

# **SOEN 6441 - Advanced Programming Practices**

Project - Warzone Risk Based game (Build #3)

Winter 2025

## **Refactoring Document**

Submitted by **DABSV**

**Arvind Nachiappan Lakshmanan - 40310757**

**Barath Sundararaj - 40324920**

**Devasenan Murugan - 40302170**

**Swathi Priya Pasumarthi- 40322468**

Submitted to

**Prof. M. Taleb**

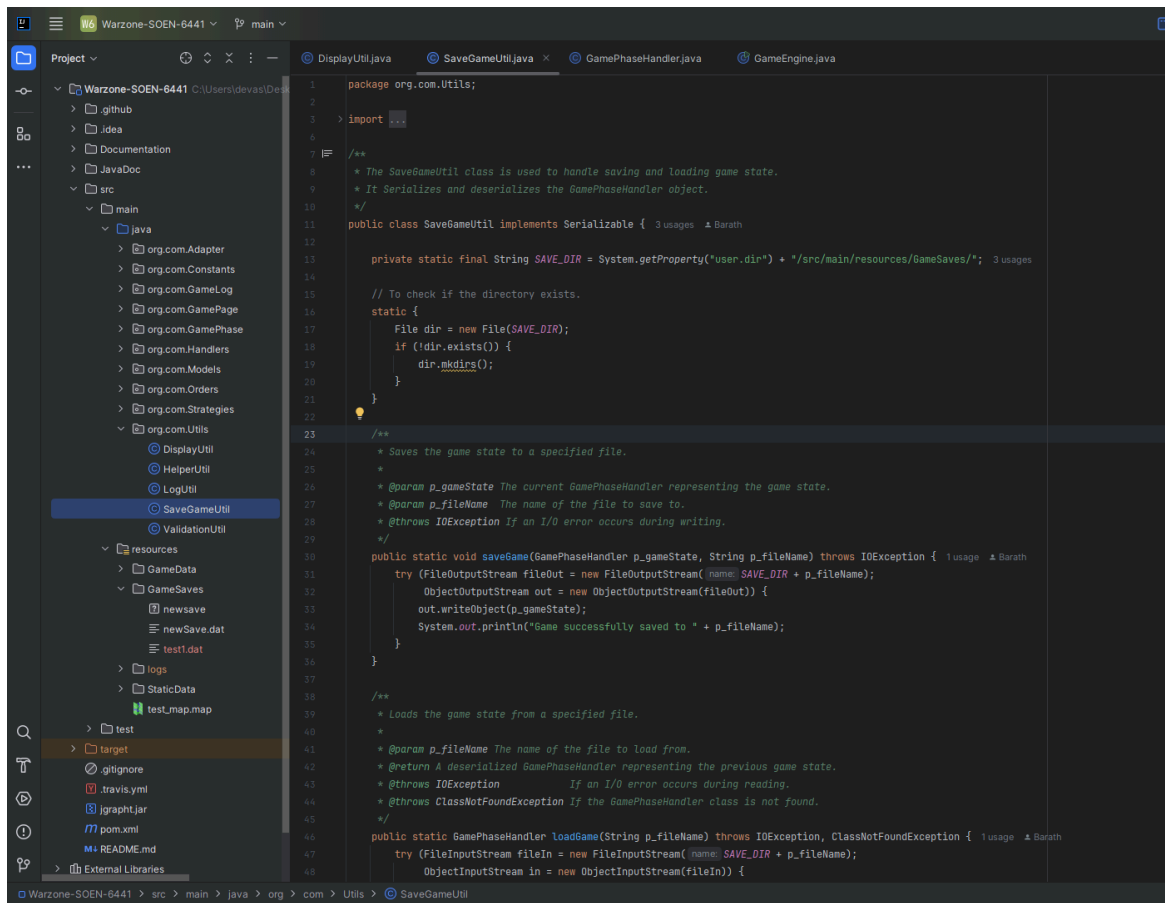
## **Potential Refactoring Targets**

Given below is the list of potential refactoring targets:

