Concordia University

Department of Computer Science and Software Engineering

Advanced Programming Practices

SOEN 6441 - Fall 2023

Group Name: W4
Submitted to: Prof. Amin Ranj Bar

Members:
Sneh Satishkumar Patel - 40264053
Romit Manishkumar Patel - 40273551
Yash Subhashbhai Chhelaiya - 40257682
Rushi Hasubhai Donga - 40269583
Abdul Sameer Jani Syed - 40272408
Mayank Nilkanth Parmar - 40269385

Risk Computer Game

A developed program which is compatible with the rules and map files and the command-line play of the “Warzone” version of Risk, which can be found at: https://www.warzone.com/.

About Risk Game

- A Warzone game setup consists of a connected graph map representing a world map, where each node is a country and each edge represents adjacency between countries.
- Two or more players can play by placing armies on countries they own, from which they can attack adjacent countries to conquer them.
- The objective of the game is to conquer all countries on the map.

Goal of Build 2

- RiskGame Build 2's goal is to to refactor the code to effectively use the State pattern and different phases of the application, including map editing and gameplay, are encapsulated in separate states.
- This build specifies the operational version of the final project demonstrating a subset of the capacity of the system.
- It also covers implementation of Comprehensive Game Logging using the Observer pattern and adopted the Command pattern for managing and executing game orders.
- Building upon the foundation established in Build 1, refactoring the code to incorporate the Command pattern for improved game order management. This refactoring is in alignment with the project's strong focus on unit testing and the integration of CI/CD pipelines.

CODING STANDARDS

Code Layout

- **Indentation:**
  - Used 4 spaces for indentation.
  - The body of loop and conditional statements are indented in relation to their starting lines.
  - Body of a function is appropriately indented beneath its header.
- **Line Spacing:**
  - Blank lines serve the purpose of creating separation within code.
  - They are inserted between class declarations, methods, and significant segments within complex functions to enhance code readability.
- **Positioning of curly braces:**
  - The position of curly braces involves placing an opening curly brace immediately after the statement preceding it, thereby reducing the length of the code.
- **Spacing around operators and operands:**
  - Inserting spaces before and after operators to enhance the legibility of code.

MapHelper.java

```java
/**
 * Helper method to add/remove a new continent to the game map.
 *
 * @param p_mapFileName Name of map file.
 * @return Edited game map
 */

public GameMap editMap(String p_mapFileName) {
    String l_filePath = Constant.MAP_PATH + p_mapFileName;
    this.d_gameMap = new GameMap(p_mapFileName);
    File l_file = new File(l_filePath);

    if (l_file.exists()) {
        System.out.println(p_mapFileName + " map file exists. You can edit it.");
        this.readMap(l_filePath);
    } else {
        System.out.println(p_mapFileName + " does not exist.");
        System.out.println("Creating a new Map named: " + p_mapFileName);

        this.d_gameMap = new GameMap(p_mapFileName);
    }
    return this.d_gameMap;
}
```

Guidelines for Naming

- **Classes:**
  - Class names follow the convention of being written in UpperCamelCase, also known as Pascal Case.
  - Example : class MyClassName { }
- **Method Parameters, Member functions and Data Members:**
  - These are written in lowerCamelCase.
  - For method parameters, we use **‘p_’** as prefix and for data members, we use ‘**d_’** as prefix.
  - Example : int d_mapIndex = 1, public GameMap loadMap(String p_mapFileName)
- **Local Variables:**
  - Adheres to lower camelCase with the prefix **‘l_’** to indicate that it is a local variable.
  - Example : String l_line;
- **Constants:**
  - Uppercase letters with underscores.
  - Example : public static final String MAP_PATH = "src/main/resource/map/";

