COURSE: SOEN 6441:ADVANCED PROGRAMMING PRACTICES

SUPERVISOR: JOEY PAQUET

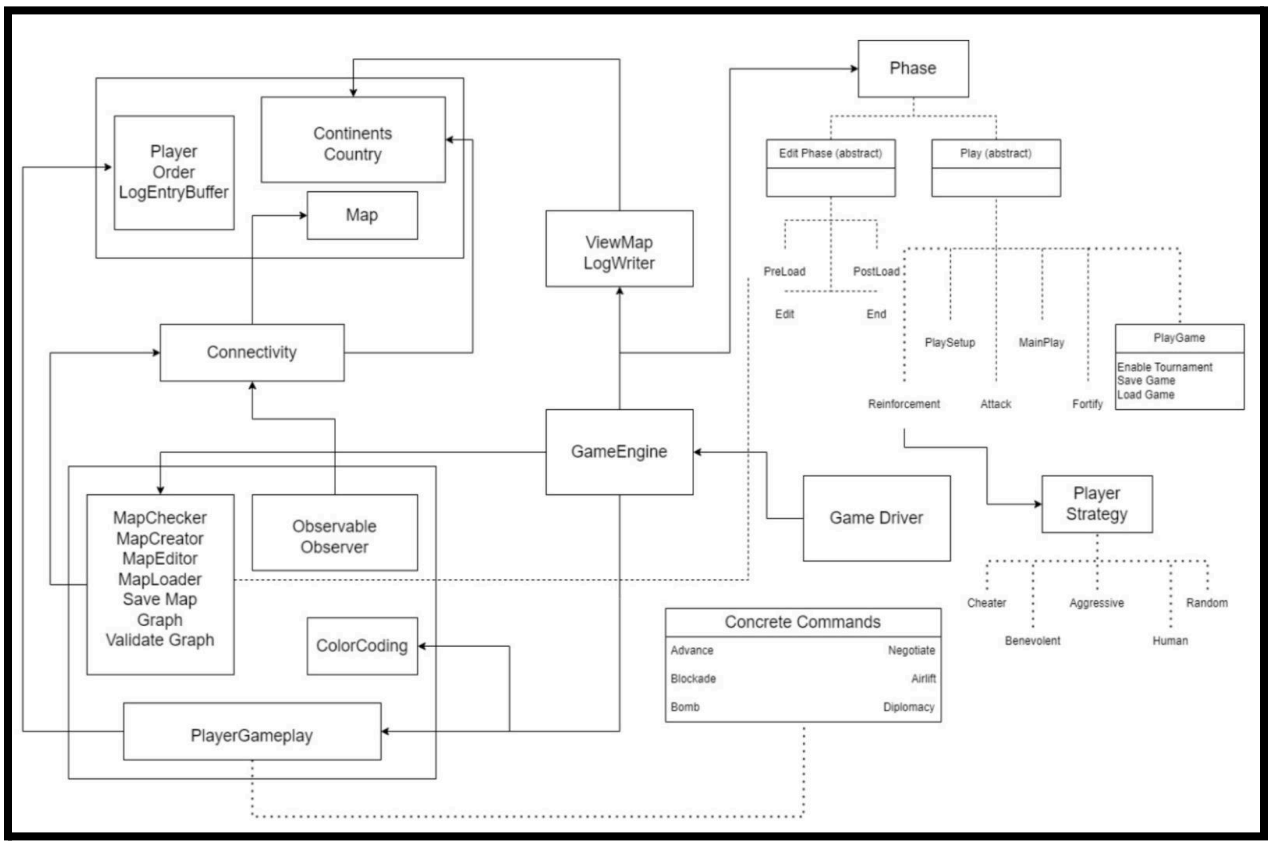
PROJECT: WARZONE GAME BUILD 3

TEAM 16: ARCHITECTURAL REPRESENTATION

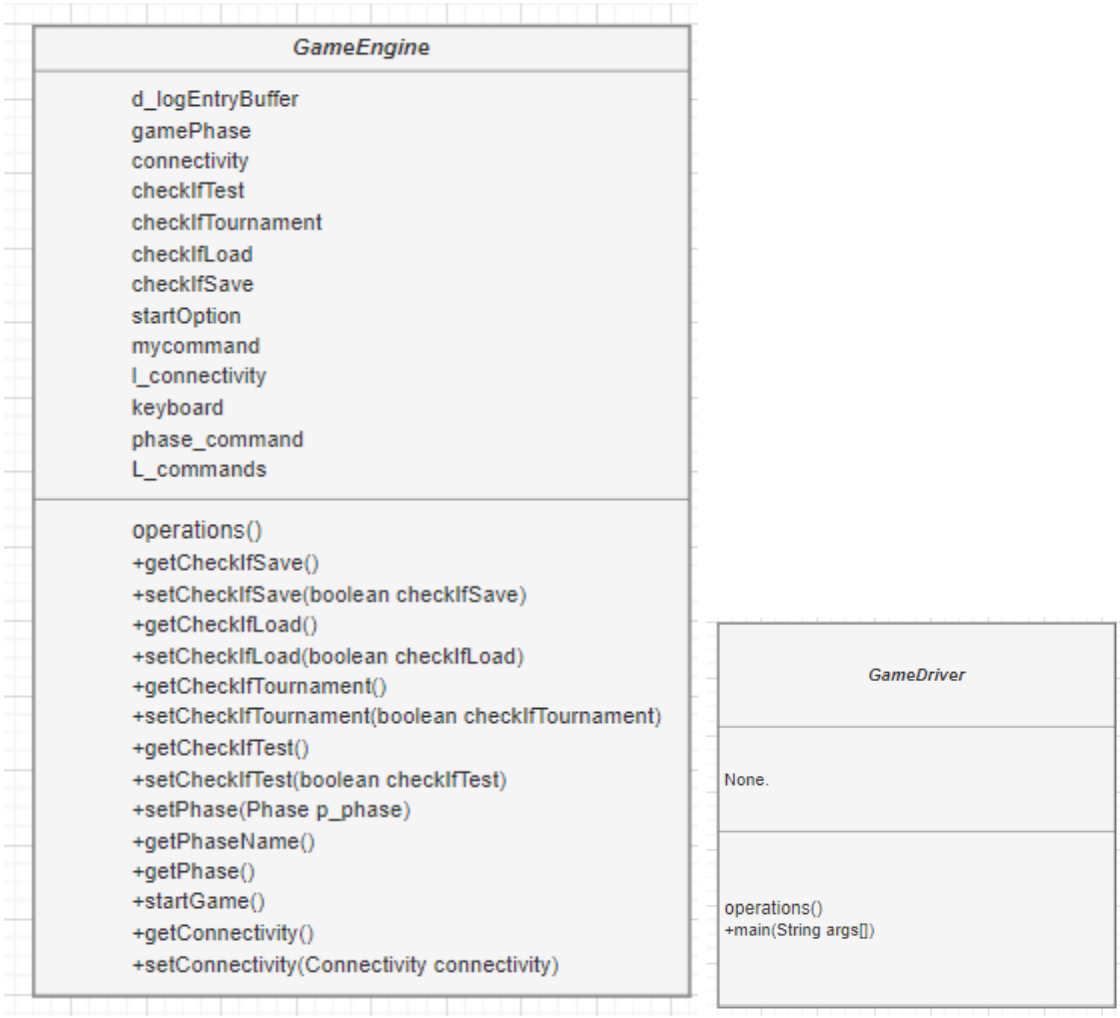
TEAM MEMBERS:

1. Sanjay Bhargav Pabbu
2. Piyush Gupta
3. Blesslin Jeba Shiny Augustin Moses
4. Mahfuzzur Rahman
5. Susmitha Mamula
6. Poojitha Bhupalli