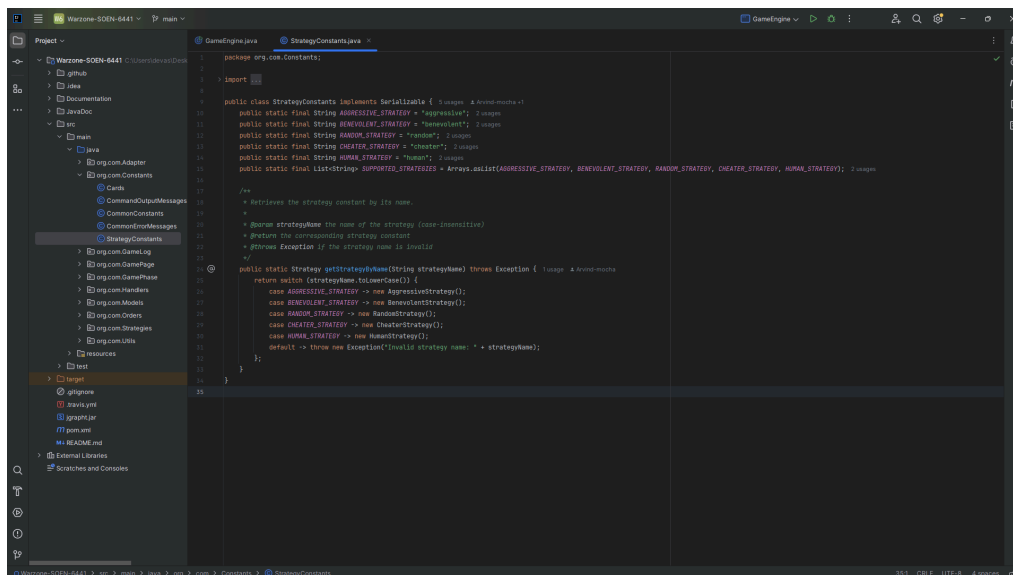
- 1. Updated all the Java classes to implement Serializable** - This was implemented to make the classes compatible for saving and loading game files.
- 2. Encapsulated functions to handle different map types in the Adapter Module** - for the purpose of handling domination or conquest maps.
- 3. Decomposed GameEngine into GameEngine and GameModeExecuter** - Either for a Single mode or Tournament mode the core logic and operations for executing the game is the same. So the GameModeExecuter has the logic to run a single game.
- 4. Added new Model for the Tournament feature**
- 5. Added new Module for implementing all the computer Strategies**
- 6. Command validation for invalid option types and param length** - Validation of commands were properly handled to overcome the errors in case the player entered the wrong option type or parameter length.
- 7. Command validation for invalid and redundant game phase commands**
- 8. Updating the access modifiers of the methods** - This was discovered to ensure that methods have appropriate levels of visibility in order to maintain encapsulation and control access to specific functionalities.
- 9. Updating Unintuitive variable names**
- 10. Removing dead code and unused Import statements**- We identified this by seeing code and import statements that appear to have no effect on the program's behavior.
- 11. Replacing if..else..if with switch..case** - This target was chosen to improve code readability and maintainability in situations where a switch statement would be preferable to a series of if-else statements
- 12. Adding constants for the valid commands** - Constants were introduced to reduce the repetition of the same statements.



