

Architecture

Concordia University



Build 03 : WarZone Game

Team members

Yesha	40290892
Bharati	40294202
Meet	40294571
Kapil	40290467
Nishith	40289556
Tejasvi	40292854

Table of contents

Primary architecture	3
Primary design patterns	4
Folder structure	5
Maps	8
Controllers	10
Logger	13
Orders	15
Strategy	17
Adapter	18
GameEngine	19

Primary architecture

The primary architecture for this project includes the combination of **Data driven architecture** and **Model View Controller** architecture.

With help of this architecture our goal is to achieve following things ::

Clear Separation :

MVC enforces a strong separation between data logic (Model), presentation (View), and interaction handling (Controller). This keeps your code organised as your application grows more complex.

Data Focus :

Data-driven development further highlights the importance of your data structures. This combined approach leads to code that is modular and focused on specific concerns.

Easier Modification :

Changing how you store data has less impact on your views and controllers, and vice versa. This makes making updates or adding features significantly smoother.

Parallel Development :

MVC with data-driven design lends itself to teams working on different aspects of the project simultaneously. For instance, few can focus on the data models, while other developers concentrate on views.

Collaboration :

Clear boundaries between responsibilities aid in improved communication and understanding within the team.

Primary design patterns

1. Observer ::

In the WarZone game, the logger is implemented using the observer pattern. The class implements the interface, and the class implements the interface. This allows the to be notified whenever the changes. When the changes, the is notified and it logs the change to a file. This allows us to keep track of all the changes that occur in the game, which can be useful for debugging and troubleshooting. changes. The method logs the change to a file.

2. Adapter ::

- a. The Adapter design pattern enables seamless integration between classes that have similar capabilities but incompatible interfaces. In this scenario, it allows your GameMap system to read both Domination and Conquest map formats, despite their initially incompatible interfaces. A MapAdapter class is created, acting as a bridge between the GameMap and the ConquestMapReader. The MapAdapter implements the interface expected by the GameMap. Inside the MapAdapter, it holds an instance of the ConquestMapReader.
- b. When the GameMap interacts with the MapAdapter, the adapter translates the requests, calls the relevant ConquestMapReader methods, and adapts the returned data to the format the GameMap expects.

3. Strategy ::

- a. The Strategy pattern allows you to dynamically change how players behave in your game. You have different strategies like "Aggressive," "Benevolent," and "Cheater," each defining how a player generates orders. A central "OrderCreator" works with these strategies without knowing the specifics of each one. This makes it easy to change player behaviour on the fly, add new player types, and keep your code organised.



















4. State ::

In my Warzone project, we used the state pattern to create a flexible architecture for game phases. The GamePhase abstract class defines the interface, with concrete subclasses for each phase. The GameEngine manages game phases, tracking the current phase and calling appropriate methods. This design allows for easy addition of new game phases. The GameEngine observes phase changes and updates the game state accordingly.

5. Command ::

The Command pattern was implemented in the Warzone game to model various unit orders. Each order is represented by a concrete Command class, encapsulating the execution logic. This pattern offers benefits such as encapsulation, reusability, and undo/redo capabilities. For instance, the AdvanceOrder class encapsulates the logic to move a unit to a destination, and the execute method moves the unit accordingly. The Command pattern provides a structured way to manage and execute orders in the game, making it flexible and maintainable.

Folder structure

- ✓  org.team21.game
 - >  controllers
 - >  game_engine
 - ✓  interfaces
 - >  game
 - >  observers
 - ✓  models
 - >  cards
 - >  map
 - >  order
 - >  strategy
 - >  tournament
 - ✓  utils
 - >  adapter
 - >  logger
 - >  validation
 -  Constants
 -  SaveMap

controllers ::

The controllers module will take care of all controllers which will handle core game phases features. Some of the controllers have 1-1 relationships with models while others serve as general purposes. For an example IssueOrderController is directly associated with issue order phase while ShowMapController is used by MapEditorController, StartGameController and Exit phase.

game_engine ::

GameEngine will handle all the game phases with proper flow to make sure certain phases run after one another.

interfaces ::

The interfaces will provide schema to the other classes. GamePhase has been done with Interfaces on top of that Observer, state and command pattern for Logger, GamePhase and Orders have been done with interfaces.

