Architecture Design

Concordia University

Build 03: WarZone Game

Group: 10

Team members

Ankush Desai	40271170
Archil Katrodiya	40270119
Mihir Gediya	40272399
Krupali Dobariya	40292874
Hongwu Li	40280054

Table of Contents

1 PRIMARY ARCHITECTURE

The main architectural framework for this project involves integrating both **Data-Driven** and **Model-View-Controller (MVC)** architectures.

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

Clear Separation:
MVC ensures a clear distinction between data logic (Model), presentation (View), and interaction handling (Controller), facilitating organized code maintenance, especially as your application becomes more intricate.

Data Focus:
Data-driven development emphasizes the value of your data structures. This combination approach produces code that is modular and focuses on certain topics.

Easier Modification:
Changing the way, you store data has a less influence on your views and controllers, and vice versa. This makes it much easier to update or add new features.

Parallel Development:
MVC with data-driven design allows teams to work on multiple areas of the project at the same time. For example, few developers can focus on data models, whilst others can focus on displays.

Collaboration:
Clear responsibilities help to increase communication and understanding among team members.

2 PRIMARY DESIGN PATTERNS

2.1 Observer

In the WarZone game, the logger uses the observer pattern. One class implements the interface, and another does the same. This configuration allows them to receive notifications anytime a change occurs. When notified, it logs the modification to a file. This method tracks all game modifications, which aids debugging and troubleshooting efforts.

2.2 Adapter

The Adapter design pattern provides smooth integration of classes with similar functionality but conflicting interfaces. In this case, it enables your GameMap system to read both Domination and Conquest map formats despite their original incompatibility. A MapAdapter class is constructed to serve as a bridge between the GameMap and the ConquestMapReader.The MapAdapter implements the interface specified by the GameMap.The MapAdapter contains a ConquestMapReader instance.
The GameMap interacts with the MapAdapter, which translates requests, calls ConquestMapReader functions, and transforms the returned data to the GameMap's expected format.

2.3 Strategy

The Strategy pattern allows for dynamic adjustment of player behavior throughout the game. Players generate orders using various techniques such as "Aggressive," "Benevolent," and "Cheater." A central "OrderCreator" works with these techniques without knowing the specifics of each. This enables seamless change of player behavior, the addition of new player kinds, and code organization.

2.4 State

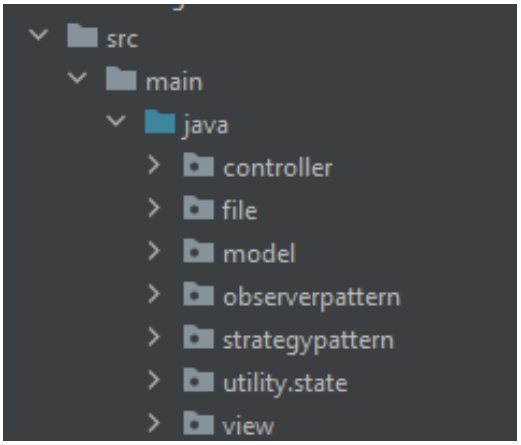
The state pattern was used in the Warzone project to create a flexible architecture for game stages. The GamePhase abstract class defines the interface, with concrete subclasses for each phase. The GameEngine manages game phases, keeping track of the current phase and implementing appropriate functions as needed. This method allows for the smooth incorporation of new game stages. Additionally, the GameEngine detects phase changes and updates the game state accordingly.

2.5 Command

The Command pattern was included in the Warzone game to simulate various unit orders. Each order is represented by a concrete Command class, which encapsulates the execution logic. This pattern offers encapsulation, reusability, and undo/redo functionality. For example, the AdvanceOrder class encapsulates the logic for moving a unit to a location, with the execute

function properly carrying out the unit movement. The Command pattern provides an organized approach to managing and executing orders in the game, which improves flexibility and maintainability.

3 FOLDER STRUCTURE



- **controller:**
  The controller's module manages all controllers that are responsible for basic game phase functionality. Some controllers have one-on-one connections with models, whereas others provide more generic functions. For example, the MapController is directly related to the map model.

