Group - ADYST: REFACTORING DOCUMENT

COURSE: SOEN 6441

TEAM MEMBERS:

- Akhilesh Kanbarkar (40301665)
- Disha Padsala (40324578)
- Yash Koladiya (40324106)
- Shrey Hingu (40305516)
- Taksh Rana (40303723)

Potential Refactoring Targets:

Following the completion of Build 1, several enhancements and code optimizations were essential to align with the evolving requirements of Build 2. The following refactoring actions were implemented to improve code maintainability, scalability, and readability:

- **Adapter Pattern Restructuring for Map Loading**
  Restructured the Adapter design pattern to support seamless loading and saving of both Domination and Conquest map types.

- **Strategy Pattern for Player Behavior**
  Integrated the Strategy pattern to manage diverse player behavioral strategies for flexible and extensible gameplay logic.

- **Tournament Mode Command Integration**
  Extended existing functions to support new tournament mode commands, enhancing overall gameplay flexibility.

- **LoadMap Enhancement for Conquest Support**
  Modified the LoadMap function to handle Conquest map format, ensuring compatibility across multiple map types.

- **Game Services for Save/Load Features**
  Introduced GameService components to enable saving and loading of game states, supporting session persistence.

- **Support for Single and Tournament Game Modes**
  Redesigned core gameplay to accommodate both standard single-player and extended tournament modes.

- **JavaDoc Enhancement**
  Documented private fields and methods using JavaDoc annotations

- **Command Syntax Validation**
  Implemented rigorous input validation to prevent syntactic errors in user-entered commands.

- **Logger Integration for New Features**
  Implemented logging for newly added functionalities and remaining methods from Build 2 to ensure traceability and facilitate debugging.

- **Observer Pattern Modularization**
  Refactored and modularized the Observer pattern by relocating all related components into the view directory to improve maintainability.

- **Refactoring of Existing Test Cases**
  Updated the existing test cases to match the new structure and logic introduced in Build 2.

- **Expanded Test Coverage**
  Added new unit tests to validate the functionality of newly introduced and existing features.

- **Console Output Enhancement**
  Improved the clarity and structure of console outputs to enhance user experience and gameplay readability.

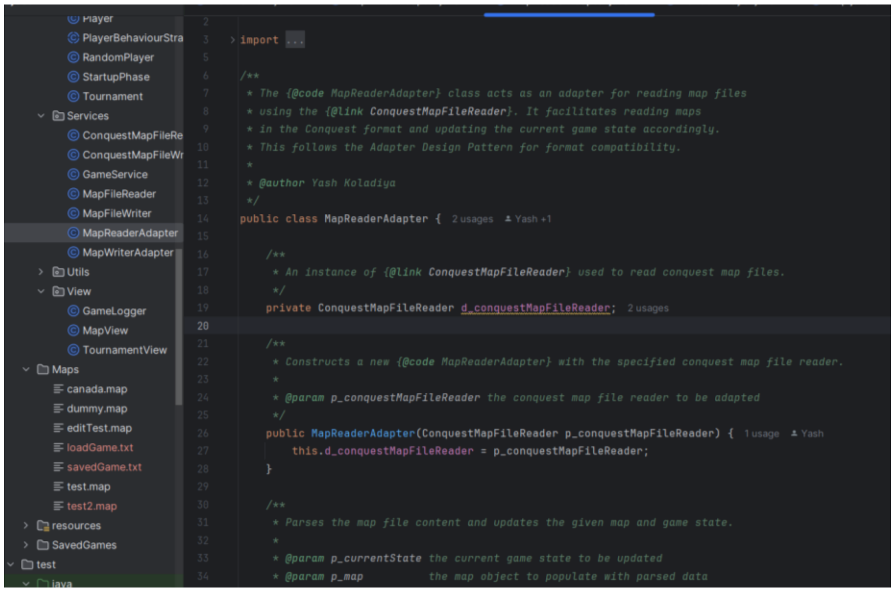
- **Comprehensive Test Suite Organization**
  Developed a structured test suite that includes tests across all major modules and folders for streamlined quality assurance.

Actual Refactoring Targets:

1) **Utilization of Adapter Pattern for Map File Handling**
   Before Refactoring (Build 2):
   In Build 2, the application was limited to supporting only the Domination map file format for reading and writing map data. The logic for handling map operations was directly embedded into the Map component. This tight coupling restricted extensibility—any support for new map formats like Conquest would require intrusive changes to the existing codebase. The absence of a common interface or abstraction made the system inflexible and hard to scale for multiple map types.