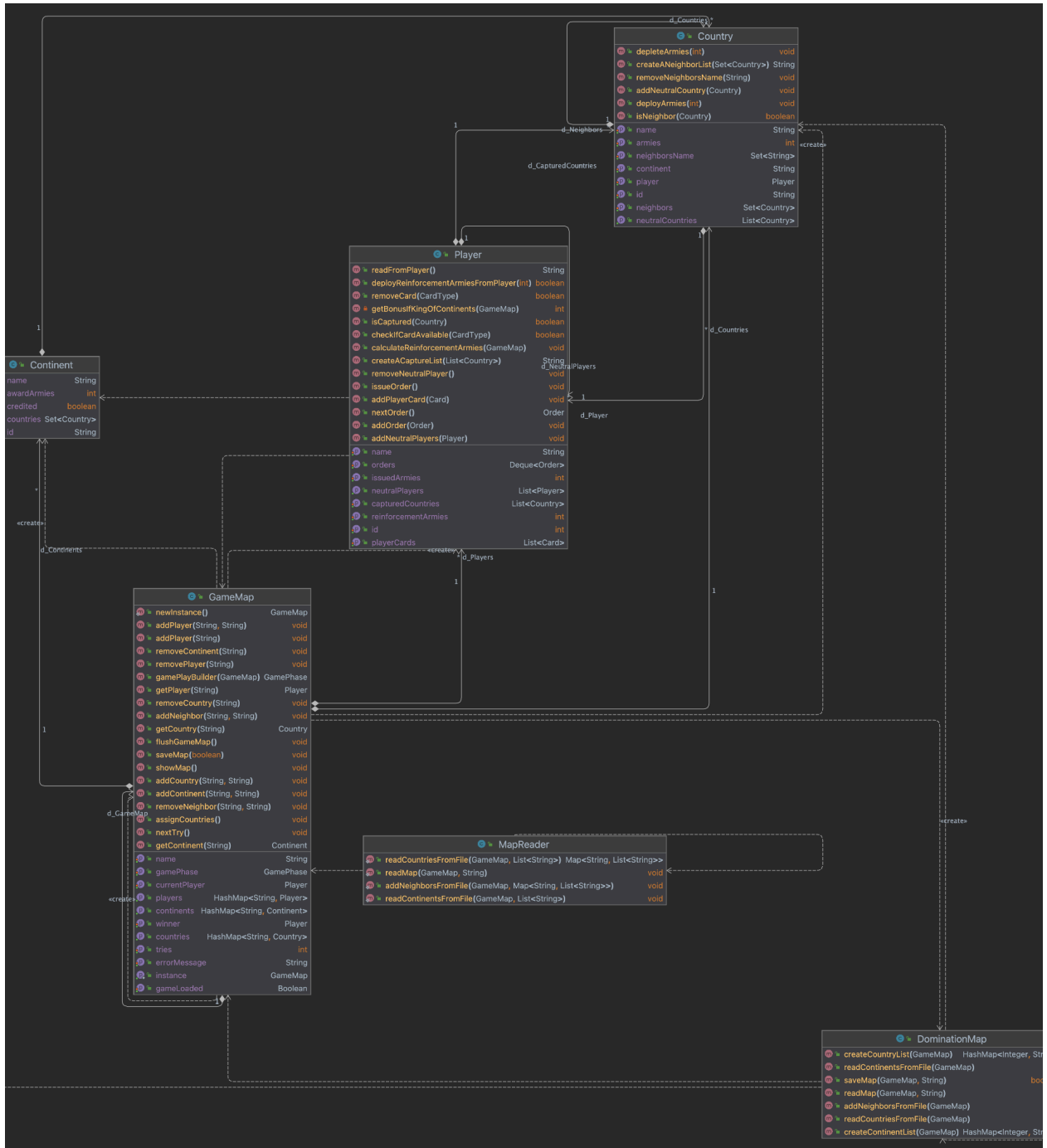
models ::

The models have been used to develop, maintain and preserve the code of all the orders, map, gameplay etc. It also contains various methods to work with different parts of the games.

utils ::

The utils folder will accommodate Constants.Java which has all the constants for hardcoded strings, team name. Common functions to show to users for guiding during orders and other phases. Apart from that, Logger is also implemented here to keep track of all the log files. Moreover, validations for numerous game phases are also handled by utils class.

Maps



The primary components of map :

GameMap ::

Map class holds the details of map in the game. It consists of data structures to access countries and continents and their neighbours of the map. Moreover it has a singleton instance which provides one instance of GameMap across any active game or map editor.

Continent ::

This Class is for the information storage of Continent IDs, their Control Values and Country within that continent. A Hashmap is used to store all the countries belonging to one Continent, Key is CountryId and the Value is its object.