ARCHITECTURAL DIAGRAM



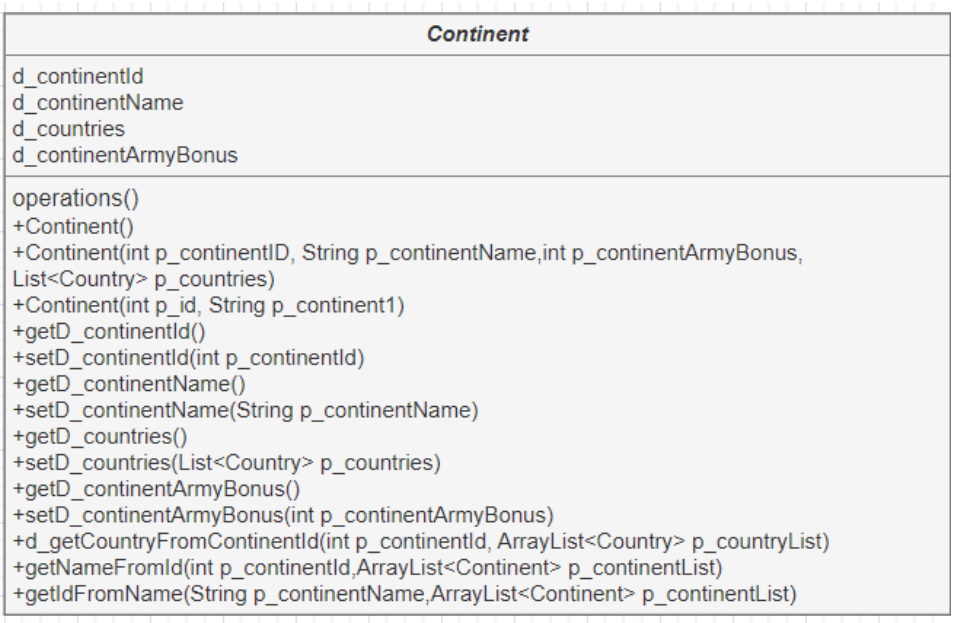
1. Controllers



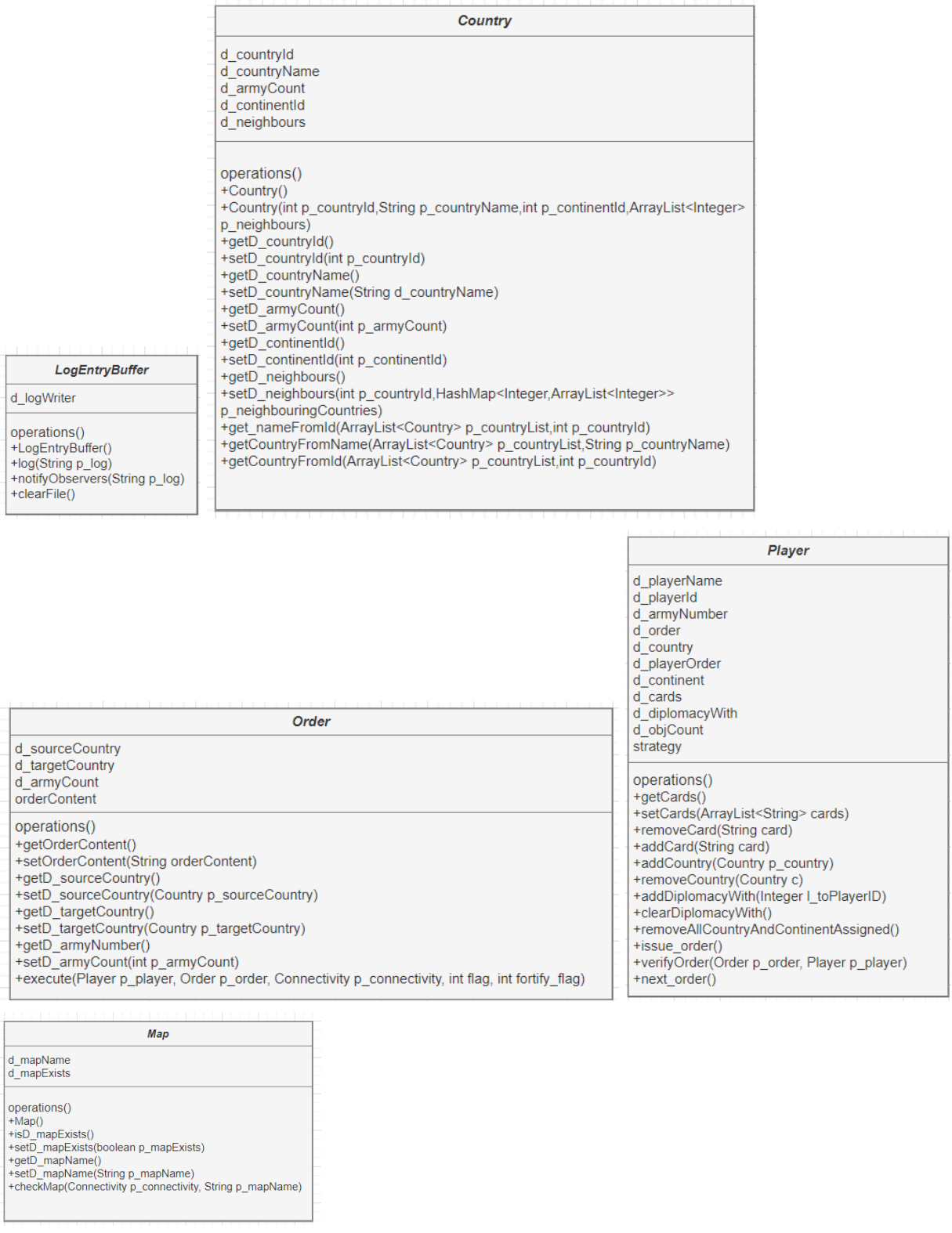
a. GameDriver: This class is responsible for initiating the game engine.

b. GameEngine: This code represents a game engine implementing the State pattern. It manages game phases, handles user input, and controls game flow based on the selected options and commands.

2. Models



a. **Continent**: A model class Continent representing a continent in the game. It contains methods to get and set continent properties such as ID, name, list of countries, and