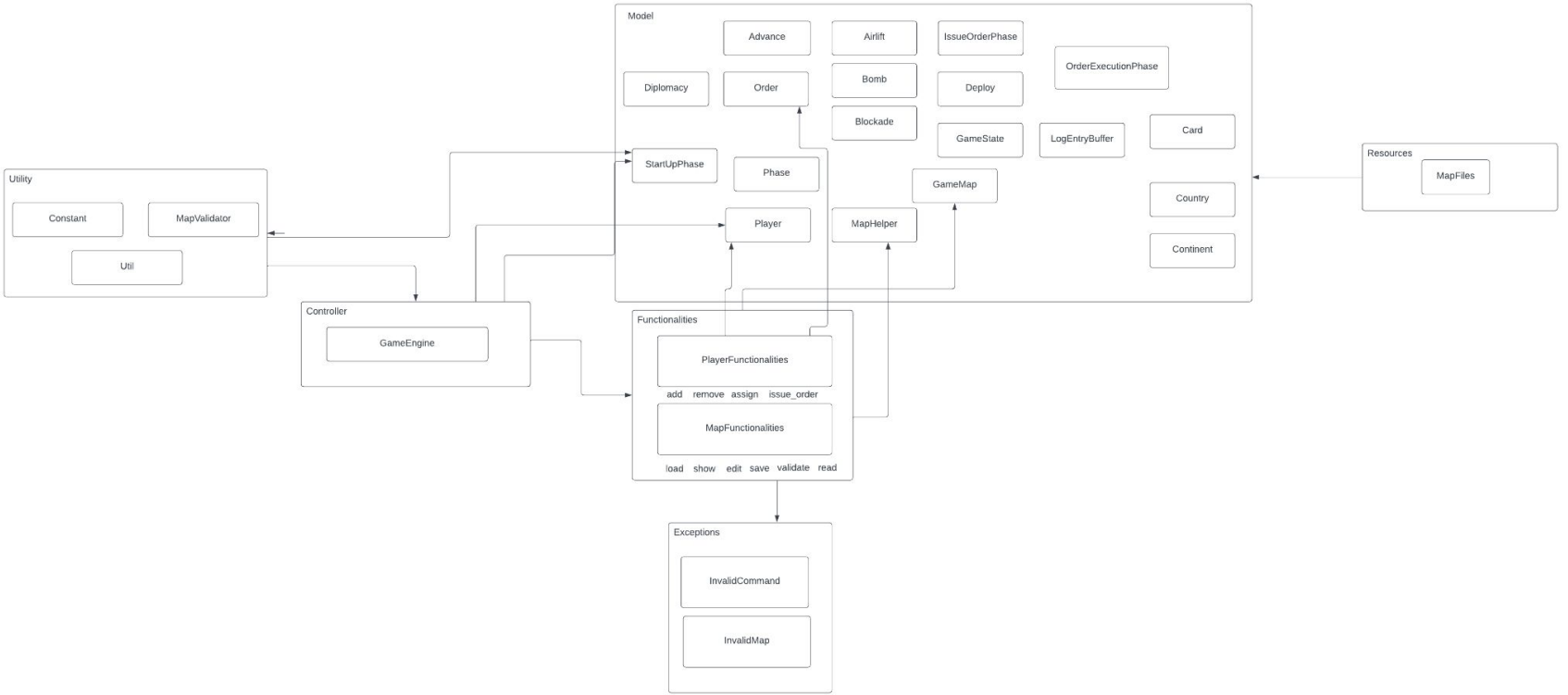
Guidelines for Writing Comments

- **Code without any commented-out sections:**
  - Excess commented code is removed to enhance clarity and readability.
- **Documentation comments (JavaDoc):**
  - For each class, data member, and member function, there will be JavaDoc documentation provided above them.
- **In Complex Methods :**
  - In lengthy and complex methods, explanatory content is inserted to improve code comprehension.