- **file:**
  This package has the code for adapter design pattern to save and load the game stats.

- **model:**
  The models serve as the foundation for creating, updating, and organizing code for all orders, maps, gameplay, and other components of the game. They provide multiple techniques for interacting with different game components, ensuring effective handling and management of game-related functionalities.

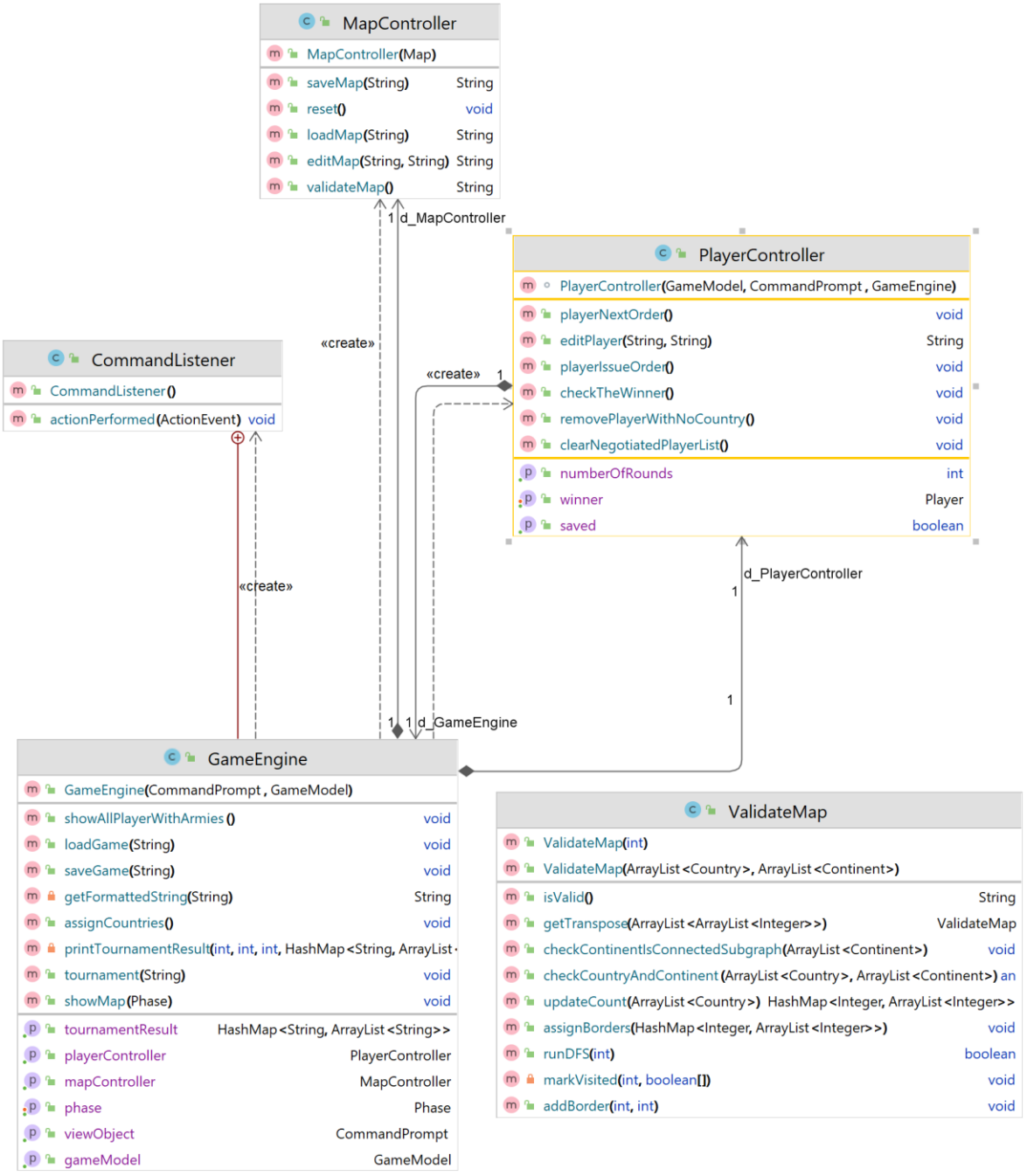
- **observerpattern:**
  The obeserverpattern package contains the implementation of the logger functionality. It used observer pattern and logs the output to the log file, following the design pattern.

