SOEN 6441 - Advanced Programming Practices

Winter 2024

Warzone Risk Game

Team 11

Submitted by

Tania Sanjid (40255010)
Tanzina Nasrin (40235506)
Khandaker Rifah Tasnia (40276078)
Masum Newaz (40292704)
Shehzar Aurangzeb Abbasi (40291795)

Architectural Design Document for Warzone Risk Game

1. Introduction

This document outlines the architectural design for the Warzone game project, a strategy game inspired by the classic board game Warzone Risk. The project emphasizes flexibility, maintainability, and scalability, leveraging the Model-View-Controller (MVC) architecture while incorporating additional components for enhanced functionality. The design facilitates easy updates, scalable development, and efficient maintenance processes.

2. System Overview

The Warzone Risk game allows players to engage in strategic territorial conquests. It features a dynamic map editor for creating, modifying, and playing on custom maps. Core gameplay mechanics include deploying armies, attacking territories, and fortifying positions to achieve global domination. The game supports single-player and multiplayer modes, accommodating various gameplay styles and strategies.

3. High-Level Architecture

3.1 Model-View-Controller (MVC) Pattern

Model: Manages game data and rules, including game map, territories, players, and game state. It encapsulates the domain logic and data structures such as Continent, Country, Map, Player, and Order.

View: Handles data presentation to the user, updating the display based on game state changes. It includes text-based interfaces.

Controller: Serves as an intermediary between Model and View, processing user input, applying game rules, and updating the model.

3.2 Additional Components

Adapters: Implements the Adapter pattern through the FileAdapter class, offering flexible file operations.

Constants: Centralizes game constants for easy management.

Utilities: Provides shared functionalities across the application for code reusability and efficiency

3.3 Architectural Diagram

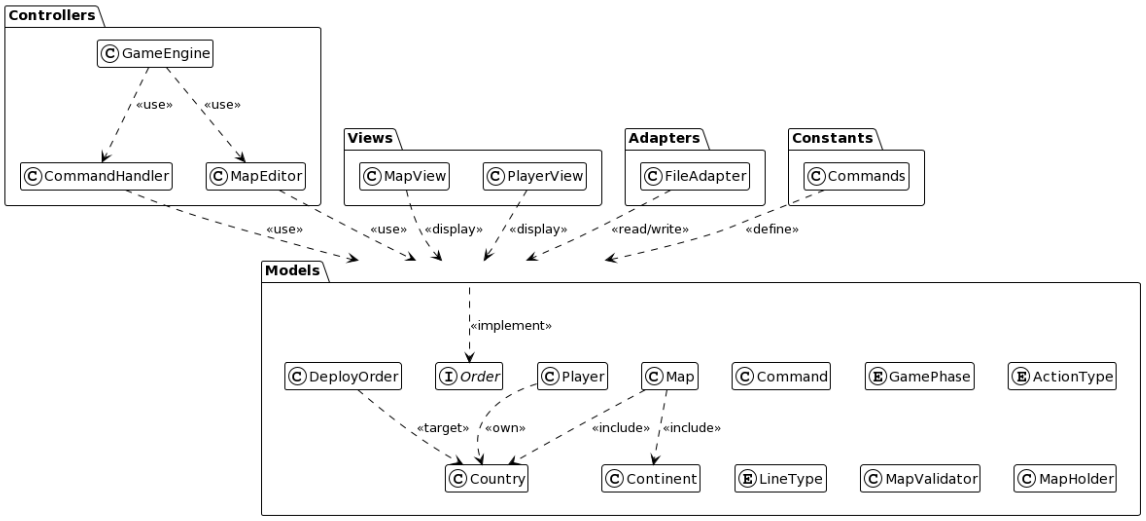

Fig 1: Architectural Diagram of Warzone

4. Architectural Design

The application's architecture, centered around the MVC pattern, supports a clear separation of concerns and modular development. This structure ensures that each component of the application has a defined role, facilitating easier maintenance and scalable development.