army bonus. Additionally, it provides methods to retrieve information about countries based on the continent ID, and to get continent information based on its name or ID.



b. **Country**: The Country class representing a country in the game. It contains methods to get and set country properties such as ID, name, army count, continent ID, and

neighboring countries. Additionally, it provides methods to retrieve country information based on its ID or name.

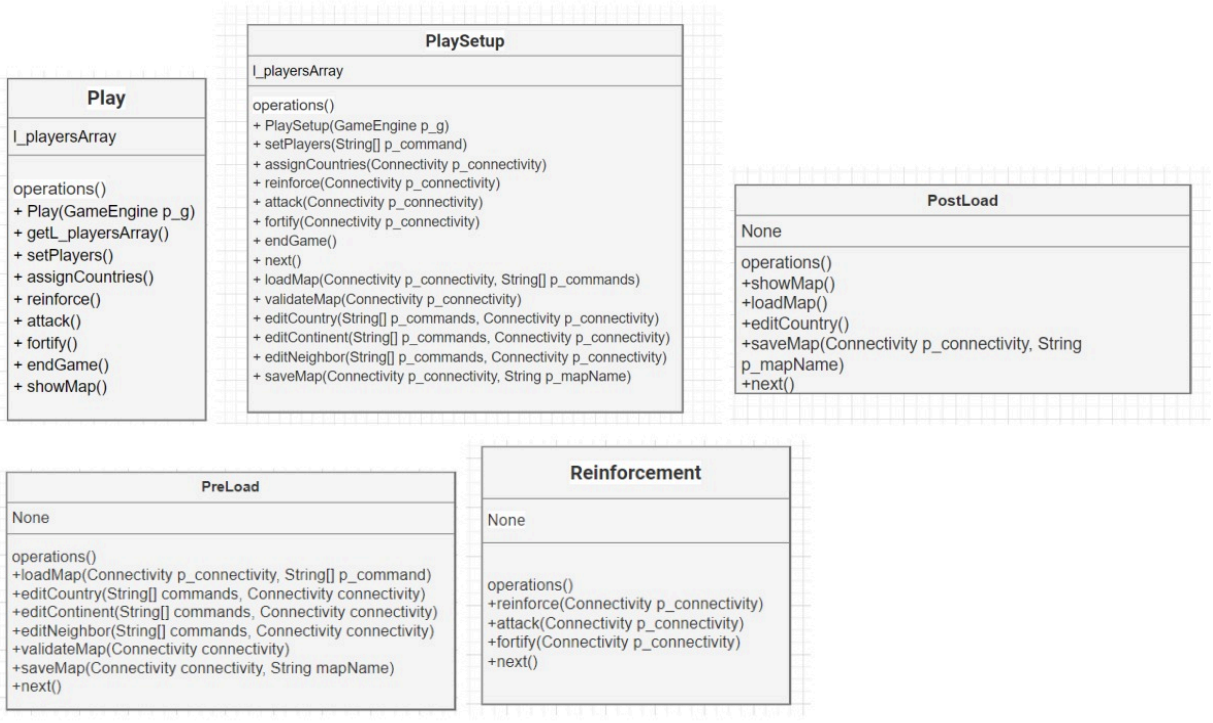
c. **LogEntryBuffer:** It handles logging of game events to a file.