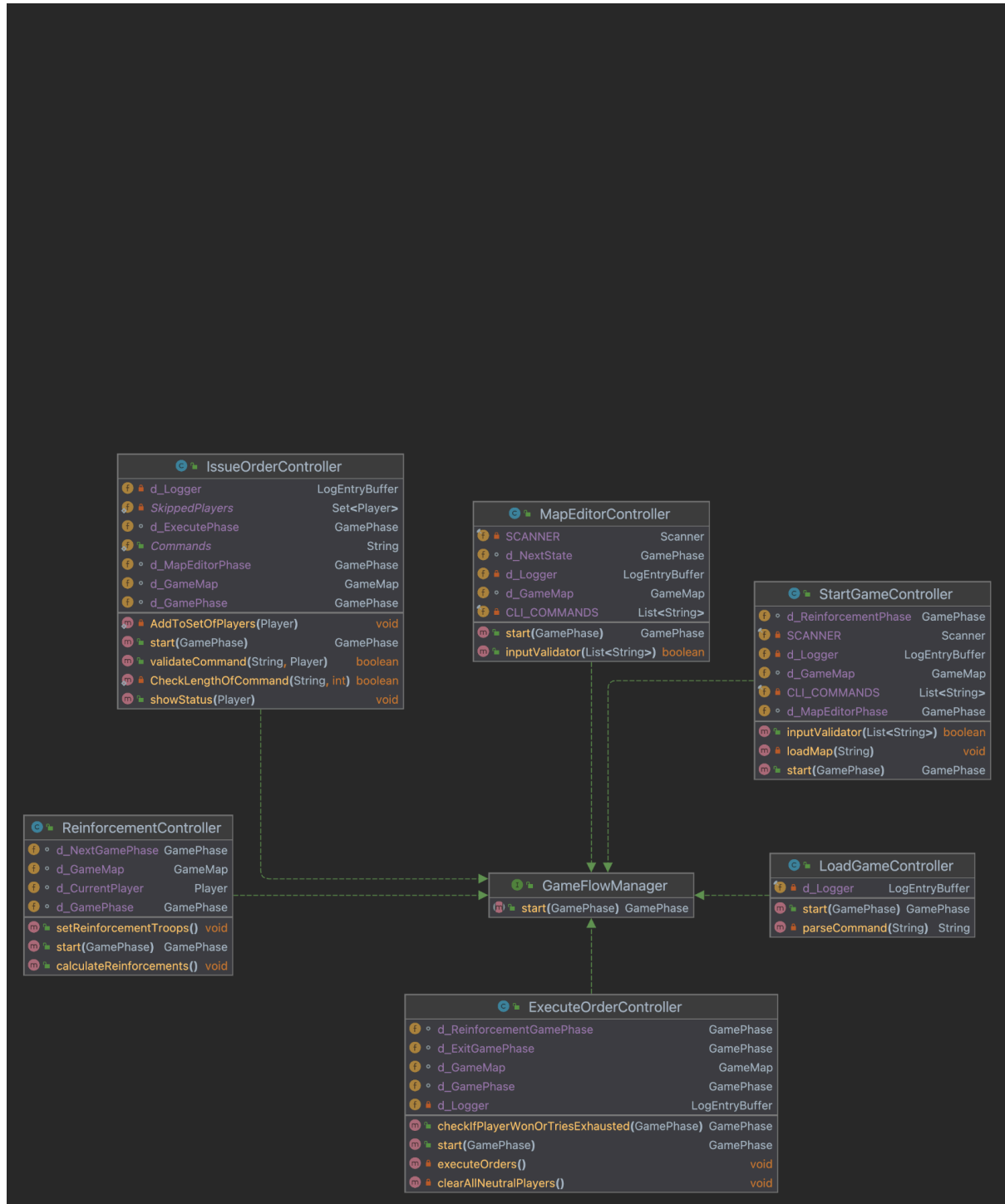
Country ::

This class represents a country in a world map. Each country has an index, ID, parent continent, neighbours, number of armies, and coordinates for CMD display.

Controllers

Interfaces

1. GameFlowManager.Java



Classes

StartGameController.java :

It will load the start game phase and later it will invoke the reinforcement controller.

Main features of this class are ::

1. Loadmap
2. Showmap
3. Add or remove gameplayer
4. Assign Countries
5. Start reinforcement phase

ReinforcementController.java :

It will assign troops based on game features and it will also check validations on different bases.

Main features of this class are ::

1. Assign troops
2. Calculate armies

IssueOrderController.java :

It will issue orders which are provided by users. It will get data from CMD, store it and later issues it with help of *ExecuteOrderController*.