   After Refactoring (Build 3): Implementing the Adapter Pattern
   In Build 3, we introduced the Adapter Pattern to support both Domination and Conquest map formats seamlessly. A common interface (MapReaderAdapter and MapWriterAdaptor) was defined, which handle format-specific reading and writing logic.

During runtime, the system now prompts the user to select the desired map format. Based on the selected file type, the application dynamically instantiates the appropriate adapter. This decouples the format-specific logic from the core game engine and promotes open-closed design principles—enabling the system to support additional formats in the future with minimal changes. The result is a flexible, extensible, and maintainable architecture for map file handling.



```java
import ...

/**
 * The {@code MapReaderAdapter} class acts as an adapter for reading map files
 * using the {@link ConquestMapFileReader}. It facilitates reading maps
 * in the Conquest format and updating the current game state accordingly.
 * This follows the Adapter Design Pattern for format compatibility.
 *
 * @author Yash Koladiya
 */
public class MapReaderAdapter {  2 usages  Yash +1

    /**
     * An instance of {@link ConquestMapFileReader} used to read conquest map files.
     */
    private ConquestMapFileReader d_conquestMapFileReader;  2 usages

    /**
     * Constructs a new {@code MapReaderAdapter} with the specified conquest map file reader.
     *
     * @param p_conquestMapFileReader the conquest map file reader to be adapted
     */
    public MapReaderAdapter(ConquestMapFileReader p_conquestMapFileReader) {  1 usage  Yash
        this.d_conquestMapFileReader = p_conquestMapFileReader;
    }

    /**
     * Parses the map file content and updates the given map and game state.
     *
     * @param p_currentState the current game state to be updated
     * @param p_map          the map object to populate with parsed data
```

MapReaderAdaptor class

2) **Implementation of Strategy Pattern for Player Behaviors**

Before Refactoring (Build 2):

In Build 2, the game supported only a human player type, and the issueOrder() method in the Player class was implemented with direct logic requiring user interaction for decision-making. There was no separation of player behavior or strategy, and the code lacked flexibility.

After Refactoring (Build 3): Implementing the Strategy Pattern

In Build 3, we refactored the Player class by introducing the Strategy Pattern to support multiple player behaviors through interchangeable strategy objects. An abstract PlayerStrategy interface was created, with concrete implementations for each type of player:

HumanPlayerStrategy: Requires user interaction to make decisions.
AggressivePlayerStrategy: Focuses on deploying on the strongest country and attacking neighbors to consolidate forces.
BenevolentPlayerStrategy: Deploys on the weakest country, avoids attacks, and reinforces vulnerable positions.

RandomPlayerStrategy: Executes deployment, attacks, and movements in a random manner.
CheaterPlayerStrategy: Instantly conquers all neighboring enemy territories and reinforces borders.

The Player class now delegates the execution of issueOrder() to the currently assigned strategy. This decouples player behavior from the player logic, enabling dynamic assignment of behaviors and supporting AI-controlled gameplay. The result is a more modular, extensible, and testable system that allows for scalable player strategy development.

3) **Addition of Tournament Mode for Automated Multi-Map Gameplay**
Before Refactoring (Build 2):
In Build 2, the game supported only single-game mode with human player interaction. All player actions, including map selection, issuing orders, and game progression, required manual input. There was no concept of a tournament mode, and the system lacked automation to run multiple games across different configurations. This limited the game's usability for testing AI strategies or simulating varied scenarios across multiple maps.