d. **Map**: It defines a map and its properties, including the map name and existence status.

e. **Order**: The order class represents orders in the game, specifying source and target countries and the number of armies to be transferred.

f. **Player:** This class represents a player in the game, including their name, ID, number of armies, countries owned, orders issued, continents owned, and cards possessed.

3. State



a. Attack: It represents the Attack phase in a game, handling player commands related to attacks.

b. Edit: It represents the editing phase in a game, providing methods for editing various aspects of the game map.

c. End: It represents the end phase of the game, handling actions and commands associated with ending the game.

d. Fortify: It represents the phase where players can choose to fortify their countries, allowing them to strengthen their positions on the game map.

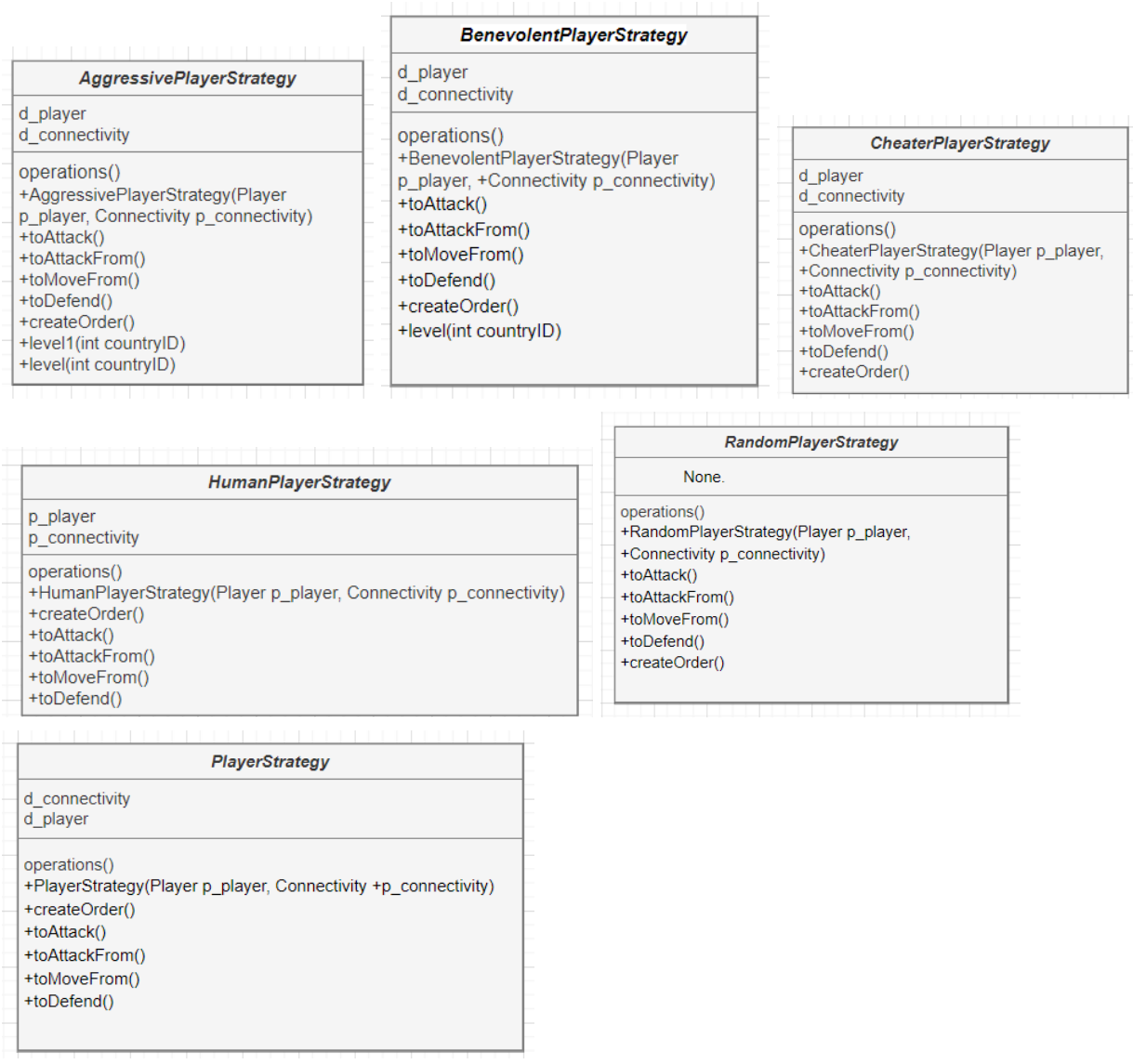
e. MainPlay: MainPlay class is a ConcreteState in the StatePattern. It encapsulates the behavior for commands applicable for states like Reinforcement, Attack, Fortify while indicating invalidity for others. This state encompasses a set of states, providing shared behavior among them. All states within this set must inherit from this class.

f. Phase: This class manages the different states within the State Pattern in the game. It serves as a base class for various game phases, providing common functionality and defining abstract methods to be implemented by concrete phase classes.

g. Play: It defines the behavior for commands that are valid during gameplay phases and signifies invalid commands for other phases. Instances of this class represent a group of gameplay states and provide common behavior for all states within this group. All states within this group must extend this class.