Main feature of this class are ::

1. Get data from CMD
2. Controls the how many players are remaining
3. Stores data with Order interface and child methods

ExecuteOrderController.java :

It will execute all the orders based on Orders e.g. DeployOrder, AdvanceOrder, BombadeOrder etc are stored by IssueOrderController. If countries are left to win then again Reinforcement will start else it will exit the game.

Main feature of this class are ::

1. Execute all the orders
2. Check neutral players
3. Check if countries are won by players
4. Change phase of game to exit or reinforcement

MapEditorController.java :

It will perform map related commands such as editmap, create map, validate map, add or remove countries and continents and also save map.

Main feature of this class are ::

1. Create a map if not exists
2. Edit map
3. Add/Remove continents

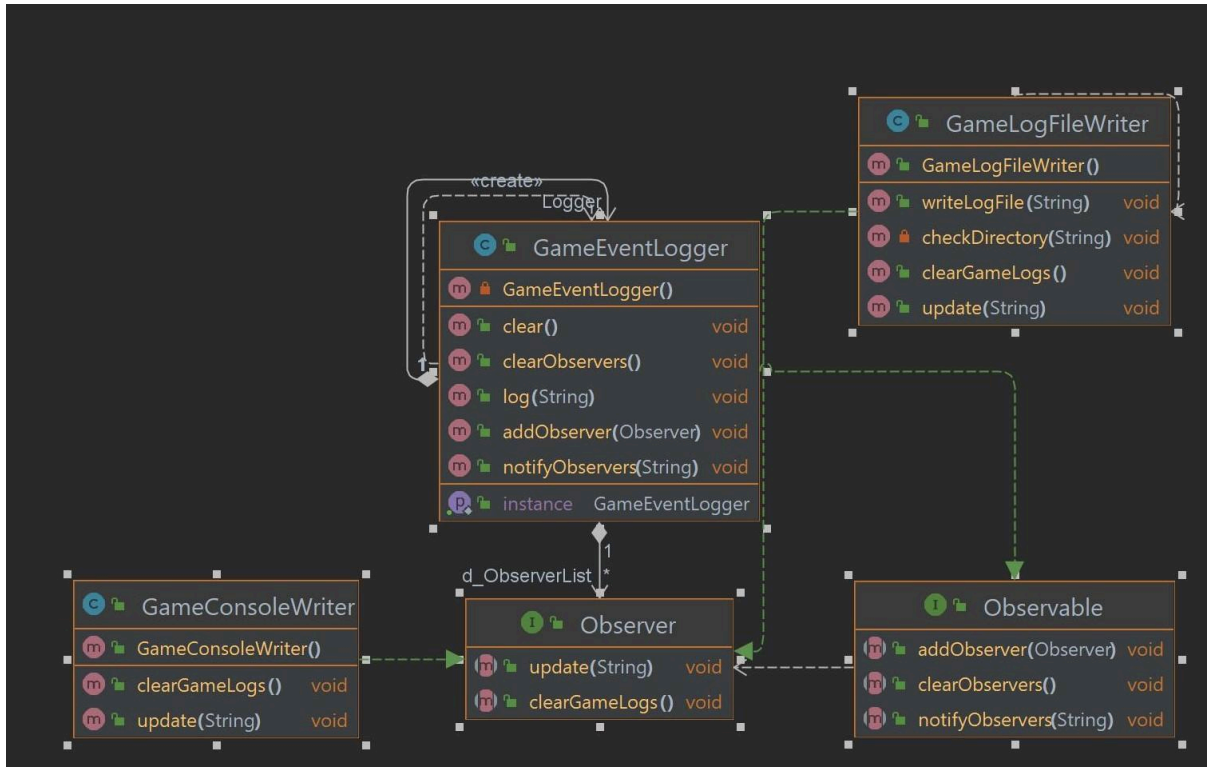
4. Add/Remove neighbours
5. Add/Remove countries
6. Validate map
7. Save map as a file
8. Change phase to start game phase when exit

LoadGamerController.java :

This will load the desired game from the resource and it will pass it to the other controller to play the game.

Logger

The Observer pattern is used to create a logger feature. It will store each and every action provided by the user during the game to keep a track of it. The logs will be stored in **src/** folder.