## 14. Added savegame & loadgame features



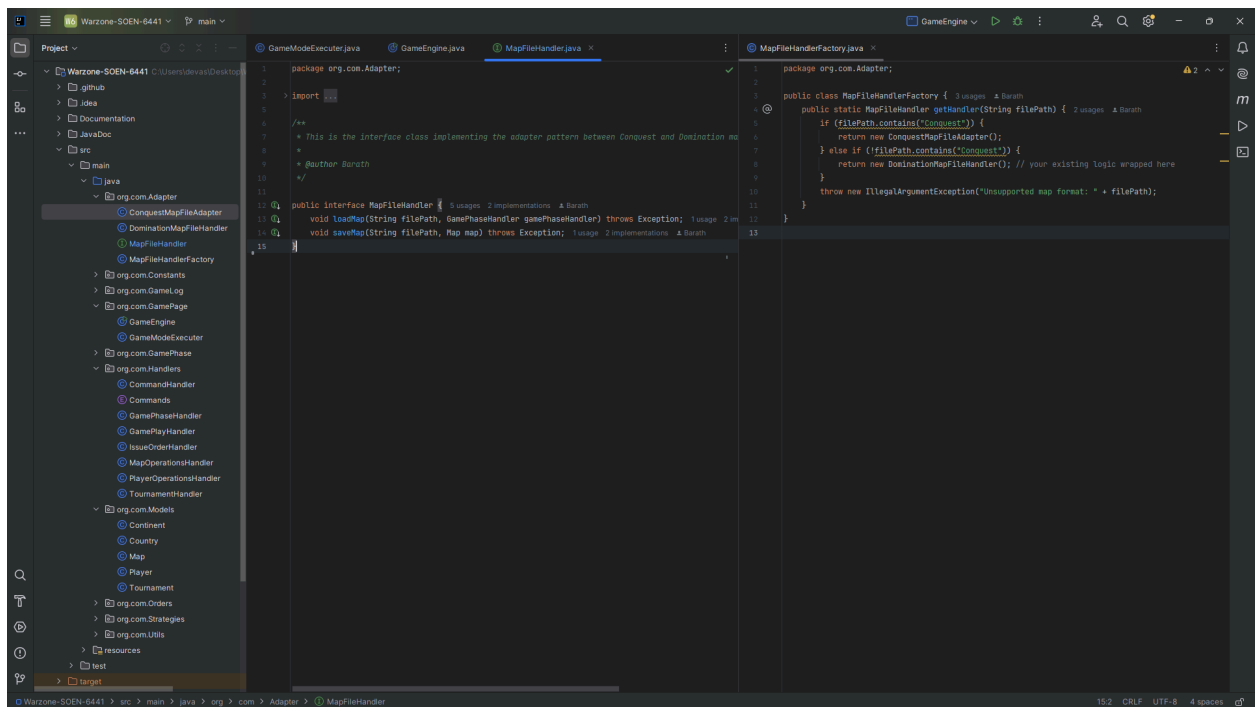
## 15. Adding Common constants for Game Strategies



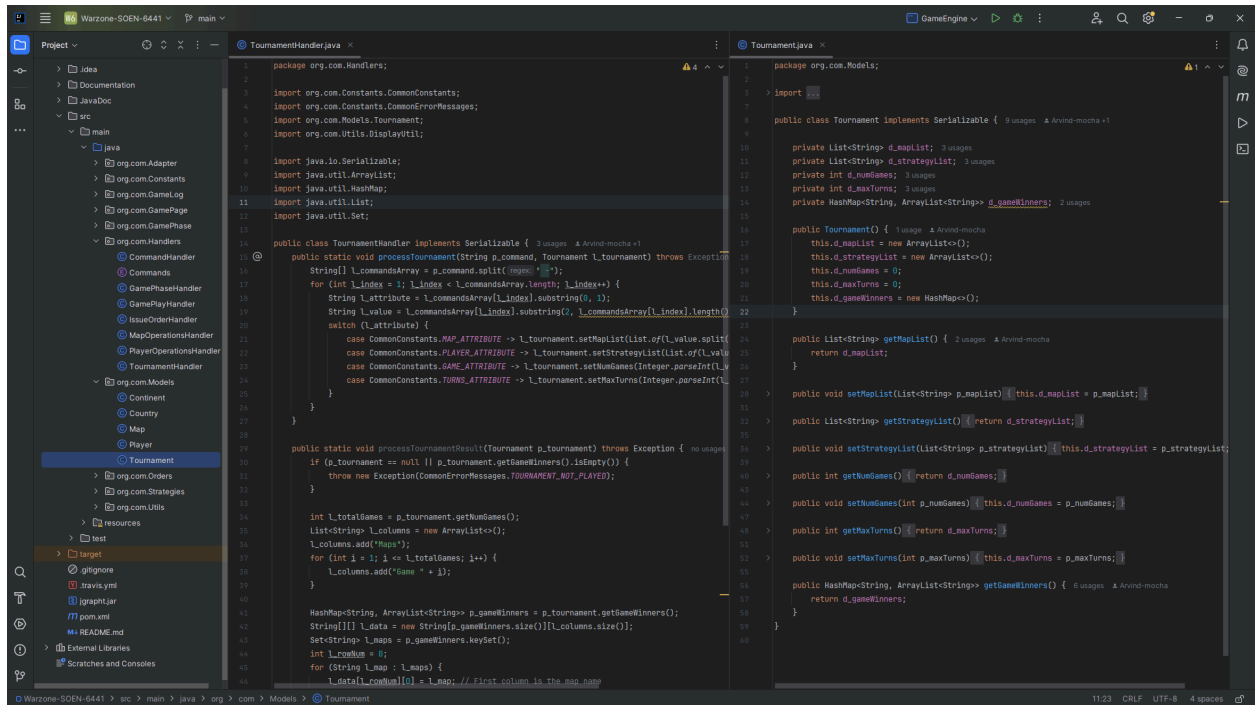
## Actual Refactoring Targets

Below are the 5 refactoring targets chosen from the above mentioned list of all potential targets, based on the requirements established in Build 3.