- **strategypattern:**
  The package contains the implementation of the strategy pattern to providing strategy of the different player behaviors.

- **Utililty.state:**
  It provides the state design pattern implementation. It ensures the commands are executing in the current state. i.e, while editing map, user cannot use the commands for attack the country.

- **View:**
  This provides basic files for applet, to take input and showing outputs and starting point of the game.

4 Controller



- **GameEngine.java**
  Main Functionality of this code is:
  1. CommandListener
  2. GamePhases
  3. TournamentMode

  It features a robust architecture where models represent game entities, views provide user interfaces, and controllers manage game logic and user interactions. The GameEngine class is

the central controller, coordinating interactions between different components and implementing game phases. Notably, it supports a tournament mode, allowing users to simulate multiple games across various maps and player strategies. Error handling mechanisms ensure graceful handling of unexpected conditions, enhancing the overall reliability of the application.

- **MapController.java**

  Main Functionality of this code is:
  1. Map Editing Operations
  2. Map Loading and Saving

  MapController is responsible for managing map-related functionalities in a game. It facilitates operations such as saving and loading maps, editing map elements like continents, countries, and borders, and validating map integrity. Additionally, it communicates with the Map model to execute these operations and provides feedback messages to the user based on the outcomes.

- **PlayerController.java**

  Main Functionality of this code is:
1. Issue Player Orders
   2. Player Order Executions

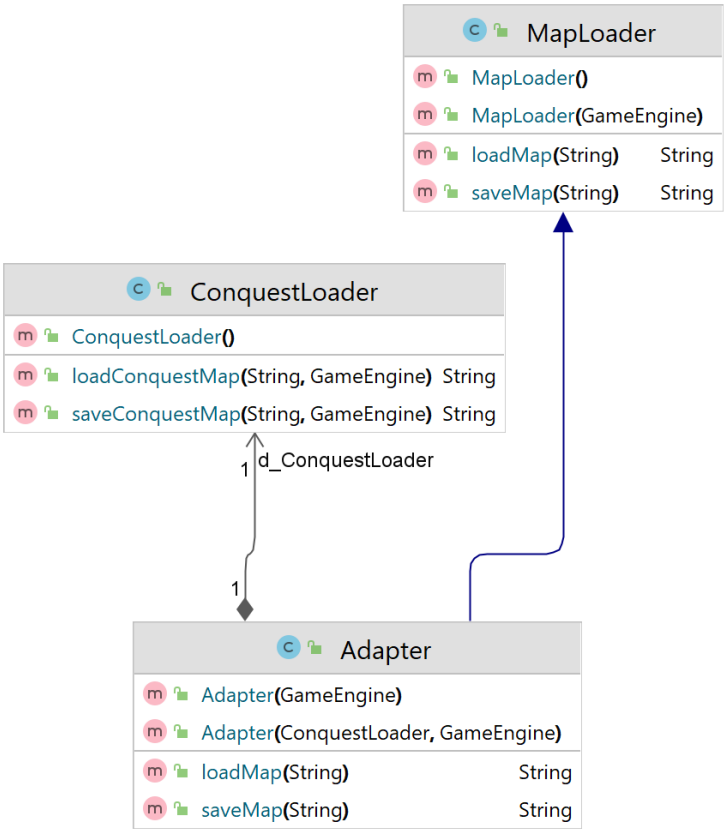
  The playerController class is responsible for managing player activities in a strategy game. It orchestrates the issuance and execution of orders by players, handles adding or removing players from the game, checks for game-winning conditions, and assigns cards to victorious players. The controller interacts with the game model and views components and player strategies to coordinate gameplay and ensure fair competition. It employs round-robin scheduling to manage player turns and provides feedback to players through a command prompt interface. The code encapsulates the logic for player management and order processing in a turn-based strategy game environment.

