SOEN 6441: BUILD 3 REFACTORING DOCUMENT

TEAM 16

TEAM MEMBERS:

Sanjay Bhargav Pabbu- 40265895

Blesslin Jeba Shiny Augustin Moses - 40266442

Susmitha Mamula - 40277873

Poojitha Bhupalli - 40232528

Piyush Gupta- 40294464

Mahfuzzur Rahman - 40293992

LIST OF POTENTIAL REFACTORING TARGETS

1. **Introduce Adapter Pattern:** Users can now specify filenames for both domination and conquest maps. Through an adapter class, conquest map data can be converted to domination map data, and vice versa.
2. **Implement Regex in mapLoader.java: The** mapLoader class will adopt regex instead of fixed array indices to extract data from the map file.
3. **Apply Strategy Pattern:** Player strategies are undergoing a comprehensive reconstruction to align with the strategy pattern specifications.
4. **Eliminate Explicit Phase Naming:** Player strategies will be revamped entirely to adhere to the strategy pattern specifications, removing the need for explicitly stating phase names to transition to a new phase.
5. **Replace 'System.exit()' with 'return':** The application will now gracefully finish and close, shifting away from reliance on 'System.exit()', which abruptly shuts down the JVM and the application.
6. **New Parameter Addition (WinnerPlayer):** The connectivity class will incorporate a new parameter called WinnerPlayer to enhance the efficiency of determining the game winner.
7. **Refactor setPlayers in gameplayer:** setPlayers method is being refactored to utilize strategies as player names during the tournament phase.
8. **Update MapLoader to Handle Various Characters**: MapLoader is being enhanced to successfully load maps containing characters such as ',', '.', '&', '-' etc.
9. **Introduce recordResult Function**: A new function, recordResult, will be created to store details regarding the outcome of each game played.

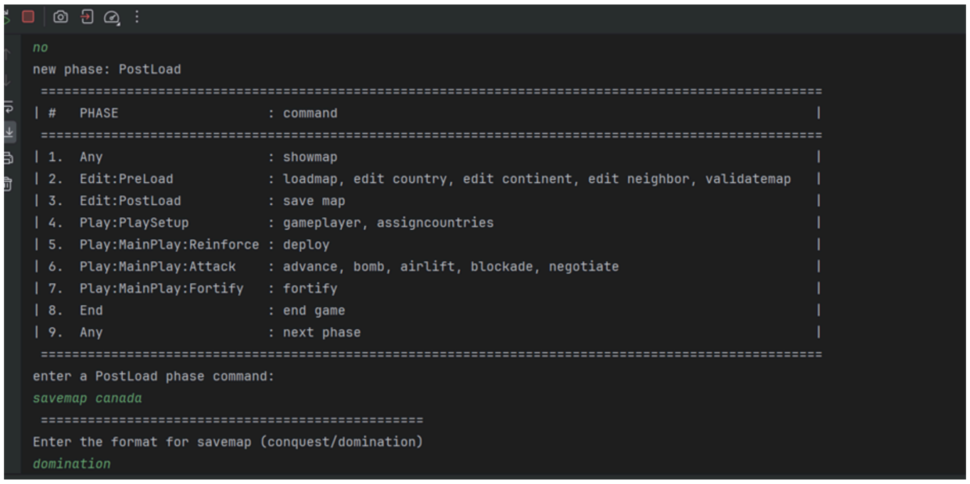
10. **Revise Terminal Print Messages:** Terminal print messages for commands are being reformatted to enhance user-friendliness.
11. **Enhance Help Method:** The help method will be updated to include additional commands for the user's convenience.
12. **Modify Attack Phase Execution Order:** The attack phase execution order will be adjusted to handle null orders returned by the next_order method in the player.java class.
13. **Standardize Indentation in attack.java:** Indentation in the attack.java file will be adjusted to conform to project standards.
14. **Remove Unnecessary Test Print Statements**: All unnecessary print statements used for testing purposes are being removed from the code.
15. **Clean up Code Comments:** Unnecessary comments within the code are being removed to improve code cleanliness, with relevant details moved to the respective function's Javadoc documentation.
16. **Refactor issueOrder() method:** The refactoring of the issue_order() method in the Player class to utilize the Strategy pattern represents a significant design improvement.
17. **Refactor saveMap() method:** The refactored method now prompts the user to input the desired map format.