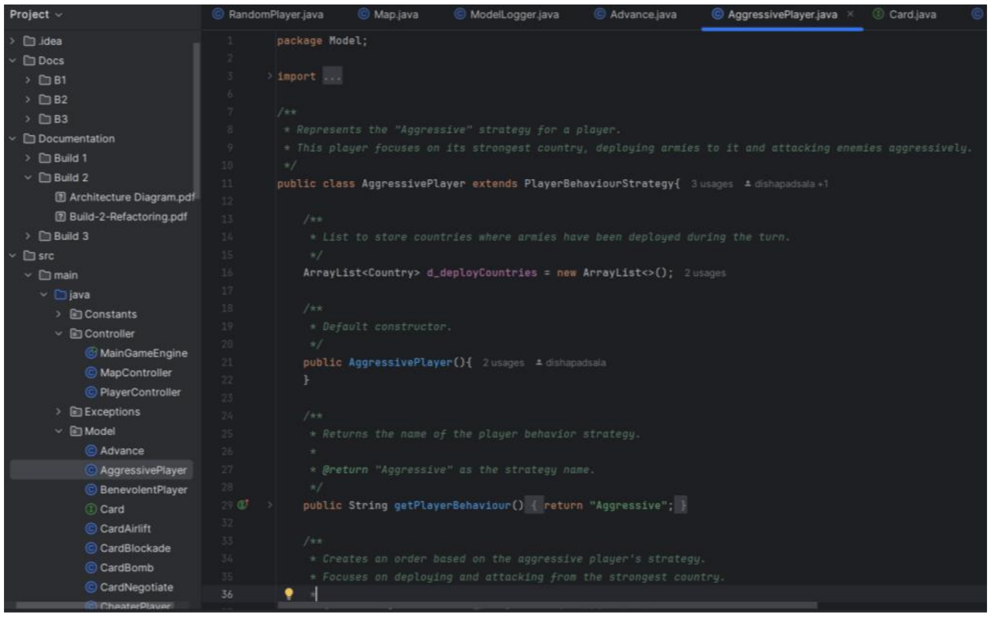
After Refactoring (Build 3): Implementing Tournament Mode
In Build 3, we introduced a new Tournament Mode to support automated simulation of multiple games across different maps and player strategies. This was achieved by implementing a Tournament class in the model and a TournamentView class in the view module to manage user input and output display.

In this mode, the player can configure the following parameters:

1 to 5 different maps to be played.
2 to 4 computer player strategies (Aggressive, Benevolent, Random, Cheater).
1 to 5 games per map.
10 to 50 turns per game.

Once configured, the tournament runs automatically without requiring user interaction. Each game progresses turn-by-turn until a player wins or the specified number of turns (D) is reached. If no player wins within D turns, the game is declared a draw.

This feature enhances the game's scalability, supports automated testing of computer strategies, and allows players to analyze performance across a variety of scenarios—bringing a powerful and structured simulation capability to the game engine.



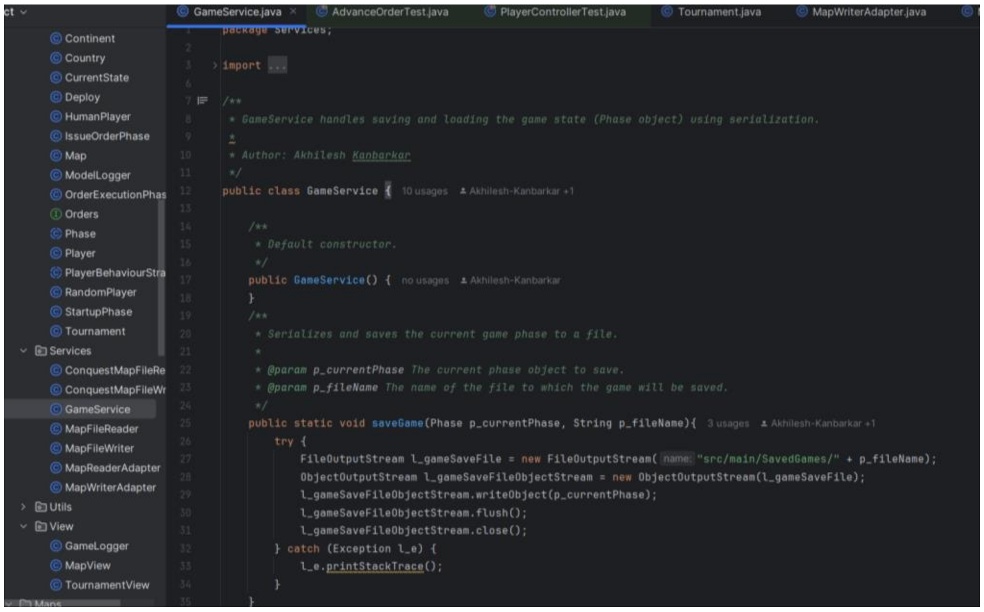
```java
package Model;

import ...

/**
 * Represents the "Aggressive" strategy for a player.
 * This player focuses on its strongest country, deploying armies to it and attacking enemies aggressively.
 */
public class AggressivePlayer extends PlayerBehaviourStrategy{

    /**
     * List to store countries where armies have been deployed during the turn.
     */
    ArrayList<Country> d_deployCountries = new ArrayList<>();

    /**
     * Default constructor.
     */
    public AggressivePlayer(){
    }

    /**
     * Returns the name of the player behavior strategy.
     *
     * @return "Aggressive" as the strategy name.
     */
    public String getPlayerBehaviour() { return "Aggressive"; }

    /**
     * Creates an order based on the aggressive player's strategy.
     * Focuses on deploying and attacking from the strongest country.
     *
```