GameEventLogger.java ::

The `GameEventLogger` class facilitates logging of game events by initialising a `GameLogFileWriter` for writing log entries. It offers methods to log event messages and initialise new log files for game sessions, with the ability to overwrite existing files. By implementing the `Observable` interface, it enables observers to subscribe to events and be notified of changes, achieved by calling the `update` method of the `GameLogFileWriter`. Overall, it serves as a central component for managing game event logging and observer notification within the game system.

GameLogFileWriter.java ::

The `GameLogFileWriter` class is responsible for writing log entries to designated log files. Upon initialization, it sets up the directory path for log files, and if the directory doesn't exist, it creates it. It provides a method to set the filename prefix for log files and another method to write log entries, appending a timestamp to each entry. The `update` method, implemented from the `Observer` interface, serves to receive messages from observers and writes them as log entries using the `writeLogEntry` method. Overall, it manages the creation and writing of log entries for game events.

Observable.java ::

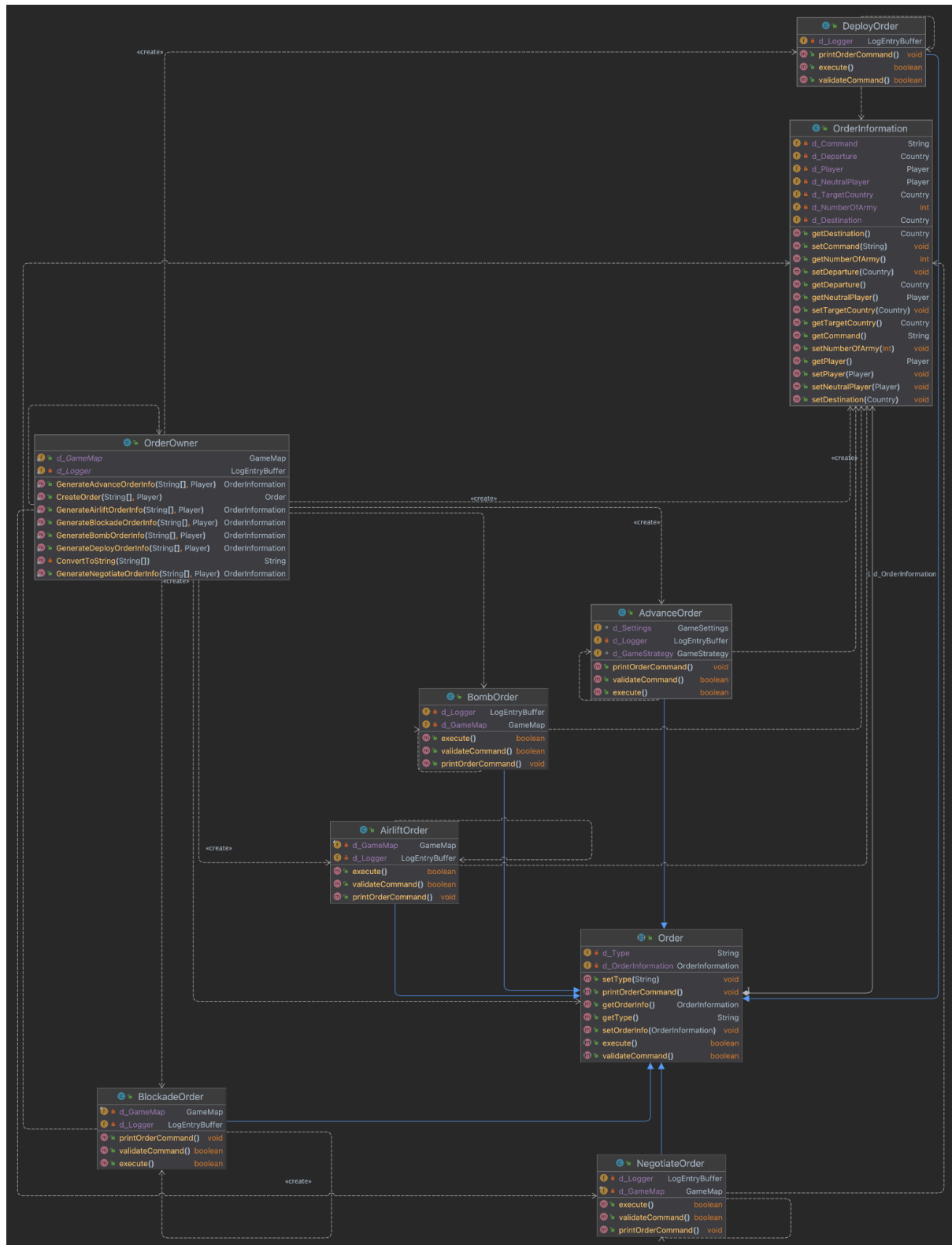
The Observable interface outlines a contract for objects that can be observed by other components. It declares a method `notifyObservers(String p_S)` which is used to inform observers about changes or events within the observable object. The parameter `p_S` typically represents the message or data being sent to the observers. Classes that implement this interface are expected to provide an implementation for notifying observers about changes occurring within the object. This interface is commonly used in observer design patterns where objects need to communicate changes to a set of dependent objects.