<u>REFACTORING OPERATIONS:</u>

1. **Adapter Pattern Implementation:**

   a) Users can now input filenames for both domination and conquest maps. An adapter class facilitates the conversion of conquest map data to domination map data and vice versa.

   b) Tests have been developed, including tournamentValidation, to ensure the proper execution of the Adapter pattern for converting conquest map data to domination map data.

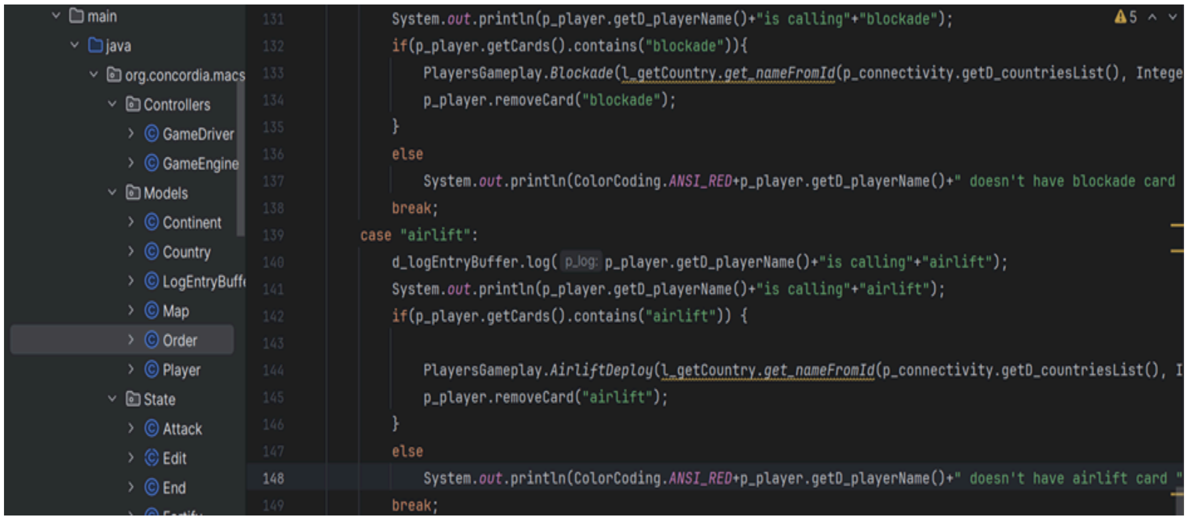


```
no
new phase: PostLoad
 =====================================================================================================
| #   PHASE                   : command                                                              |
 =====================================================================================================
| 1.  Any                     : showmap                                                              |
| 2.  Edit:PreLoad            : loadmap, edit country, edit continent, edit neighbor, validatemap    |
| 3.  Edit:PostLoad           : save map                                                             |
| 4.  Play:PlaySetup          : gameplayer, assigncountries                                          |
| 5.  Play:MainPlay:Reinforce : deploy                                                               |
| 6.  Play:MainPlay:Attack    : advance, bomb, airlift, blockade, negotiate                          |
| 7.  Play:MainPlay:Fortify   : fortify                                                              |
| 8.  End                     : end game                                                             |
| 9.  Any                     : next phase                                                           |
 =====================================================================================================
enter a PostLoad phase command:
savemap canada
 =================================================
Enter the format for savemap (conquest/domination)
domination
```

Fig: SaveMap format in console

2. **Refactor by Changes in command pattern:**

   a) After refactoring we consider validating each command.



```
            System.out.println(p_player.getD_playerName()+"is calling"+"blockade");
            if(p_player.getCards().contains("blockade")){
                PlayersGameplay.Blockade(l_getCountry.get_nameFromId(p_connectivity.getD_countriesList(), Intege
                p_player.removeCard("blockade");
            }
            else
                System.out.println(ColorCoding.ANSI_RED+p_player.getD_playerName()+" doesn't have blockade card
            break;
        case "airlift":
            d_logEntryBuffer.log( p_log: p_player.getD_playerName()+"is calling"+"airlift");
            System.out.println(p_player.getD_playerName()+"is calling"+"airlift");
            if(p_player.getCards().contains("airlift")) {

                PlayersGameplay.AirliftDeploy(l_getCountry.get_nameFromId(p_connectivity.getD_countriesList(), I
                p_player.removeCard("airlift");
            }
            else
                System.out.println(ColorCoding.ANSI_RED+p_player.getD_playerName()+" doesn't have airlift card "
            break;
```