4.1 Class Diagram

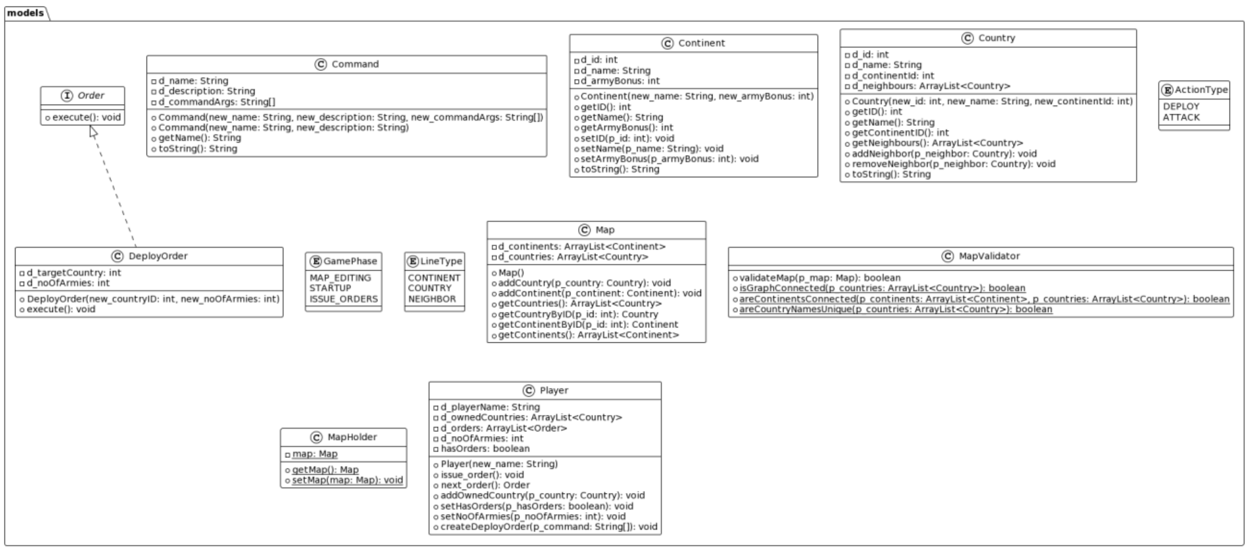
models
Order
execute(): void
Command
d_name: String
d_description: String
d_commandArgs: String[]
Command(new_name: String, new_description: String, new_commandArgs: String[])
Command(new_name: String, new_description: String)
getName(): String
toString(): String
Continent
d_id: int
d_name: String
d_armyBonus: int
Continent(new_name: String, new_armyBonus: int)
getID(): int
getName(): String
getArmyBonus(): int
setID(p_id: int): void
setName(p_name: String): void
setArmyBonus(p_armyBonus: int): void
toString(): String
Country
d_id: int
d_name: String
d_continentId: int
d_neighbours: ArrayList<Country>
Country(new_id: int, new_name: String, new_continentId: int)
getID(): int
getName(): String
getContinentID(): int
getNeighbours(): ArrayList<Country>
addNeighbor(p_neighbor: Country): void
removeNeighbor(p_neighbor: Country): void
toString(): String
ActionType
DEPLOY
ATTACK
DeployOrder
d_targetCountry: int
d_noOfArmies: int
DeployOrder(new_countryID: int, new_noOfArmies: int)
execute(): void
GamePhase
MAP_EDITING
STARTUP
ISSUE_ORDERS
LineType
CONTINENT
COUNTRY
NEIGHBOR
Map
d_continents: ArrayList<Continent>
d_countries: ArrayList<Country>
Map()
addCountry(p_country: Country): void
addContinent(p_continent: Continent): void
getCountries(): ArrayList<Country>
getCountryByID(p_id: int): Country
getContinentByID(p_id: int): Continent
getContinents(): ArrayList<Continent>
MapValidator
validateMap(p_map: Map): boolean
isGraphConnected(p_countries: ArrayList<Country>): boolean
areContinentsConnected(p_continents: ArrayList<Continent>, p_countries: ArrayList<Country>): boolean
areCountryNamesUnique(p_countries: ArrayList<Country>): boolean
MapHolder
map: Map
getMap(): Map
setMap(map: Map): void
Player
d_playerName: String
d_ownedCountries: ArrayList<Country>
d_orders: ArrayList<Order>
d_noOfArmies: int
hasOrders: boolean
Player(new_name: String)
issue_order(): void
next_order(): Order
addOwnedCountry(p_country: Country): void
setHasOrders(p_hasOrders: boolean): void
setNoOfArmies(p_noOfArmies: int): void
createDeployOrder(p_command: String[]): void