Observer.java ::

The Observer interface defines a contract for objects that observe and react to changes in other objects, typically implemented in the observer design pattern. It specifies a method `update(String p_S)` that is called when the observed object notifies its observers about a change. The parameter `p_S` represents the message or data being sent from the observed object to the observer. Classes that implement this interface are expected to provide functionality to handle updates received from observed objects. This interface facilitates loose coupling between observed and observing objects, enabling flexible and maintainable software design.

Orders

Command Pattern

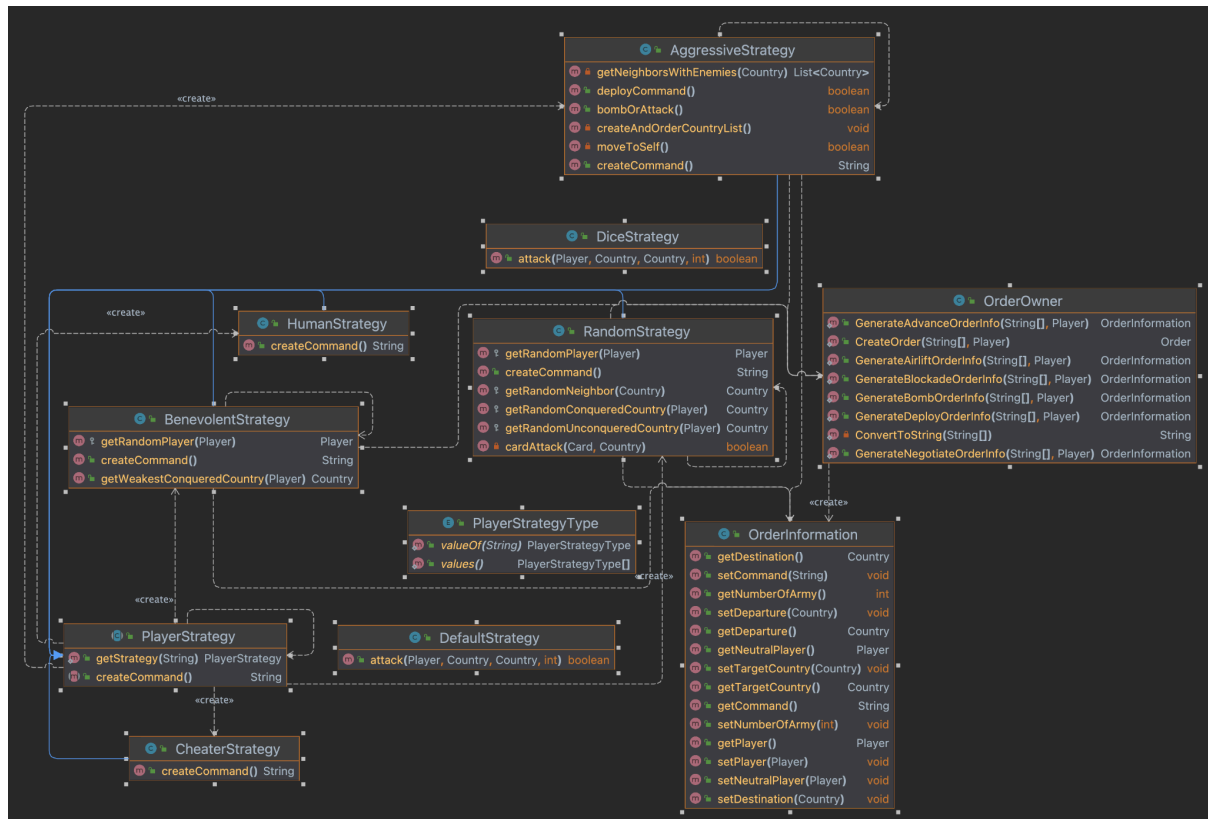


```
public void setType(String p_Type) {  
    this.d_Type = p_Type;  
}
```

The Command pattern makes it straightforward to add or modify how orders are processed within your game. This flexibility stems from separating order creation and execution. Orders can be generated and queued for later execution, enhancing the dynamism of gameplay. The pattern also improves testability by keeping order logic isolated. By separating the object requesting an operation from the object that executes it, you gain the ability to easily introduce new commands without modifying existing classes.