StartUpPhase.java

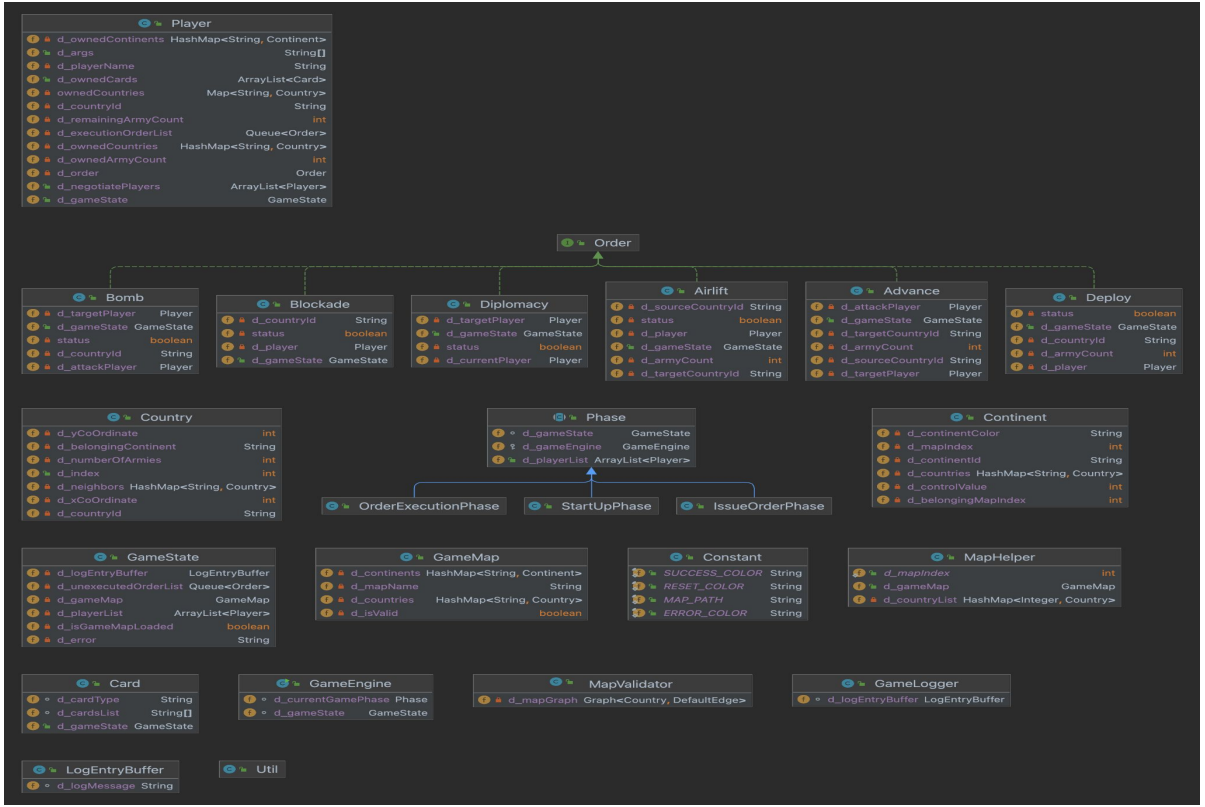
```java
/**
 * Getter method for player list.
 *
 * @return Returns list of game player.
 */
public ArrayList<Player> getPlayerList() {
    return this.d_playerList;
}

/**
 * Setter method for game phase.
 *
 * @param p_gamePhase GamePhase to set
 */
public void setGamePhase(Phase p_gamePhase) {
    this.d_gamePhase = p_gamePhase;
}
```