Aggressive Player class

4) **Game Services for Save/Load Features**
Before Refactoring (Build 2):
In Build 2, the game lacked the ability to save and load progress. Players were forced to either complete the game or quit without any option to resume later. This limitation hindered flexibility and user experience, especially in longer gameplay sessions. The game had no mechanism for persisting game state, meaning that any interruption would result in the loss of progress.

After Refactoring (Build 3): Implementing Save/Load Game Functionality
In Build 3, we introduced the GameService components to enable save and load functionality for game states, allowing players to pause and resume their sessions at any point. To implement this, we leveraged the existing classes that implement Serializable for the game's objects. The game now automatically stores its current state to a file based on the ongoing phase.

Whenever the user issues the savegame command, the game serializes the necessary game state information and writes it to a file. On the other hand, the loadgame command loads the game state from the file and continues gameplay from the point where it was last saved. This feature greatly enhances the user experience by allowing players to pick up their progress at any time, ensuring session persistence and flexibility.



```java
package Services;

import ...

/**
 * GameService handles saving and loading the game state (Phase object) using serialization.
 *
 * Author: Akhilesh Kanbarkar
 */
public class GameService {

    /**
     * Default constructor.
     */
    public GameService() {
    }
    /**
     * Serializes and saves the current game phase to a file.
     *
     * @param p_currentPhase The current phase object to save.
     * @param p_fileName The name of the file to which the game will be saved.
     */
    public static void saveGame(Phase p_currentPhase, String p_fileName){
        try {
            FileOutputStream l_gameSaveFile = new FileOutputStream( name: "src/main/SavedGames/" + p_fileName);
            ObjectOutputStream l_gameSaveFileObjectStream = new ObjectOutputStream(l_gameSaveFile);
            l_gameSaveFileObjectStream.writeObject(p_currentPhase);
            l_gameSaveFileObjectStream.flush();
            l_gameSaveFileObjectStream.close();
        } catch (Exception l_e) {
            l_e.printStackTrace();
        }
    }
```

Game service class

5) **Test Cases for Newly Added Components in Build 3**
Before Refactoring (Build 2):
In Build 2, test cases were implemented for core gameplay features such as player actions, map loading, and issue orders. While functional, the test coverage did not extend to new features like tournament mode, save/load functionality, or the strategy pattern. This created a gap in testing for newly added components, leaving certain parts of the game untested and potentially unstable after refactoring.

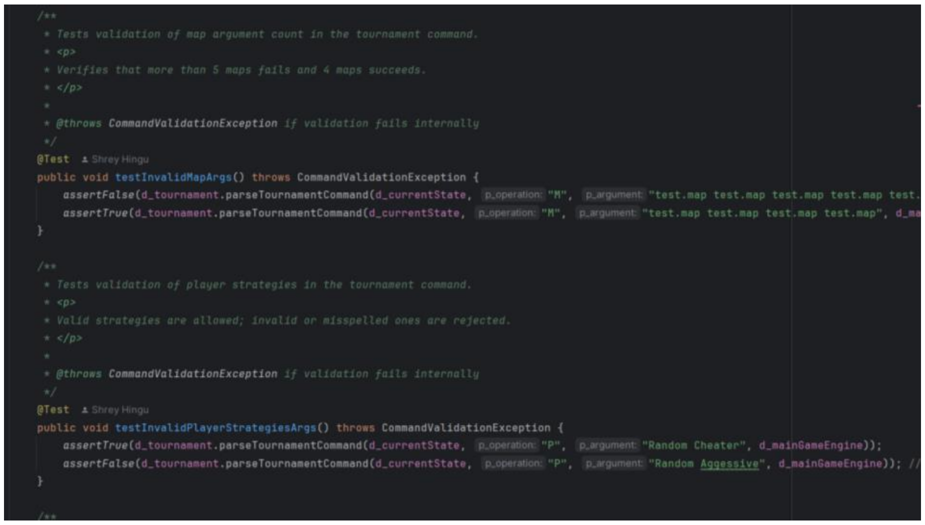
After Refactoring (Build 3): Extending Test Cases to Cover New Features
In Build 3, we expanded the test suite to include comprehensive test cases for the newly implemented features. To meet the criteria of a minimum of 20 test cases, we added tests specifically for the following new components:

Tournament Mode: Test cases were added to verify that the tournament setup handles multiple maps, strategies, games, and turn counts correctly. Edge cases were tested, such as ensuring tournaments with 1 map and 2 AI strategies are processed without errors.