Strategy

Strategy pattern



Dynamic Player Behavior through the Strategy Pattern

In this game design, the Strategy pattern enables flexible and adaptable player behaviour during gameplay. The "OrderCreator" class serves as the context, utilising various player strategies to generate game commands.

Strategic Blueprints

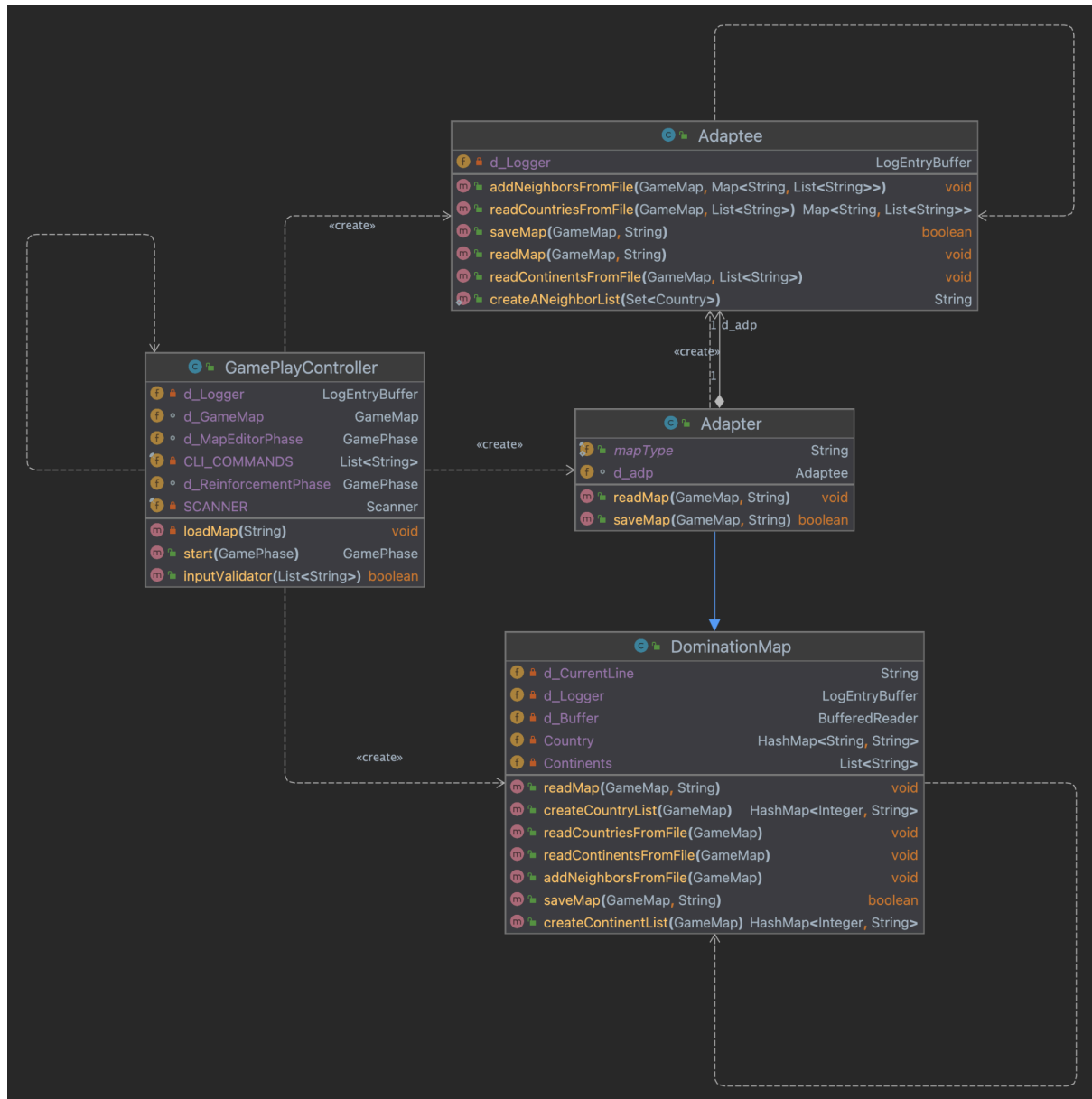
Each player strategy (e.g., "Aggressive," "Benevolent," "Cheater") encapsulates a distinct approach to command creation. "Aggressive" strategies prioritise conquest, "Benevolent" strategies emphasise defence, while "Cheater" strategies subvert game rules. The "Human" strategy uniquely incorporates player input.