Fig: Order.java

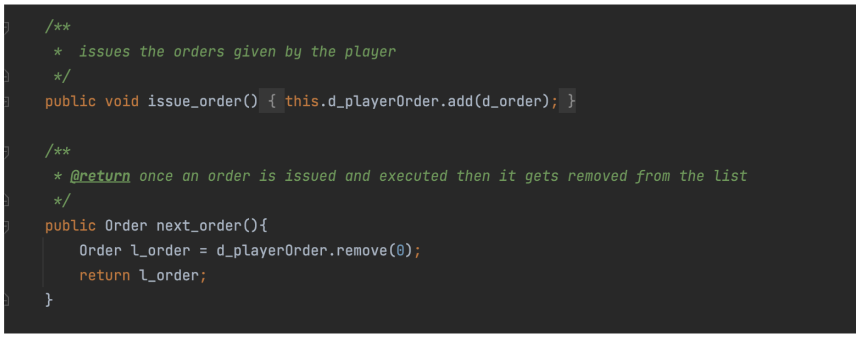
   b) Tests, such as tournamentValidation, phaseValidation, and MapLoaderTest, are conducted to validate the functionality of the command pattern.

3. Refactor the Player's issueOrder() method to use Strategy Pattern

The refactoring of the issue_order() method in the Player class to utilize the Strategy pattern represents a significant design improvement. Originally, the method for issuing orders might have been tightly coupled to the Player class, limiting the flexibility in how orders were created and processed.

Before Refactoring

Initially, the issue_order() method directly contained the logic to create and manage orders, potentially leading to a high degree of coupling between the Player's behavior and the specific order logic.



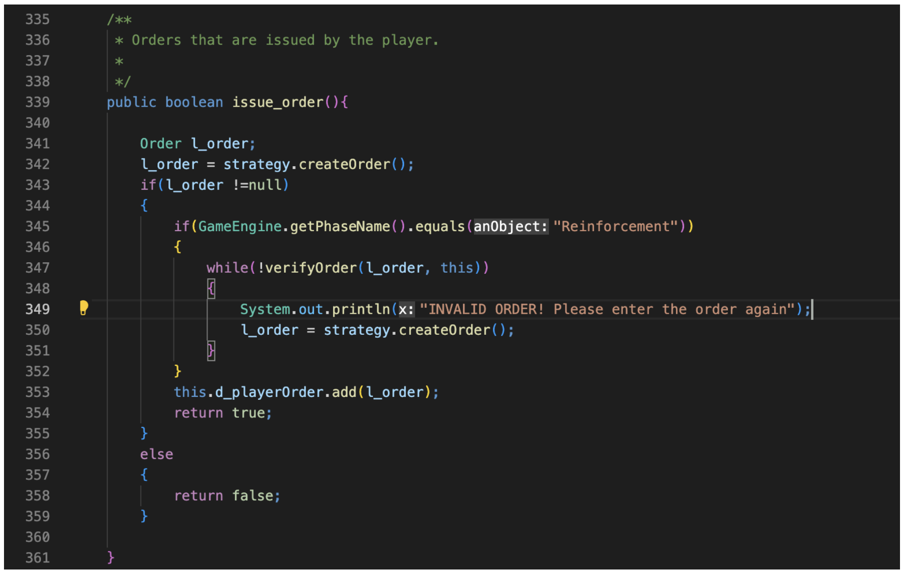
```
/**
 *  issues the orders given by the player
 */
public void issue_order() { this.d_playerOrder.add(d_order); }

/**
 * @return once an order is issued and executed then it gets removed from the list
 */
public Order next_order(){
    Order l_order = d_playerOrder.remove(0);
    return l_order;
}
```