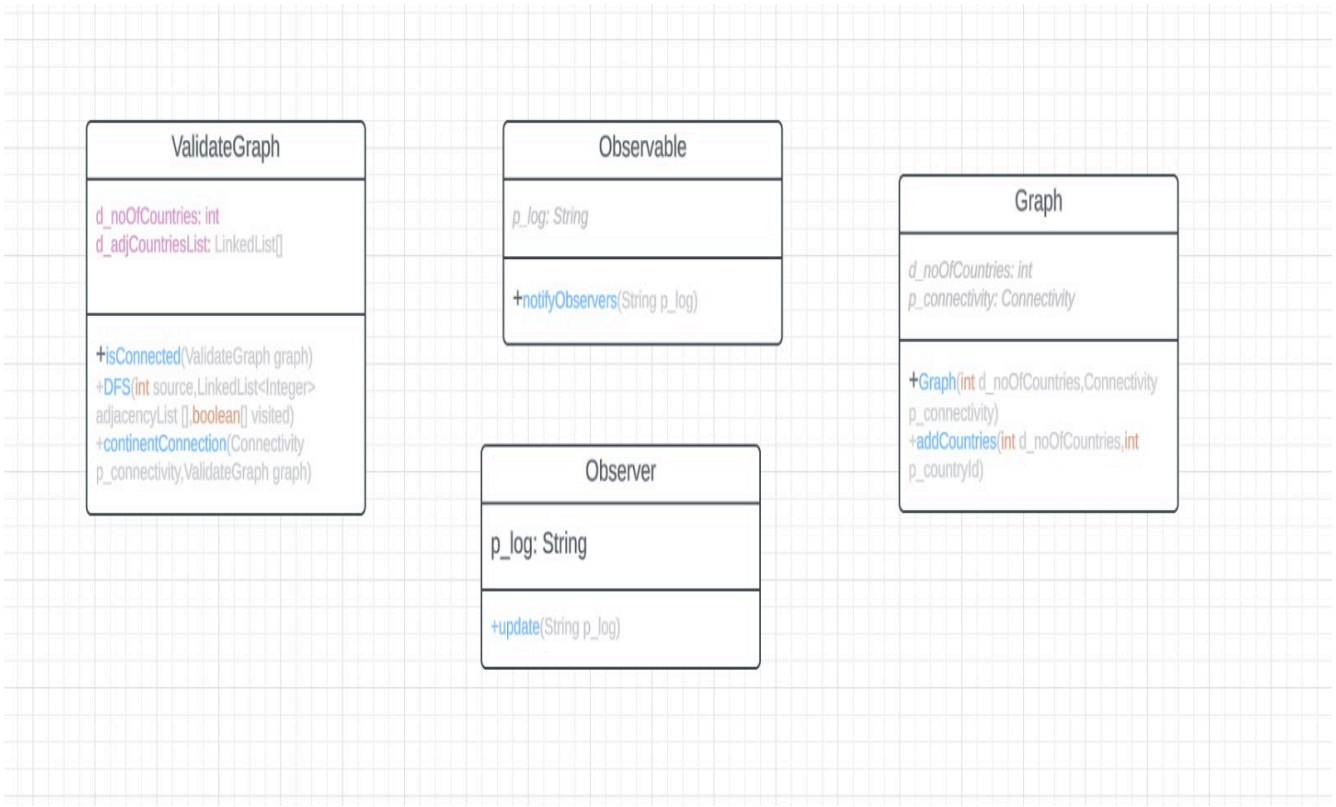
h. PlaySetup: It represents the setup phase of the game where players can add or remove players, assign countries, and begin the game.
i. PostLoad: This class represents the phase after loading a map. Allows the user to save the map and proceed to game play phases.
j. PreLoad: The class represents the phase before loading a map. Allows the user to load, edit, and validate the map before entering the PostLoad phase.
k. Reinforcement: This class represents the game's reinforcement phase. Allows players to deploy armies and proceed to the Attack phase.