Fig 4.1.1: Models Package Overview

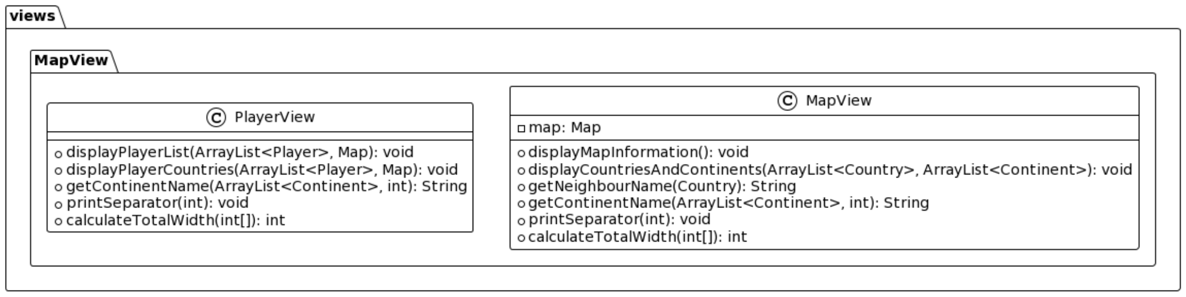
views
MapView
PlayerView
displayPlayerList(ArrayList<Player>, Map): void
displayPlayerCountries(ArrayList<Player>, Map): void
getContinentName(ArrayList<Continent>, int): String
printSeparator(int): void
calculateTotalWidth(int[]): int
MapView
map: Map
displayMapInformation(): void
displayCountriesAndContinents(ArrayList<Country>, ArrayList<Continent>): void
getNeighbourName(Country): String
getContinentName(ArrayList<Continent>, int): String
printSeparator(int): void
calculateTotalWidth(int[]): int