Fig: Player.java before refactoring

After Refactoring

With the Strategy pattern applied, the issue_order() method delegates the responsibility of creating orders to a separate strategy object. This object is an instance of a class that implements the PlayerStrategy interface, which defines the createOrder() method used to generate orders dynamically based on the strategy in use.



```
/**
 * Orders that are issued by the player.
 *
 */
public boolean issue_order(){

    Order l_order;
    l_order = strategy.createOrder();
    if(l_order !=null)
    {
        if(GameEngine.getPhaseName().equals(anObject: "Reinforcement"))
        {
            while(!verifyOrder(l_order, this))
            {
                System.out.println(x: "INVALID ORDER! Please enter the order again");
                l_order = strategy.createOrder();
            }
        }
        this.d_playerOrder.add(l_order);
        return true;
    }
    else
    {
        return false;
    }

}
```

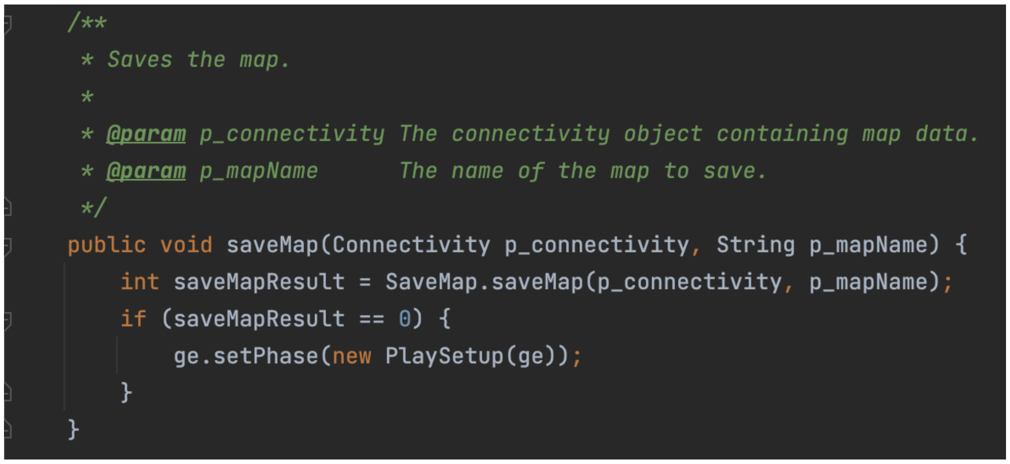
Fig: Player.java after refactoring

4. Refactor PostLoad.java's saveMap() method

This refactoring operation was necessary to improve the flexibility, adaptability, and user experience of the map-saving functionality in the PostLoad phase of the game engine.

Before Refactoring:

In the PostLoad class of build 2, the saveMap() method directly called the SaveMap.saveMap() method to save the map. It didn't provide any flexibility in choosing the map format. The code was not adaptable to saving maps in different formats based on user input.

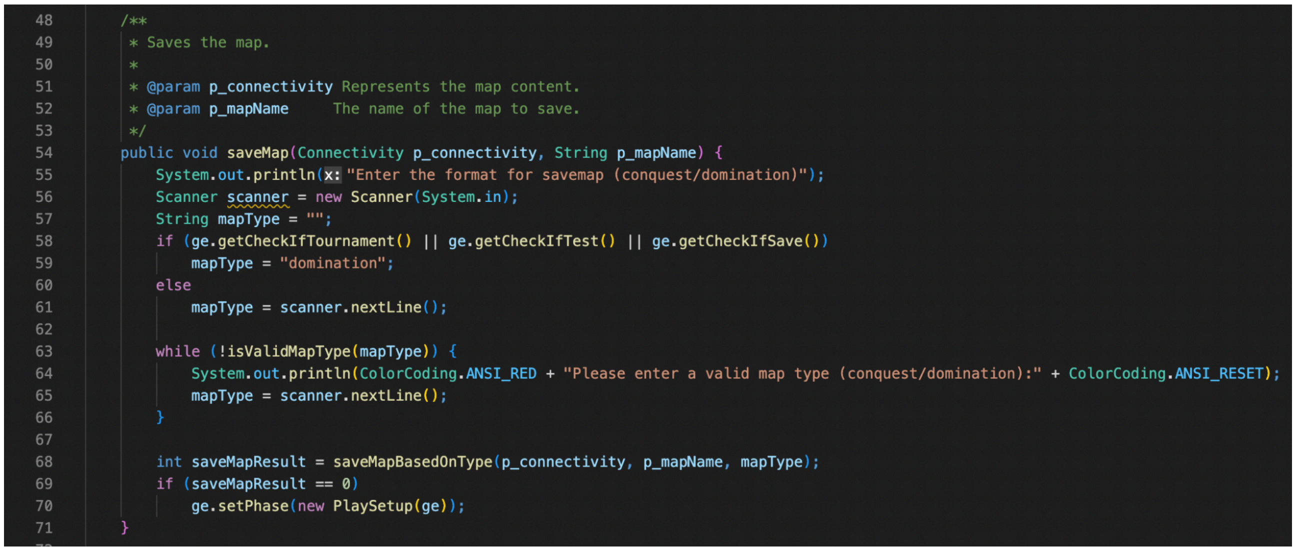


```java
/**
 * Saves the map.
 *
 * @param p_connectivity The connectivity object containing map data.
 * @param p_mapName      The name of the map to save.
 */
public void saveMap(Connectivity p_connectivity, String p_mapName) {
    int saveMapResult = SaveMap.saveMap(p_connectivity, p_mapName);
    if (saveMapResult == 0) {
        ge.setPhase(new PlaySetup(ge));
    }
}
```