- **MapValidate.java**

  Main Functionality of this code is:
  1. Map Validation

  The ValidateMap class validates the integrity of a strategy game map by representing it as a graph and conducting various checks. It ensures each continent contains at least one country, verifies internal connectivity within continents, and confirms that the overall map forms a connected graph. DFS traversal and graph transposition guarantee the map's validity for gameplay, reinforcing adherence to game rules and ensuring a robust foundation.

5 file



- **ConquestLoader .java**

  This is the ConquestLoader class which has roadmap and save map functions for the conquest map type. ConquestLoader's functionality is similar to the target, but it is used for another type of map file.

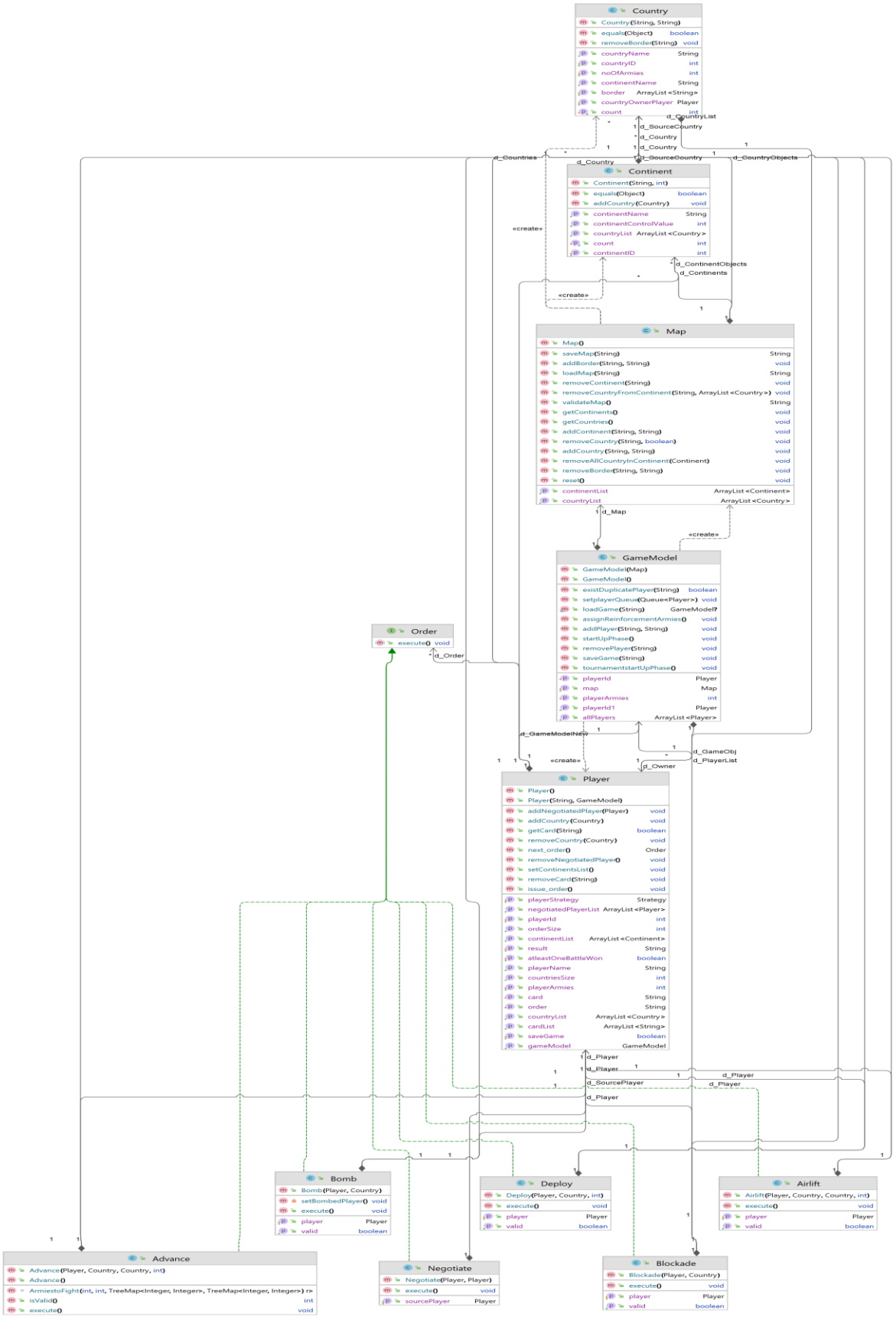
- **Adapter.java**

  Adapter class works as a bridge between the target and adaptee. We can use the adapter to call the method of adaptee by calling the methods used in target.

- **MapLoader.java**

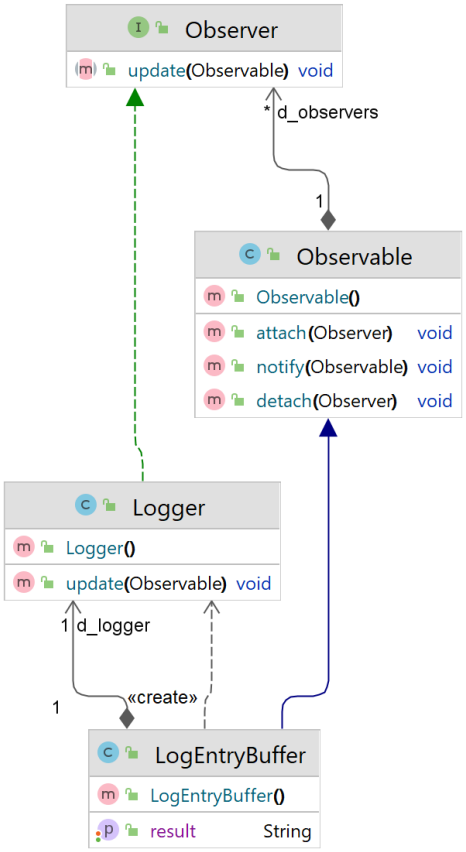
  This is the target class for the adapter pattern. This class is used to call the existing implementation of loadmap and savemap functionality of domination map type.

6 Model


Country
Country(String, String)
equals(Object) boolean
removeBorder(String) void
countryName String
countryID int
noOfArmies int
continentName String
border ArrayList<String>
countryOwnerPlayer Player
count int
d_CountryList
d_SourceCountry
d_Country
d_Countries
d_CountryObjects
Continent
Continent(String, int)
equals(Object) boolean
addCountry(Country) void
continentName String
continentControlValue int
countryList ArrayList<Country>
count int
continentID int
«create»
d_ContinentObjects
d_Continents
Map
Map()
saveMap(String) String
addBorder(String, String) void
loadMap(String) String
removeContinent(String) void
removeCountryFromContinent(String, ArrayList<Country>) void
validateMap() String
getContinents() void