1. **Updated all the Java classes to implement Serializable** - This was implemented to make the classes compatible for saving and loading game files.
2. **Encapsulated functions to handle different map types in the Adapter Module**



### 3. Added new Model for the Tournament feature -



The screenshot shows an IDE with two files open: `TournamentHandler.java` and `Tournament.java`. The `TournamentHandler` class implements `Serializable` and contains a `processTournament` method that takes a command and a tournament object. It processes the command by splitting it into attributes and values, then sets the tournament's strategy list, number of games, number of turns, and game winners. The `Tournament` class also implements `Serializable` and contains fields for strategy list, number of games, number of turns, and game winners. It has methods to get and set these values.

```
package org.com.Handlers;

import org.com.Constants.CommonConstants;
import org.com.Constants.CommonErrorMessages;
import org.com.Models.Tournament;
import org.com.Utils.DisplayUtil;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Set;

public class TournamentHandler implements Serializable {
    public static void processTournament(String p_command, Tournament t_tournament) throws Exception {
        String[] l_commandsArray = p_command.split(CommonConstants.COMMAND_SEPARATOR);
        for (int l_index = 1; l_index < l_commandsArray.length; l_index++) {
            String l_attribute = l_commandsArray[l_index].substring(0, 1);
            String l_value = l_commandsArray[l_index].substring(1, l_commandsArray[l_index].length());
            switch (l_attribute) {
                case CommonConstants.MAP_ATTRIBUTE:
                    t_tournament.setMapList(List.of(l_value.split(CommonConstants.MAP_SEPARATOR)));
                case CommonConstants.PLAYER_ATTRIBUTE:
                    t_tournament.setStrategyList(List.of(l_value.split(CommonConstants.PLAYER_SEPARATOR)));
                case CommonConstants.GAME_ATTRIBUTE:
                    t_tournament.setNumGames(Integer.parseInt(l_value));
                case CommonConstants.TURNS_ATTRIBUTE:
                    t_tournament.setNumTurns(Integer.parseInt(l_value));
            }
        }

        public static void processTournamentResult(Tournament p_tournament) throws Exception {
            if (p_tournament == null || p_tournament.getGameWinners().isEmpty()) {
                throw new Exception(CommonErrorMessages.TOURNAMENT_NOT_PLAYED);
            }

            int l_totalGames = p_tournament.getNumGames();
            List<String> l_columns = new ArrayList<>();
            l_columns.add("Maps");
            for (int i = 1; i <= l_totalGames; i++) {
                l_columns.add("Game " + i);
            }

            HashMap<String, ArrayList<String>> p_gameWinners = p_tournament.getGameWinners();
            String[][] l_data = new String[p_gameWinners.size()][l_columns.size()];
            Set<String> l_maps = p_gameWinners.keySet();
            int l_column = 0;
            for (String l_map : l_maps) {
                l_data[l_column][0] = l_map; // First column is the map name
                l_column++;
            }
        }
    }
}
```

```
package org.com.Models;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class Tournament implements Serializable {
    private List<String> d_mapList;
    private List<String> d_strategyList;
    private int d_numGames;
    private int d_maxTurns;
    private HashMap<String, ArrayList<String>> d_gameWinners;

    public Tournament() {
        this.d_mapList = new ArrayList<>();
        this.d_strategyList = new ArrayList<>();
        this.d_numGames = 0;
        this.d_maxTurns = 0;
        this.d_gameWinners = new HashMap<>();
    }

    public List<String> getMapList() {
        return d_mapList;
    }

    public void setMapList(List<String> p_mapList) {
        this.d_mapList = p_mapList;
    }

    public List<String> getStrategyList() {
        return d_strategyList;
    }

    public void setStrategyList(List<String> p_strategyList) {
        this.d_strategyList = p_strategyList;
    }

    public int getNumGames() {
        return d_numGames;
    }

    public void setNumGames(int p_numGames) {
        this.d_numGames = p_numGames;
    }

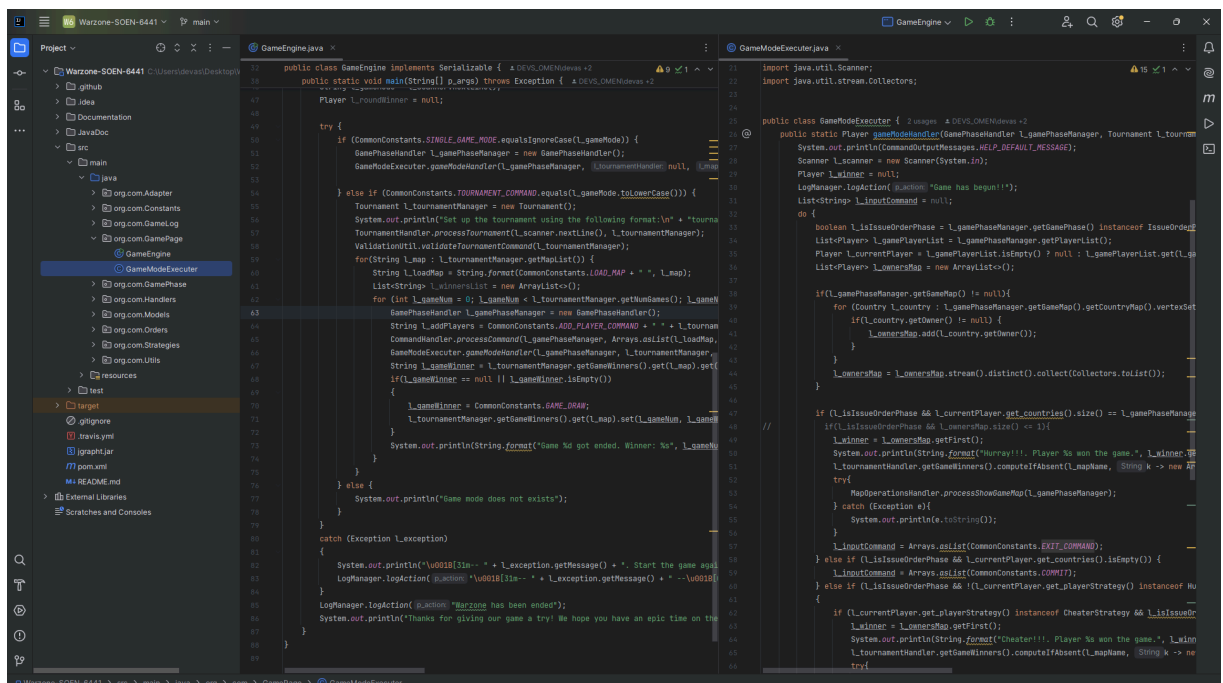
    public int getMaxTurns() {
        return d_maxTurns;
    }

    public void setMaxTurns(int p_maxTurns) {
        this.d_maxTurns = p_maxTurns;
    }

    public HashMap<String, ArrayList<String>> getGameWinners() {
        return d_gameWinners;
    }

    public void setGameWinners(HashMap<String, ArrayList<String>> p_gameWinners) {
        this.d_gameWinners = p_gameWinners;
    }
}
```