Fig: PostLoad.java before refactoring

After Refactoring:

In build 3, the saveMap() method was refactored to provide flexibility in choosing the map format (conquest/domination) before saving. The refactored method now prompts the user to input the desired map format. It then validates the input and delegates the saving operation to different methods (ConquestSaveMap.conquestMapSaver() or SaveMap.saveMap()) based on the chosen format.



```java
/**
 * Saves the map.
 *
 * @param p_connectivity Represents the map content.
 * @param p_mapName      The name of the map to save.
 */
public void saveMap(Connectivity p_connectivity, String p_mapName) {
    System.out.println(x: "Enter the format for savemap (conquest/domination)");
    Scanner scanner = new Scanner(System.in);
    String mapType = "";
    if (ge.getCheckIfTournament() || ge.getCheckIfTest() || ge.getCheckIfSave())
        mapType = "domination";
    else
        mapType = scanner.nextLine();

    while (!isValidMapType(mapType)) {
        System.out.println(ColorCoding.ANSI_RED + "Please enter a valid map type (conquest/domination):" + ColorCoding.ANSI_RESET);
        mapType = scanner.nextLine();
    }

    int saveMapResult = saveMapBasedOnType(p_connectivity, p_mapName, mapType);
    if (saveMapResult == 0)
        ge.setPhase(new PlaySetup(ge));
}
```

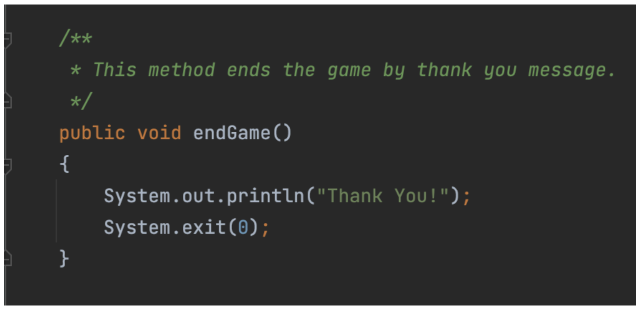
Fig: PostLoad.java after refactoring

5. Transition from 'System.exit()' to Return Statements:

The refactoring operation on the endGame method from Build 2 to Build 3 introduces significant changes to enhance functionality and user interaction at the end of the game phase. This modification allows for a more dynamic end-game scenario by giving players the option to save their game before exiting, which was not present in the initial implementation. The method is designed to handle situations differently if the game is in tournament or test mode, bypassing the save option directly to ensure smooth automated operations without unnecessary interruptions.

Before Refactoring

This method simply prints a "Thank You!" message and terminate the application using System.exit(0). The method does not provide any option for the player to save the game before exiting. Once the end phase is reached, the game concludes immediately without any choice for the player to preserve their progress.