Save/Load Functionality: We wrote tests to ensure that game states are correctly saved and loaded, verifying that the game can resume from any saved point and that data integrity is maintained after reloading. Edge cases like saving at the end of a game or during mid-phase were also tested.

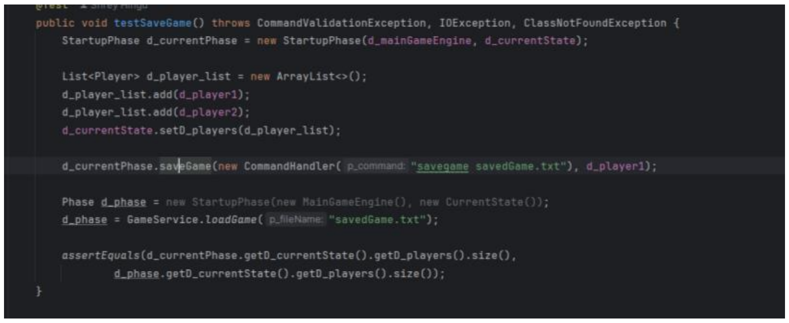
Strategy Pattern: Tests were added to confirm that each player strategy (Human, Aggressive, Benevolent, Random, and Cheater) performs as expected during the game. We validated that the correct strategy was applied and that each player behavior aligned with its respective strategy.

Each test case is thoroughly documented with Javadoc to provide clarity on the purpose of the test, the expected behavior, and edge case handling. This expansion in test coverage ensures that all new functionality in Build 3 is thoroughly validated and integrates seamlessly with existing features, improving the stability and reliability of the application.



```java
/**
 * Tests validation of map argument count in the tournament command.
 * <p>
 * Verifies that more than 5 maps fails and 4 maps succeeds.
 * </p>
 *
 * @throws CommandValidationException if validation fails internally
 */
@Test  Shrey Hingu
public void testInvalidMapArgs() throws CommandValidationException {
    assertFalse(d_tournament.parseTournamentCommand(d_currentState, p_operation: "M", p_argument: "test.map test.map test.map test.map test.
    assertTrue(d_tournament.parseTournamentCommand(d_currentState, p_operation: "M", p_argument: "test.map test.map test.map test.map", d_ma
}

/**
 * Tests validation of player strategies in the tournament command.
 * <p>
 * Valid strategies are allowed; invalid or misspelled ones are rejected.
 * </p>
 *
 * @throws CommandValidationException if validation fails internally
 */
@Test  Shrey Hingu
public void testInvalidPlayerStrategiesArgs() throws CommandValidationException {
    assertTrue(d_tournament.parseTournamentCommand(d_currentState, p_operation: "P", p_argument: "Random Cheater", d_mainGameEngine));
    assertFalse(d_tournament.parseTournamentCommand(d_currentState, p_operation: "P", p_argument: "Random Aggessive", d_mainGameEngine)); //
}

/**
```

Tournament Test



```java
public void testSaveGame() throws CommandValidationException, IOException, ClassNotFoundException {
    StartupPhase d_currentPhase = new StartupPhase(d_mainGameEngine, d_currentState);

    List<Player> d_player_list = new ArrayList<>();
    d_player_list.add(d_player1);
    d_player_list.add(d_player2);
    d_currentState.setD_players(d_player_list);

    d_currentPhase.saveGame(new CommandHandler( p_command: "savegame savedGame.txt"), d_player1);

    Phase d_phase = new StartupPhase(new MainGameEngine(), new CurrentState());
    d_phase = GameService.loadGame( p_fileName: "savedGame.txt");

    assertEquals(d_currentPhase.getD_currentState().getD_players().size(),
            d_phase.getD_currentState().getD_players().size());
}
```

SaveGame test