Fig 4.1.2: View Package Overview

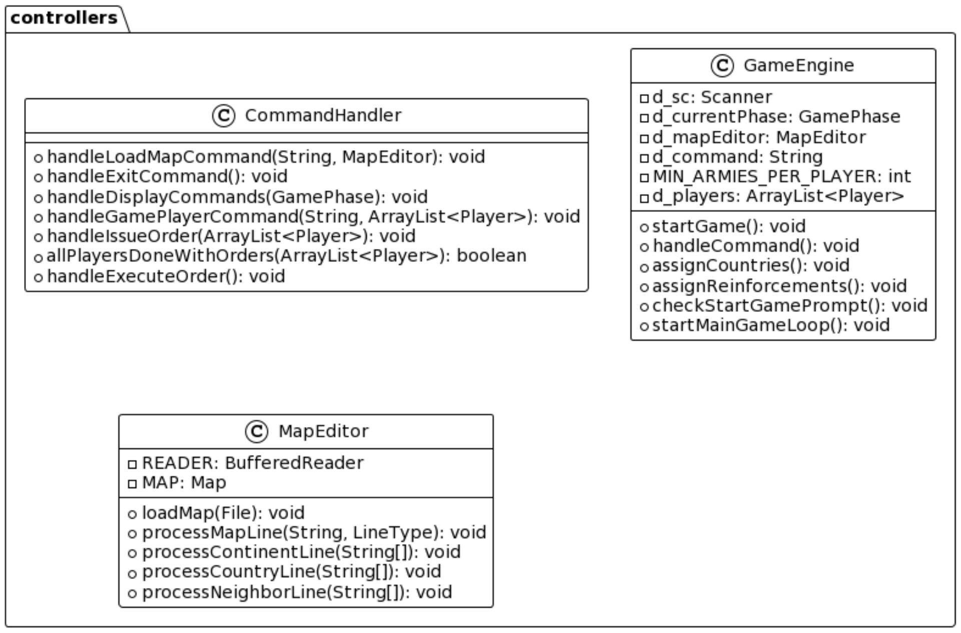
controllers
CommandHandler
handleLoadMapCommand(String, MapEditor): void
handleExitCommand(): void
handleDisplayCommands(GamePhase): void
handleGamePlayerCommand(String, ArrayList<Player>): void
handleIssueOrder(ArrayList<Player>): void
allPlayersDoneWithOrders(ArrayList<Player>): boolean
handleExecuteOrder(): void
GameEngine
d_sc: Scanner
d_currentPhase: GamePhase
d_mapEditor: MapEditor
d_command: String
MIN_ARMIES_PER_PLAYER: int
d_players: ArrayList<Player>
startGame(): void
handleCommand(): void
assignCountries(): void
assignReinforcements(): void
checkStartGamePrompt(): void
startMainGameLoop(): void
MapEditor
READER: BufferedReader
MAP: Map
loadMap(File): void
processMapLine(String, LineType): void
processContinentLine(String[]): void
processCountryLine(String[]): void
processNeighborLine(String[]): void

Fig 4.1.3: Controller Package Overview

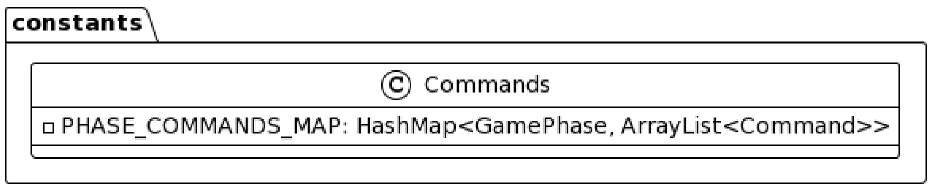

Fig 4.1.4: Constants Package Overview

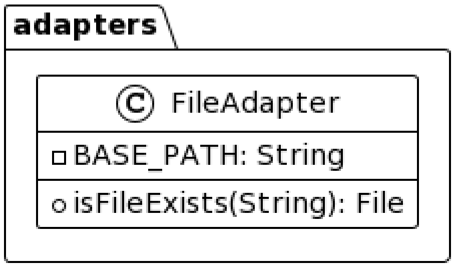

Fig 4.1.5: Adapter Package Overview

5. Testing Strategy

The testing strategy for the Warzone game project encompasses unit testing, integration testing, and system testing to ensure comprehensive coverage of all functionalities. Key areas of focus include

- **Map Validation:** Ensures that custom maps meet game requirements.
- **Command Processing:** Validates the accurate execution of user commands.
- **Player Management:** Tests the functionality related to player actions and interactions.

Tests are structured parallel to the application's main components, allowing for targeted validation of each module's functionality.

6. Conclusion

This document provides a detailed overview of the architectural design for the Warzone Game Project. By adhering to the MVC pattern and incorporating additional support components, the design aims to achieve a flexible, maintainable, and scalable system. The outlined testing strategy ensures that the game's critical functionalities are rigorously validated, contributing to a robust and reliable gaming experience.