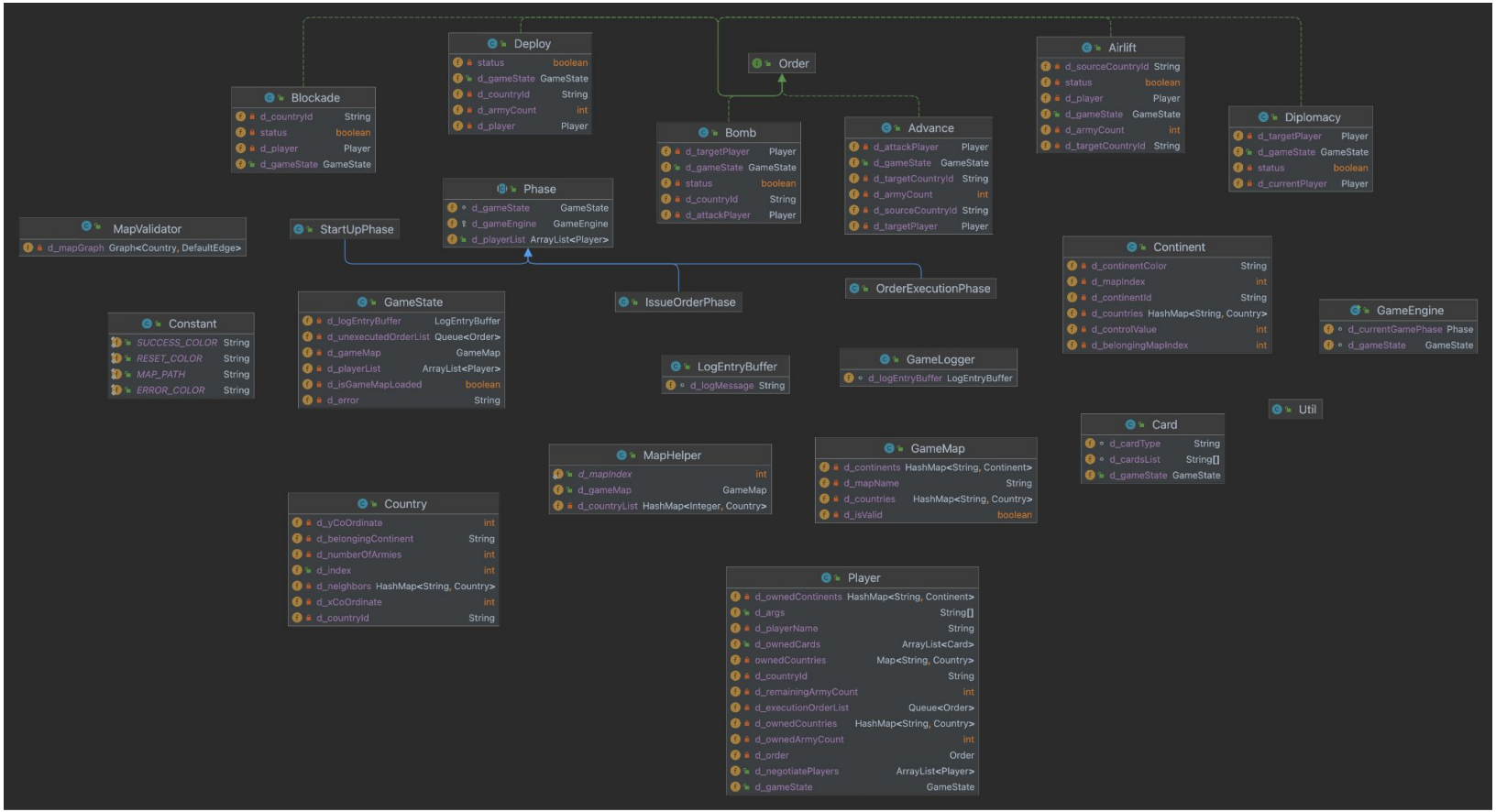
Architecture Diagram



Class Diagram -Need to edit

Functional Diagram - Need to edit




Deploy
status boolean
d_gameState GameState
d_countryId String
d_armyCount int
d_player Player
Order
Airlift
d_sourceCountryId String
status boolean
d_player Player
d_gameState GameState
d_armyCount int
d_targetCountryId String
Blockade
d_countryId String
status boolean
d_player Player
d_gameState GameState
Bomb
d_targetPlayer Player
d_gameState GameState
status boolean
d_countryId String
d_attackPlayer Player
Advance
d_attackPlayer Player
d_gameState GameState
d_targetCountryId String
d_armyCount int
d_sourceCountryId String
d_targetPlayer Player
Diplomacy
d_targetPlayer Player
d_gameState GameState
status boolean
d_currentPlayer Player
Phase
d_gameState GameState
d_gameEngine GameEngine
d_playerList ArrayList<Player>
MapValidator
d_mapGraph Graph<Country, DefaultEdge>
StartUpPhase
IssueOrderPhase
OrderExecutionPhase
Continent
d_continentColor String
d_mapIndex int
d_continentId String
d_countries HashMap<String, Country>
d_controlValue int
d_belongingMapIndex int
GameEngine
d_currentGamePhase Phase
d_gameState GameState
Constant
SUCCESS_COLOR String
RESET_COLOR String
MAP_PATH String
ERROR_COLOR String
GameState
d_logEntryBuffer LogEntryBuffer
d_unexecutedOrderList Queue<Order>
d_gameMap GameMap
d_playerList ArrayList<Player>
d_isGameMapLoaded boolean
d_error String
LogEntryBuffer
d_logMessage String
GameLogger
d_logEntryBuffer LogEntryBuffer
Util
Card
d_cardType String
d_cardsList String[]
d_gameState GameState
MapHelper
d_mapIndex int
d_gameMap GameMap
d_countryList HashMap<Integer, Country>
GameMap
d_continents HashMap<String, Continent>
d_mapName String
d_countries HashMap<String, Country>
d_isValid boolean
Country
d_yCoOrdinate int
d_belongingContinent String
d_numberOfArmies int
d_index int
d_neighbors HashMap<String, Country>
d_xCoOrdinate int
d_countryId String
Player
d_ownedContinents HashMap<String, Continent>
d_args String[]
d_playerName String
d_ownedCards ArrayList<Card>
ownedCountries Map<String, Country>
d_countryId String
d_remainingArmyCount int
d_executionOrderList Queue<Order>
d_ownedCountries HashMap<String, Country>
d_ownedArmyCount int
d_order Order
d_negotiatePlayers ArrayList<Player>
d_gameState GameState

Implementation Highlights

- **Enhanced Gameplay:** By providing diverse gameplay phases and order types, we aim to offer players an engaging and dynamic gaming experience.
- **Card System:** Our project features a card system where players earn cards for conquering territories. Cards are used strategically to influence the game, adding depth and excitement to the gameplay.
- **Game Logging:** The Observer pattern is employed for game logging. Every action, from command execution to order issuance and execution, is meticulously recorded. This transparent game log aids in understanding game events and decisions.
- **GameEngine as Client**: we Extended the capabilities of the GameEngine class as the Client, continuing to leverage the foundation set in Build 1 to execute orders and coordinate their execution.