getCountries() void
addContinent(String, String) void
removeCountry(String, boolean) void
addCountry(String, String) void
removeAllCountryInContinent(Continent) void
removeBorder(String, String) void
reset() void
continentList ArrayList<Continent>
countryList ArrayList<Country>
d_Map
GameModel
GameModel(Map)
GameModel()
existDuplicatePlayer(String) boolean
setplayerQueue(Queue<Player>) void
loadGame(String) GameModel?
assignReinforcementArmies() void
addPlayer(String, String) void
startUpPhase() void
removePlayer(String) void
saveGame(String) void
tournamentstartUpPhase() void
playerId Player
map Map
playerArmies int
playerId1 Player
allPlayers ArrayList<Player>
d_GameModelNew
d_GameObj
d_PlayerList
Order
execute() void
d_Order
Player
Player()
Player(String, GameModel)
addNegotiatedPlayer(Player) void
addCountry(Country) void
getCard(String) boolean
removeCountry(Country) void
next_order() Order
removeNegotiatedPlayer() void
setContinentsList() void
removeCard(String) void
issue_order() void
playerStrategy Strategy
negotiatedPlayerList ArrayList<Player>
playerId int
orderSize int
continentList ArrayList<Continent>
result String
atleastOneBattleWon boolean
playerName String
countriesSize int
playerArmies int
card String
order String
countryList ArrayList<Country>
cardList ArrayList<String>
saveGame boolean
gameModel GameModel
d_Player
d_SourcePlayer
d_Owner
Bomb
Bomb(Player, Country)
setBombedPlayer() void
execute() void
player Player
valid boolean
Deploy
Deploy(Player, Country, int)
execute() void
player Player
valid boolean
Airlift
Airlift(Player, Country, Country, int)
execute() void
player Player
valid boolean
Advance
Advance(Player, Country, Country, int)
Advance()
ArmiestoFight(int, int, TreeMap<Integer, Integer>, TreeMap<Integer, Integer>) r>
isValid() int
execute() void
Negotiate
Negotiate(Player, Player)
execute() void
sourcePlayer Player
Blockade
Blockade(Player, Country)
execute() void
player Player
valid boolean