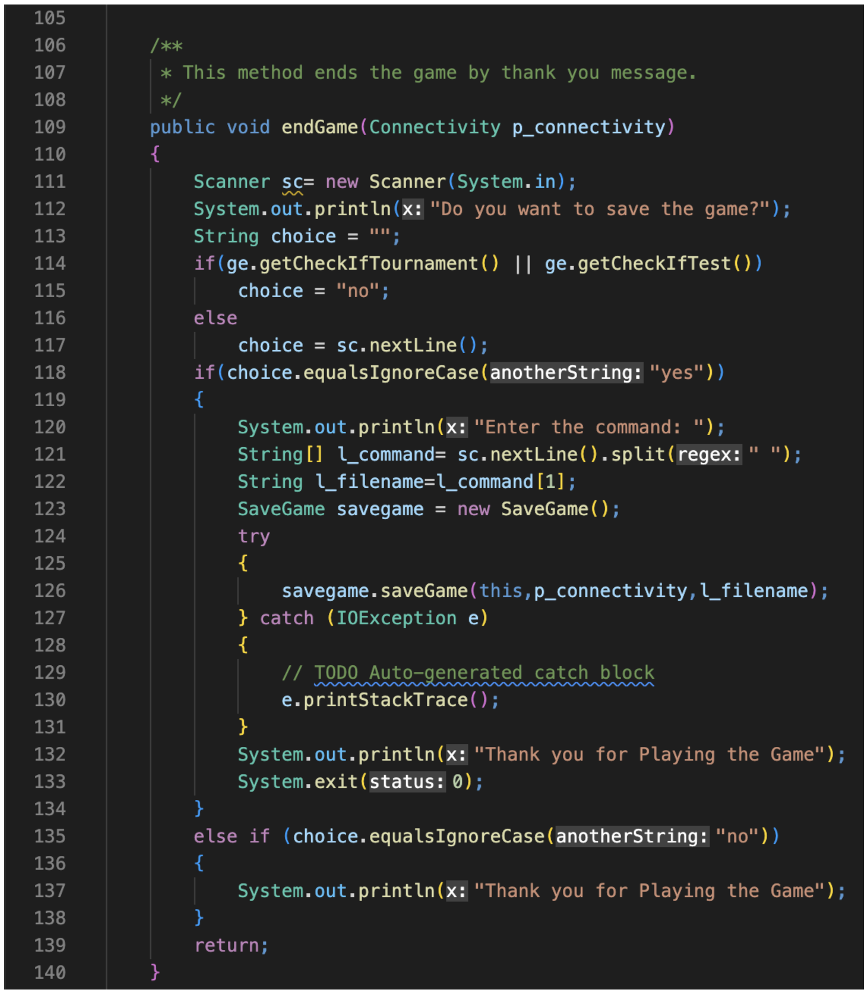


```java
/**
 * This method ends the game by thank you message.
 */
public void endGame()
{
    System.out.println("Thank You!");
    System.exit(0);
}
```

Fig: End.java before refactoring

After Refactoring

It now prompts the player with a choice to save the game before exiting. This addition significantly enhances user experience by allowing players to preserve their game state. The method is designed to handle situations differently if the game is in tournament or test mode, bypassing the save option directly to ensure smooth automated operations without unnecessary interruptions. It also includes basic error handling for potential IO exceptions during the save operation, which is crucial for a robust application.



```java

    /**
     * This method ends the game by thank you message.
     */
    public void endGame(Connectivity p_connectivity)
    {
        Scanner sc= new Scanner(System.in);
        System.out.println(x: "Do you want to save the game?");
        String choice = "";
        if(ge.getCheckIfTournament() || ge.getCheckIfTest())
            choice = "no";
        else
            choice = sc.nextLine();
        if(choice.equalsIgnoreCase(anotherString: "yes"))
        {
            System.out.println(x: "Enter the command: ");
            String[] l_command= sc.nextLine().split(regex: " ");
            String l_filename=l_command[1];
            SaveGame savegame = new SaveGame();
            try
            {
                savegame.saveGame(this,p_connectivity,l_filename);
            } catch (IOException e)
            {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            System.out.println(x: "Thank you for Playing the Game");
            System.exit(status: 0);
        }
        else if (choice.equalsIgnoreCase(anotherString: "no"))
        {
            System.out.println(x: "Thank you for Playing the Game");
        }
        return;
    }
```

Fig: End.java after refactoring