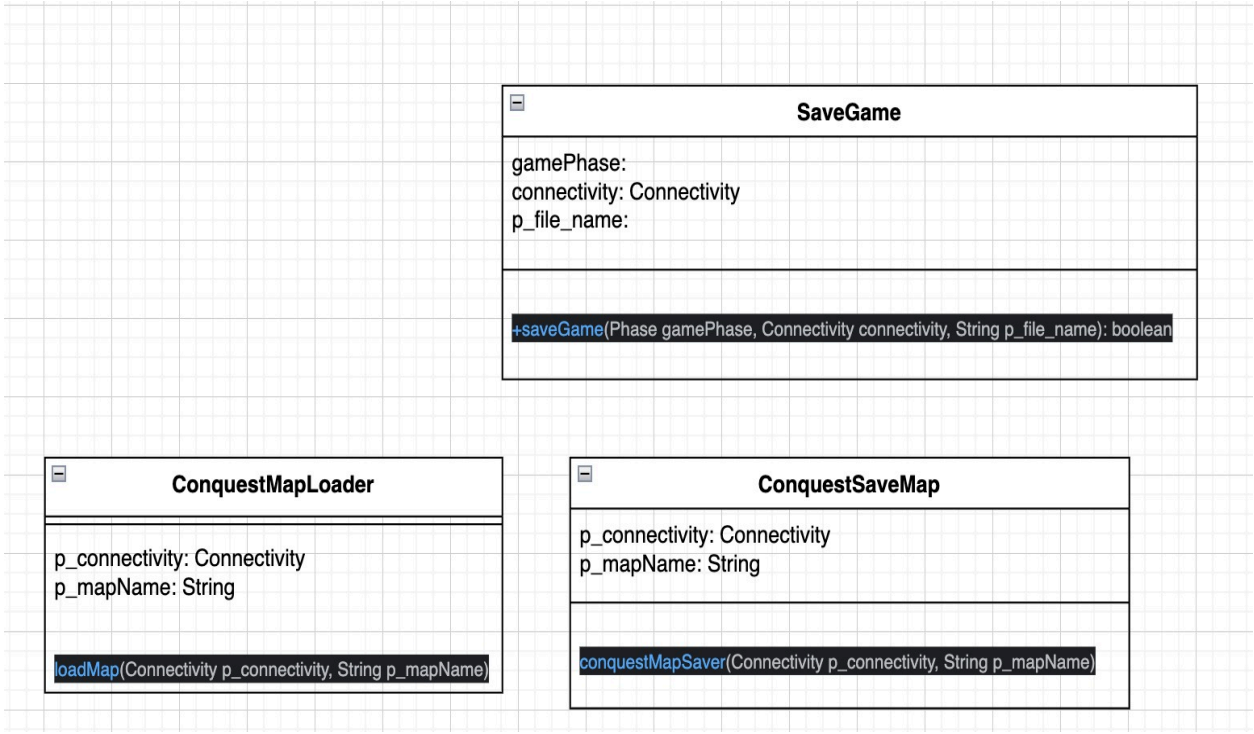
4. Strategy



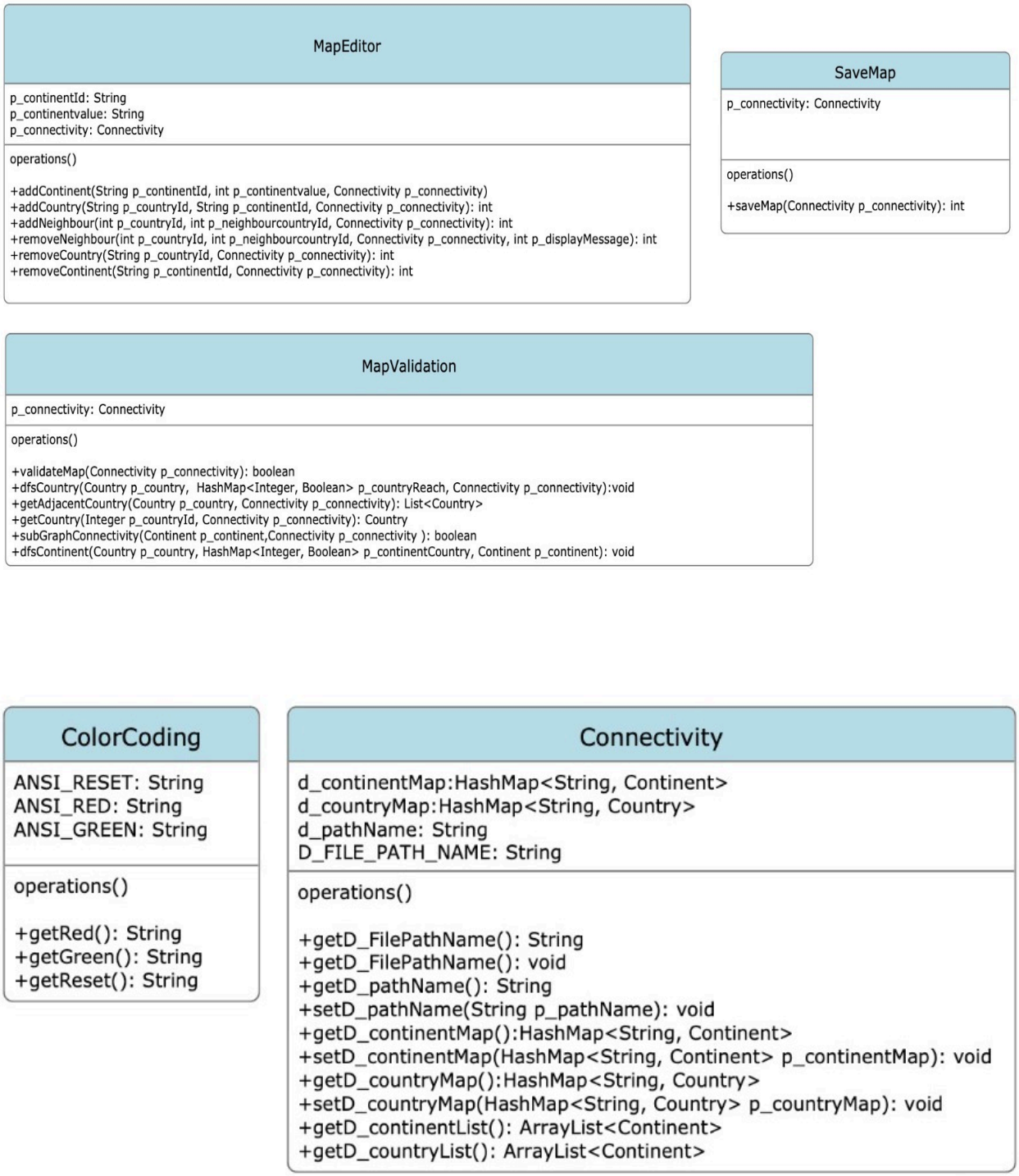
a. **AggressivePlayerStrategy**: The AggressivePlayerStrategy class encapsulates the behavior of an aggressive player strategy in the game. It focuses on deploying armies, launching attacks, and fortifying positions aggressively to exert pressure on opponents and expand territories.
b. **BenevolentPlayerStrategy**: The BenevolentPlayerStrategy class encapsulates the behavior of a benevolent player strategy in the game. It focuses on reinforcing the weakest countries owned by the player and does not engage in aggressive actions such as attacking or moving armies for expansion. Instead, it prioritizes strengthening vulnerable territories to ensure their defense.
c. **CheaterPlayerStrategy:** The CheaterPlayerStrategy class implements a strategy where the player cheats by conquering all immediate neighboring enemy countries and doubles the number of armies on its countries that have enemy neighbors. This strategy does not engage in traditional game actions like attacking, defending, or moving armies strategically; instead, it exploits the game mechanics to gain an advantage.
d. **HumanPlayerStrategy**: The HumanPlayerStrategy class represents a player strategy that involves user interaction for decision-making during gameplay. It allows human players to input their orders via the console, which are then processed and executed in the game. This strategy does not engage in traditional game actions like attacking, defending, or moving armies strategically; instead, it relies on direct user input for decision-making.
e. **PlayerStrategy**: The PlayerStrategy abstract class serves as a blueprint for implementing various player strategies in the game. It defines the common interface for all player strategies, including methods for creating orders, attacking, selecting attack origins, moving armies, and defending. Subclasses of PlayerStrategy must provide concrete implementations for these methods according to their specific strategy. Each concrete strategy subclass will override these methods to encapsulate its unique behavior and decision-making process within the game.
f. **RandomPlayerStrategy:** The RandomPlayerStrategy class represents a player strategy that prioritizes decentralized actions without a specific target. It generates random orders during gameplay, such as deploying troops to random countries, and randomly attacking neighboring enemy territories. This strategy introduces unpredictability into the game, making it challenging for opponents to anticipate the random player's moves.