### 4. Decomposed GameEngine into GameEngine and GameModeExecutor - Either for a Single mode or Tournament mode the core logic and operations for executing the game is the same. So the GameModeExecutor has the logic to run a single game.



The screenshot shows an IDE with two files open: `GameEngine.java` and `GameModeExecutor.java`. The `GameEngine` class implements `Serializable` and contains a `main` method that takes an array of arguments. It creates a `GamePhaseManager` and a `GameModeExecutor` object, then calls `processTournament` on the `GameModeExecutor` object. The `GameModeExecutor` class implements `Serializable` and contains a `gameModeHandler` method that takes a `GamePhaseManager` and a `GameModeExecutor` object. It processes the game mode by splitting the command into attributes and values, then sets the game mode's strategy list, number of games, number of turns, and game winners.

```
package org.com;

import org.com.Constants.CommonConstants;
import org.com.Constants.CommonErrorMessages;
import org.com.Models.Tournament;
import org.com.Utils.DisplayUtil;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Set;

public class GameEngine implements Serializable {
    public static void main(String[] p_args) throws Exception {
        Player l_foundWinner = null;

        try {
            if (CommonConstants.SINGLE_GAME_MODE.equalsIgnoreCase(p_args[0])) {
                GamePhaseManager l_gamePhaseManager = new GamePhaseManager();
                GameModeExecutor l_gameModeExecutor = new GameModeExecutor();
                l_gameModeExecutor.gameModeHandler(l_gamePhaseManager, l_tournamentHandler, l_foundWinner);
            } else if (CommonConstants.TOURNAMENT_COMMAND.equalsIgnoreCase(p_args[0])) {
                System.out.println("Let us play the tournament using the following format: let + 'tournament' + 'tournamentHandler' + 'gamePhaseManager' + 'gameModeExecutor'");
                System.out.println("Validating the tournament command: " + p_args[0]);
                for (String l_map : l_tournamentHandler.getMapList()) {
                    String l_loadMap = String.format(CommonConstants.LOAD_MAP + " %s", l_map);
                    List<String> l_winnerList = new ArrayList<>();
                    for (int l_gameNum = 0; l_gameNum < l_tournamentHandler.getNumGames(); l_gameNum++) {
                        GamePhaseManager l_gamePhaseManager = new GamePhaseManager();
                        String l_players = CommonConstants.ADD_PLAYER_COMMAND + " " + l_tournamentHandler.getPlayerList();
                        GameModeExecutor l_gameModeExecutor = new GameModeExecutor();
                        l_gameModeExecutor.gameModeHandler(l_gamePhaseManager, l_tournamentHandler, l_foundWinner);