Advantages of this Approach

1. **Modifiable AI:** New player strategies can be seamlessly integrated, expanding the range of in-game behaviors.
2. **Compelling Player Experiences:** Strategies create the illusion of diverse and intelligent opponents.
3. **Code Maintainability:** By separating command generation logic, the "OrderCreator" avoids becoming overly complex.

Adapter

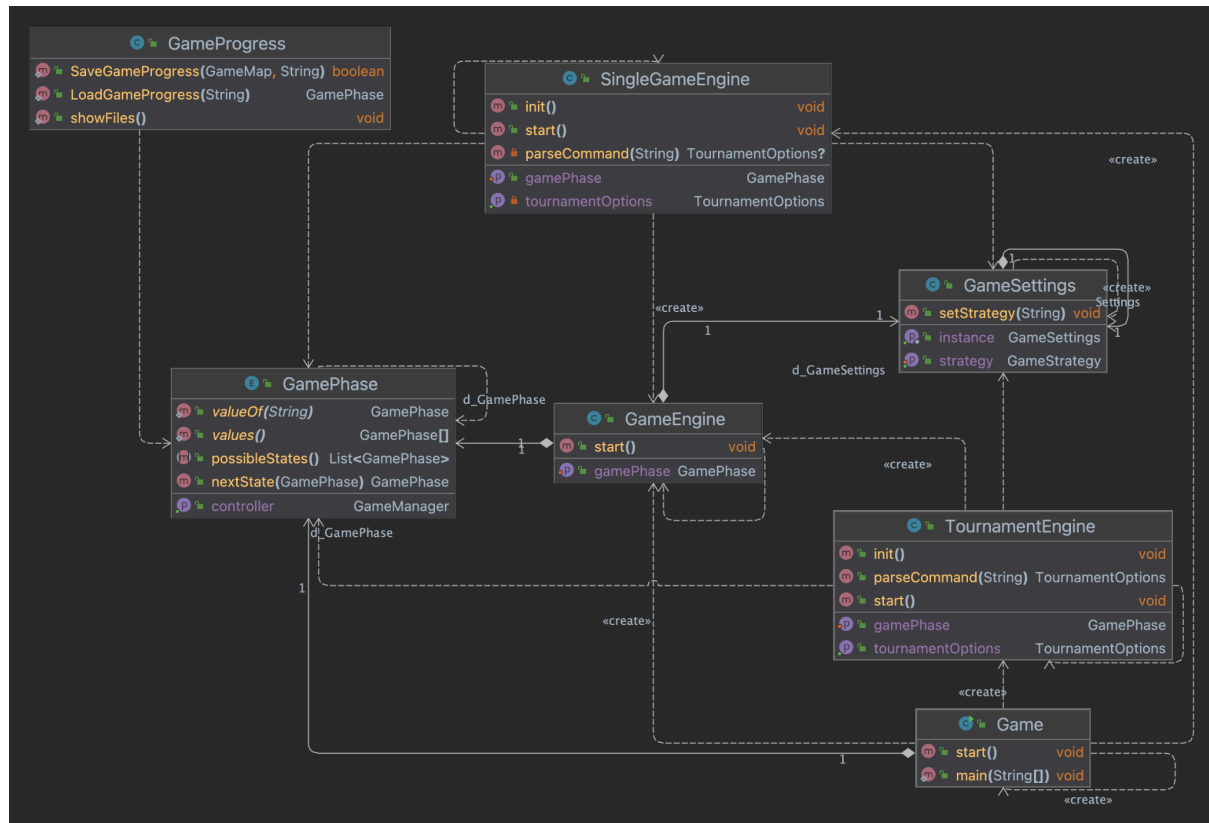
Adapter pattern



The Adapter pattern provides a bridge between classes with similar functionality but incompatible interfaces, facilitating their collaboration. In this implementation, the GameMap (client) interacts with a DominationMapReader (target). An adapter class (MapAdapter) translates between this target interface and a ConquestMapReader (adaptee). This design allows GameMap to process both Domination and Conquest map formats without modifying the underlying map reader classes.

GameEngine

State pattern



Game engine will primarily work as the main component to change phases and it will control the other aspects of the game such as tournament and single player mode. The Game class is the entry point of the game from which the entire game will work. The GamePhase is a enum class which has required methods and enums to change phases of the Game. The Game settings will set and give strategy to be used in the Game.