Implementation Highlights

- **Player as Invoker:** we Enhanced the Player class to act as the Invoker, building upon its initial design from Build 1 for order creation and management.
- **Error Handling:** we Implemented error handling mechanisms to address potential issues that may arise during execution or management.
- **Integration with Game Phases:** the refactored order management system smoothly integrates with different game phases, such as map editing, gameplay, and the startup phase.
- **Testing and Validation:** we developed a comprehensive test cases to validate the Command pattern's functionality for game management.
- **Structured Testing Framework:** We've adopted a structured testing framework, specifically **JUnit**, which provides a systematic approach to organizing and executing tests. This framework enhances our testing efficiency and makes it easier to manage and maintain our test suites.

Refactoring

- In Refactoring, we restructured and improved the existing code without changing its external behavior, And it became a crucial step after the completion of Project Build 1.
- **Identifying Potential Refactoring Targets: I**dentifying potential refactoring targets within the existing codebase was a methodical process that aimed to enhance code quality and maintainability. The following steps were undertaken: Comprehensive Code Review, Collaborative Approach, Code Complexity,Error-Prone Areas.
- **Integration with Existing Code:** we ensure that seamless integration of design patterns and refactored components with the existing codebase.And we maintained cohesion between different modules while improving the code's structure and readability.

Refactoring

- **Design Pattern Integration:** we Integrated the State pattern to manage different phases of the application more effectively. This architectural change enhances code modularity and allows for a more flexible transition between phases. We Implemented the Command pattern to handle game orders. The Order class now acts as a Command, the Player as the Invoker, and the GameEngine as the Client, providing a structured approach to order execution and management.
- **Quality and Maintenance:** Focus on code quality by adhering to coding conventions and best practices. Maintain consistent naming conventions, code layout, and commenting standards to enhance code readability and maintainability.
- **Observer Pattern for Game Logging**: we Incorporated the Observer pattern to create a comprehensive game logging system. The LogEntryBuffer class serves as an Observable, and the Observer writes log entries to a file whenever changes occur. This approach ensures transparency and provides a clear record of game actions.

Testing and Quality Assurance

- **Comprehensive Unit Testing:** Unit testing plays a pivotal role in ensuring the reliability and correctness of our project. It involves subjecting individual code components, modules, and classes to a battery of tests to verify their functionality and behavior.
- **Validation of Functional Requirements:** Our unit tests serve as concrete proof that we've met the project's functional requirements. We can point to specific test cases and results that demonstrate how our code fulfills the specified criteria. For instance, we can showcase tests that validate map editing, gameplay, and startup phases.
- **Quality Assurance Practices:** We've also embraced quality assurance practices during development. These practices include code reviews, pair programming, and adherence to coding conventions. Consistent coding standards contribute to code quality and maintainability.
- **Continuous Integration and Deployment (CI/CD):** Our project is future-ready, as we've considered integrating it into a CI/CD pipeline. These processes will automate the testing and deployment of our software, ensuring that changes are thoroughly validated and quickly deployed, streamlining our development workflows.
- In summary, comprehensive unit testing is the cornerstone of our quality assurance efforts, providing us with the confidence that our project meets its functional requirements and is well-prepared for future development phases and deployments.

Conclusion

- **Key Achievements:** Throughout the course of this project, we've achieved several key milestones that have significantly enhanced the quality and functionality of our software:
  - **Design Pattern Implementation:** One of our primary achievements is the successful implementation of key design patterns. The adoption of the State pattern has greatly improved the management of different game phases, enhancing code modularity and flexibility. Additionally, the incorporation of the Command pattern for managing game orders has provided a structured and extensible approach to order creation and execution.
  - **Observer Pattern for Transparent Game Logging:** We've successfully integrated the Observer pattern to create a comprehensive game logging system. This innovative approach ensures that every action taken during the game, whether it's a command execution or an order issue, is meticulously recorded. This transparency enhances the user experience and provides a clear record of all game events.

Conclusion

- **Challenges:** One of the challenges we encountered was the intricacy of implementing design patterns. Integrating the State and Command patterns required careful planning and restructuring of our existing codebase. The complexity of managing game phases and orders called for a deep understanding of these patterns. And The process of refactoring code to incorporate design patterns, while essential for long-term maintainability, was not without its challenges. Ensuring backward compatibility with existing components and avoiding disruption to the project's functionality demanded a systematic approach.

In the end, We're grateful for the vibrant course community that has fostered a dynamic and collaborative learning environment. The exchange of ideas and experiences with fellow students has been invaluable in shaping our approach to this project.

References

Warzone game

Domination game maps

As specified in the Project Build 2 document

Thank You