                        String l_gameMode = l_tournamentHandler.getGameWinners().get(l_map).get(l_gameNum);
                        if (l_gameMode == null || l_gameMode.isEmpty()) {
                            l_gameMode = CommonConstants.GAME_OVER;
                            l_tournamentHandler.getGameWinners().get(l_map).set(l_gameNum, l_gameMode);
                            System.out.println(String.format("Game %d has ended. Winner: %s", l_gameNum, l_gameMode));
                        } else {
                            System.out.println("Game mode does not exists");
                        }
                    }
                } catch (Exception l_exception) {
                    System.out.println("Invalid input: " + l_exception.getMessage() + ". Start the game again");
                    LogManager.logException("Invalid input: " + l_exception.getMessage() + ". Start the game again");
                }
            }
        } catch (Exception l_exception) {
            System.out.println("Invalid input: " + l_exception.getMessage() + ". Start the game again");
            LogManager.logException("Invalid input: " + l_exception.getMessage() + ". Start the game again");
        }
    }
}
```

```
package org.com;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class GameModeExecutor implements Serializable {
    private List<String> d_mapList;
    private List<String> d_strategyList;
    private int d_numGames;
    private int d_maxTurns;
    private HashMap<String, ArrayList<String>> d_gameWinners;

    public GameModeExecutor() {
        this.d_mapList = new ArrayList<>();
        this.d_strategyList = new ArrayList<>();
        this.d_numGames = 0;
        this.d_maxTurns = 0;
        this.d_gameWinners = new HashMap<>();
    }

    public List<String> getMapList() {
        return d_mapList;
    }

    public void setMapList(List<String> p_mapList) {
        this.d_mapList = p_mapList;
    }

    public List<String> getStrategyList() {
        return d_strategyList;
    }

    public void setStrategyList(List<String> p_strategyList) {
        this.d_strategyList = p_strategyList;
    }

    public int getNumGames() {
        return d_numGames;
    }

    public void setNumGames(int p_numGames) {
        this.d_numGames = p_numGames;
    }

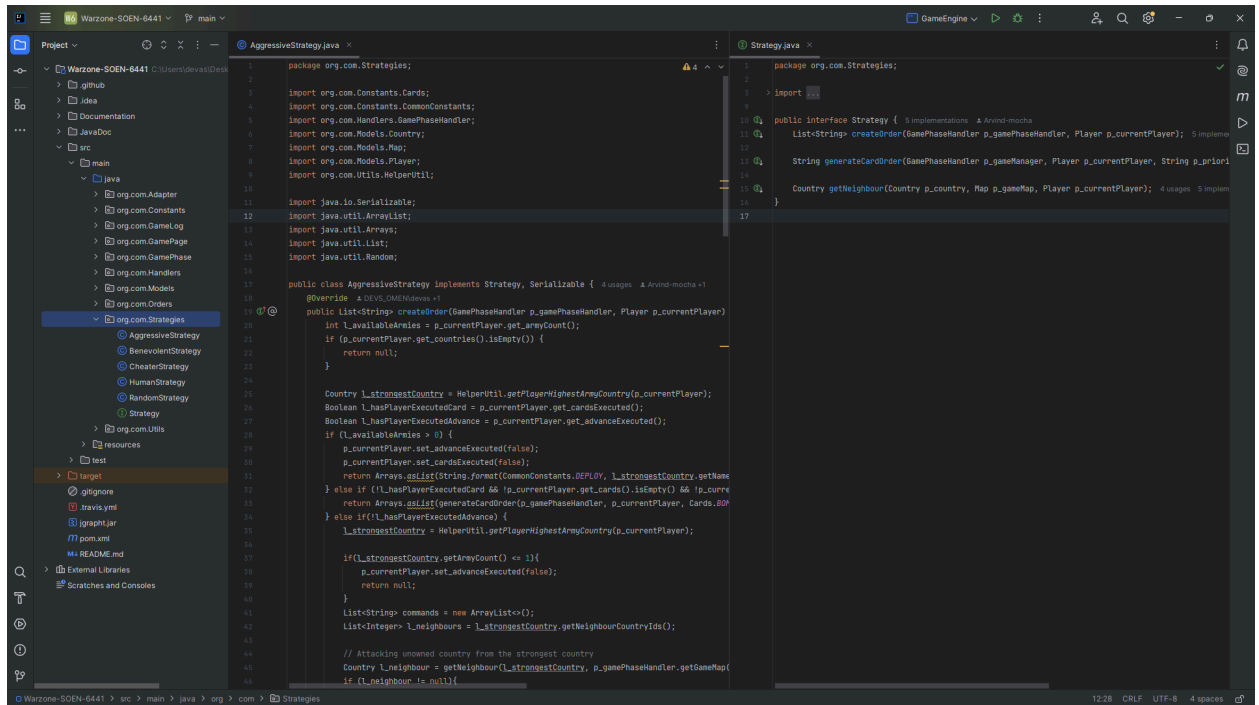
    public int getMaxTurns() {
        return d_maxTurns;
    }

    public void setMaxTurns(int p_maxTurns) {
        this.d_maxTurns = p_maxTurns;
    }

    public HashMap<String, ArrayList<String>> getGameWinners() {
        return d_gameWinners;
    }

    public void setGameWinners(HashMap<String, ArrayList<String>> p_gameWinners) {
        this.d_gameWinners = p_gameWinners;
    }
}
```