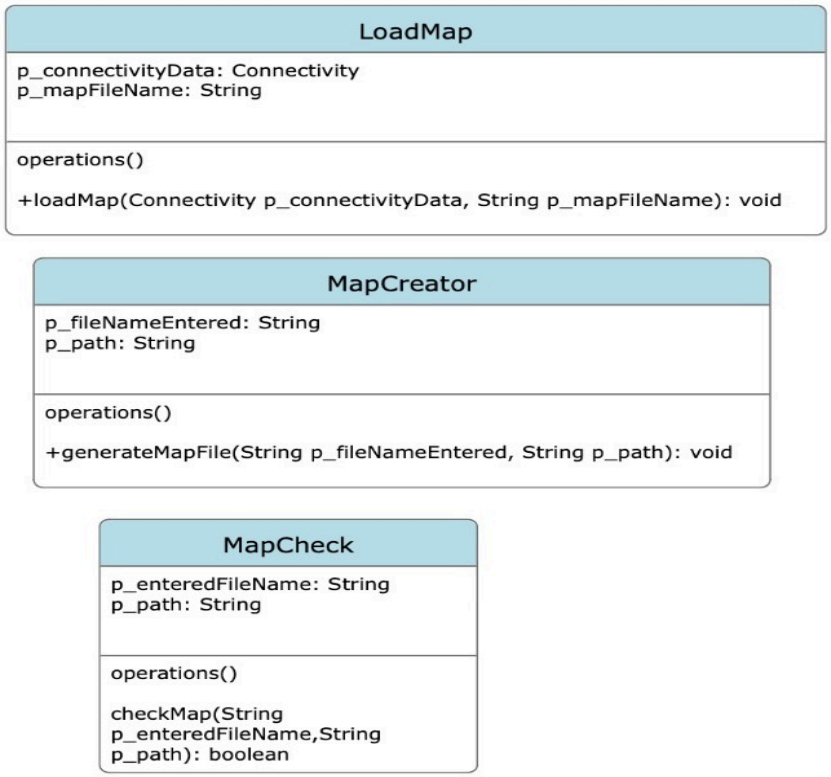
5. Utilities


ValidateGraph
d_noOfCountries: int
d_adjCountriesList: LinkedList[]
+isConnected(ValidateGraph graph)
+DFS(int source,LinkedList<Integer> adjacencyList [],boolean[] visited)
+continentConnection(Connectivity p_connectivity,ValidateGraph graph)
Observable
p_log: String
+notifyObservers(String p_log)
Graph
d_noOfCountries: int
p_connectivity: Connectivity
+Graph(int d_noOfCountries,Connectivity p_connectivity)
+addCountries(int d_noOfCountries,int p_countryId)
Observer
p_log: String
+update(String p_log)


