1.0ms
  { } com.riskgame.model 464ms
    AirliftTest 1.0ms
    BlockadeTest 16ms
    BombTest 3.0ms
    CardTest 2.0ms
    ConnectedContinentTest 18ms
    ConnectedMapTest 57ms
    ContinentTest 79ms
    CountryTest 0.0ms
    DeployTest 0.0ms
    DiplomacyTest 1.0ms
    DominationMapTest 7.0ms
      testReadMap() 4.0ms
      testSaveMapInvalidMapName() 2.0ms
      testSaveMapInvalidMap() 1.0ms
    GameMapTest 26ms
      testEditNewMap() 9.0ms
      testEditExistingMap() 12ms
      testEditNullMap() 3.0ms
      testSetMapName() 2.0ms
      testGetMapName() 0.0ms
    GameStateTest 1.0ms
      testGetSetGameMap() 1.0ms
```