## 5. Added new Module for implementing all the computer Strategies



## Reasons for choosing the Actual Refactoring Targets

The following elements were chosen based on the aspects of code quality, maintainability and better functionality of the game in whole. This has helped in improving the internal structure of the program without breaking the external behaviour of the game observed in the initial build.

- 1. Updated all the Java classes to implement Serializable** - In the second build, the classes defined were not serializable. To save or load game phase, the phase has to be stored as objects in .ser or .dat formats. Hence, the classes were implemented as serializable in this phase.
- 2. Encapsulated functions to handle different map types in the Adapter Module**  
Initially the game was developed to handle only domination based maps. Based on

the requirements, we introduced an adapter pattern to handle conquest and domination maps. This helps us in reducing re-writing lines of code and introduces the choice of using different maps.

**3. Added a new Model for the Tournament feature** - This was done to introduce the tournament feature. Tournament feature uses different player types and automates the entire game except for “human” type players, where the commands have to be given manually.

**4. Decomposed GameEngine into GameEngine and GameModeExecuter** - Decomposed the GameEngine into 2 files to handle the type of game played. Based on the mode single or tournament, the GameModeExecuter is responsible for handling the mode of the games. This is done for the game to be run smoothly, efficiently and without any conflicts.

**5. Added new Module for implementing all the computer Strategies**

A new module was introduced to implement all computer strategies because if these were added to PlayerOperationsHandler, it would have made the codebase complicated and unreadable. This would have also made the codebase extremely difficult to understand and hard to work on for future enhancements. Hence we came up with the idea to add a new module because in case if there are any new strategies to be added, this could be done easily and efficiently to the current module by just introducing a new java file.