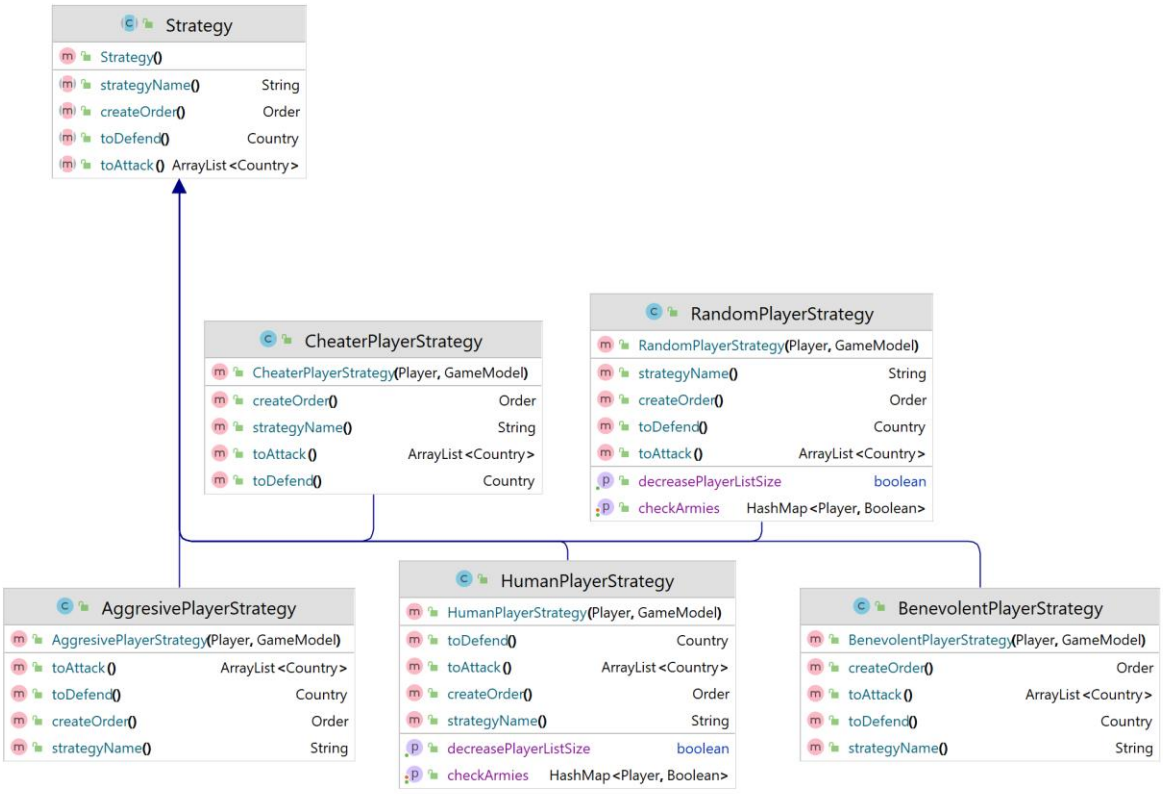
- **Continent.java**
  This Class is for all the Continents of the Map.
- **Country.java**
  This is a class for all the countries/territories.
- **GameModel.java**
  Represents the GameModel class which manages the game data and logic.
- **Map.java**
  This class consists of all the data members and behavior associated with Map.
- **Order.java**
  Parent interface which is implemented by different Orders in the game.
- **Player.java**
  The Player class represents a player in the game.
- **Advance.java**
  The Advance class is a type of an Order issued by a Player. This Order provides the ability for a player to attack a territory belonging to some other player with some number of armies.
- **Airlift.java**
  Airlift class implements the Order interface and overrides the execute method.
- **Blockade.java**
  The Blockade class represents a blockade order in the game.
- **Deploy.java**
  Represents a Deploy order, where a player deploys armies to a specific country.
- **Bomb.java**
  The bomb class represents a bomb order in the game.

7 Overserver Pattern



- **LogEntryBuffer.java**
  To log any changes in file, we use the object of this class.
- **Logger.java**
  This is a concrete class which implements observer interface.
- **Observable.java**
  Observable class which has methods to connect/disconnect with observers and notifies if there is any update.
- **Observer.java**
  Interface observer which has one update method which is called when observable notifies it.

8 Strategy Pattern



- **AggresivePlayerStrategy.java**

  This is a class which creates orders from the AggresivePlayer according to his strategy. This class extends the parent Strategy class which has createOrder method to be implemented here.

- **BenevolentPlayerStrategy.java**

  This class implements the Strategy for the benevolent type of player. This type of player always deloys on its weak country and never attacks.

- **CheaterPlayerStrategy.java**

  This Strategy class belongs to the Cheater Player. It encapsulates the behavior of a Cheater Player. This strategic player only issues deploy and advance orders to attacks on neighboring countries and increases the number of armies on its countries that have enemy neighbors.

- **HumanPlayerStrategy.java**

  This Strategy class belongs to the Human Player. It encapsulates the behavior of a Human Player.

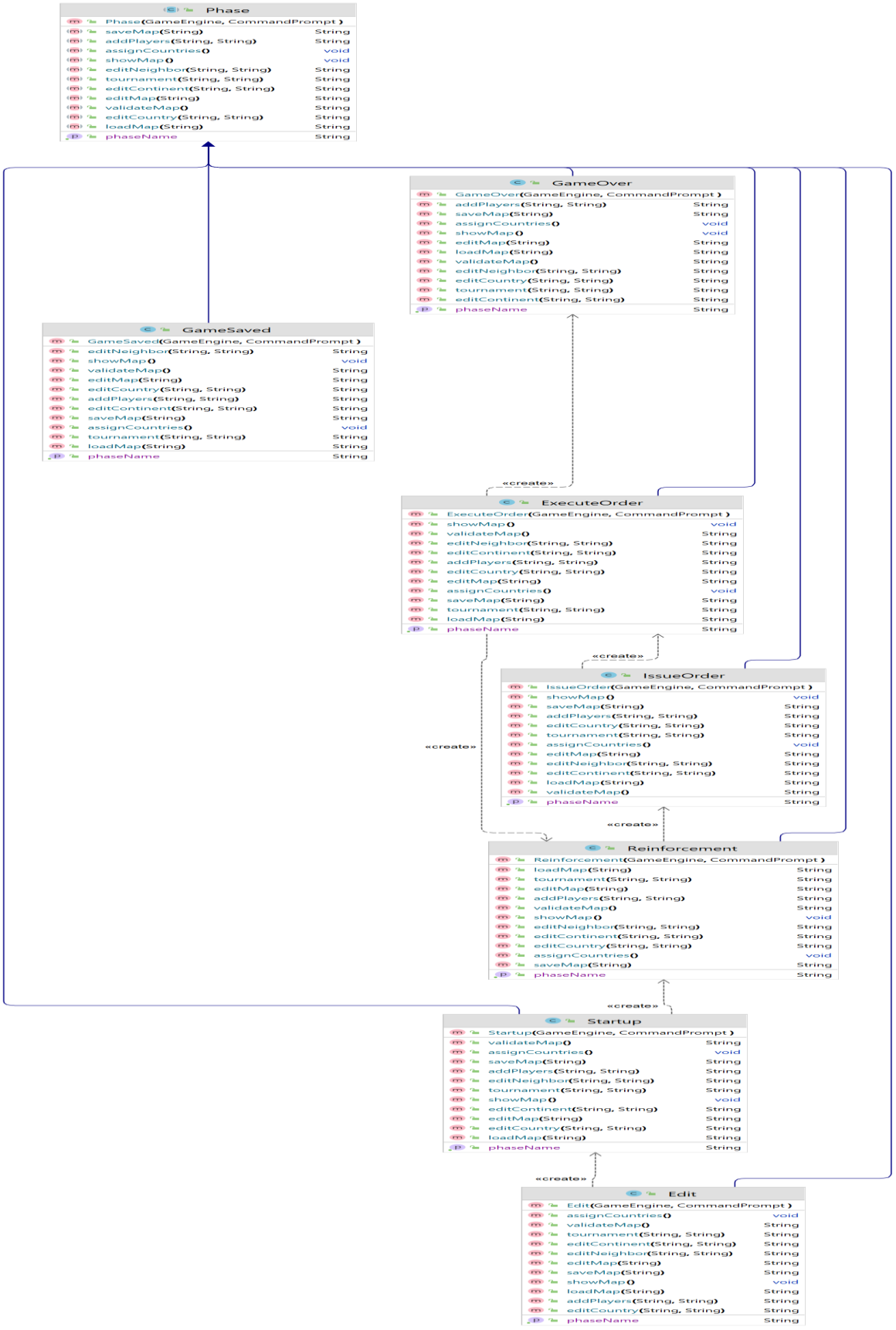
- **RandomPlayerStrategy.java**

  This Strategy class belongs to the Random Player. It encapsulates the behavior of a Random Player.

- **Strategy.java**

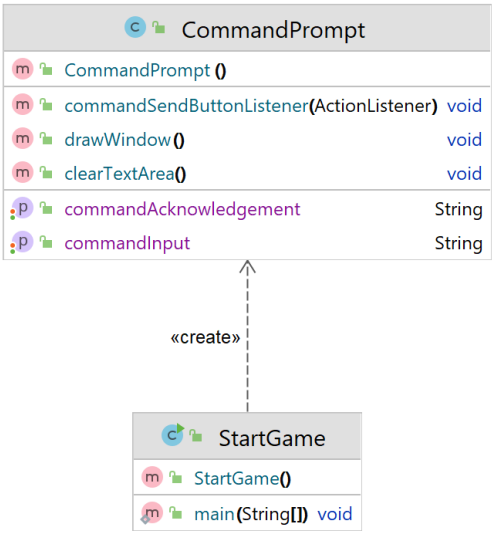
  This is a parent class for the strategy pattern. Each player has their different strategy and they extend this class and implements the methods accordingly.

9 State Pattern



- **Edit.java**
  Represents the Edit Phase, which extends the Phase class and implements methods specific to this phase.
- **ExecuteOrder.java**
  The ExecuteOrder phase extends the Phase class and implements methods specific to this phase. It handles executing orders during gameplay.
- **GameOver.java**
  It handles the end of the game and provides relevant acknowledgements.
- **IssueOrder.java**
  It handles issuing orders during gameplay.
- **Phase.java**
- Represents an abstract class for game phases, each phase extending this class represents different game states.
- **Reinforcement.java**
  It handles reinforcing armies during gameplay.
- **StartUp.java**
  It returns invalid command for others which are not compatible with this Phase.

10 View



- **CommandPrompt.java**
  The window interacts with the user and show feedback.
- **StartGame.java**
  It is a driver class of the game.