SaveGame
gamePhase:
connectivity: Connectivity
p_file_name:
+saveGame(Phase gamePhase, Connectivity connectivity, String p_file_name): boolean
ConquestMapLoader
p_connectivity: Connectivity
p_mapName: String
loadMap(Connectivity p_connectivity, String p_mapName)
ConquestSaveMap
p_connectivity: Connectivity
p_mapName: String
conquestMapSaver(Connectivity p_connectivity, String p_mapName)


PlayersGamePlay
l_armiesOfPlayers: ArrayList<Integer>
l_neutralCountry: ArrayList<Country>
l_winner: Player
+assigncountries(ArrayList<Player> p_playersArray,ArrayList<Country> p_countryList,ArrayList<Continent> p_continentList): String
+assignArmiesToPlayers(ArrayList<Player> p_playersList): void
+showPlayersCountry(Player p_player,int p_Displayflag): ArrayList<String>
+checkArmyAvailable(int p_army,Player p_player): boolean
+advance(Player p_player,ArrayList<Player> p_playersArray,Country p_fromCountry,Country p_toCountry,int p_troops,ArrayList<Continent> p_continent,Connectivity p_connectivity, int fortify_flag): int
+findPlayerWithCountry(ArrayList<Player> p_playersArray, Country p_Country): Player
+attack(Player p_player,ArrayList<Player> p_playersArray,Country p_fromCountry,Country p_toCountry,int p_troops,ArrayList<Continent> p_continent,Connectivity p_connectivity):int
+removeCountry(ArrayList<Player> p_playersArray,Country p_country,ArrayList<Continent> p_continent):int
+bomb(Player p_player, ArrayList<Player> p_playersArray,Country p_toCountry,ArrayList<Continent> p_continent): int
+updateContinent(ArrayList<Player> p_playersArray, ArrayList<Continent> p_continentList):int
+AirliftDeploy(String p_sourceCountryObj, String p_targetCountryObj, int p_armiesToAirlift, Player p_player):int
+Blockade(String p_sourceCountryObj,Player p_player,ArrayList<Player> p_playersArray,ArrayList<Continent> p_continent):boolean
+generateCard():String
+winnerPlayer(ArrayList<Player> p_players,Connectivity p_connectivity):String
+negotiate(Player p_player,ArrayList<Player> p_playersArray, String p_toPlayerID): String
+resetDiplomacy(ArrayList<Player> p_playersArray): boolean


MapEditor
p_continentId: String
p_continentvalue: String
p_connectivity: Connectivity
operations()
+addContinent(String p_continentId, int p_continentvalue, Connectivity p_connectivity)
+addCountry(String p_countryId, String p_continentId, Connectivity p_connectivity): int
+addNeighbour(int p_countryId, int p_neighbourcountryId, Connectivity p_connectivity): int
+removeNeighbour(int p_countryId, int p_neighbourcountryId, Connectivity p_connectivity, int p_displayMessage): int
+removeCountry(String p_countryId, Connectivity p_connectivity): int
+removeContinent(String p_continentId, Connectivity p_connectivity): int
SaveMap
p_connectivity: Connectivity
operations()
+saveMap(Connectivity p_connectivity): int
MapValidation
p_connectivity: Connectivity
operations()
+validateMap(Connectivity p_connectivity): boolean
+dfsCountry(Country p_country, HashMap<Integer, Boolean> p_countryReach, Connectivity p_connectivity):void
+getAdjacentCountry(Country p_country, Connectivity p_connectivity): List<Country>
+getCountry(Integer p_countryId, Connectivity p_connectivity): Country
+subGraphConnectivity(Continent p_continent,Connectivity p_connectivity): boolean
+dfsContinent(Country p_country, HashMap<Integer, Boolean> p_continentCountry, Continent p_continent): void
ColorCoding
ANSI_RESET: String
ANSI_RED: String
ANSI_GREEN: String
operations()
+getRed(): String
+getGreen(): String
+getReset(): String
Connectivity
d_continentMap:HashMap<String, Continent>
d_countryMap:HashMap<String, Country>
d_pathName: String
D_FILE_PATH_NAME: String
operations()
+getD_FilePathName(): String
+getD_FilePathName(): void
+getD_pathName(): String
+setD_pathName(String p_pathName): void
+getD_continentMap():HashMap<String, Continent>
+setD_continentMap(HashMap<String, Continent> p_continentMap): void
+getD_countryMap():HashMap<String, Country>
+setD_countryMap(HashMap<String, Country> p_countryMap): void
+getD_continentList(): ArrayList<Continent>
+getD_countryList(): ArrayList<Continent>



a. Connectivity: This class serves as a medium to transfer data from one part of the map to another and binds it. It maps continent and countries names to their objects allowing a smooth gameplay.
b. LoadMap: Responsible for loading the map selected by the user.
c. MapCheck: Responsible for checking whether the map exists in the directory path or not.
d. MapCreator: Responsible for automatically generating a new map when the user asks for it.
e. MapEditor: The MapEditor class facilitates the adding and removing of countries, continents and neighbors in an existing map.
f. MapValidation: It is responsible for validation of the entire map to check for correctness.
g. PlayersGameplay: Facilitates the entire gameplay from the players perspective of assigning

armies to check its availability to deploy, thereby functioning a smooth game process as per the structure of the game.

h. SaveMap: Responsible for saving map after the user has created a new map or modified the existing ones.

i. ColorCoding: It provides ANSI escape codes for displaying texts in red, green and setting it to default.

j. Graph: Responsible for representing a graph of countries, initializing it, and adding countries and their connections.

k. Observer: The Observer interface defines a standard way for objects to observe and update log messages.

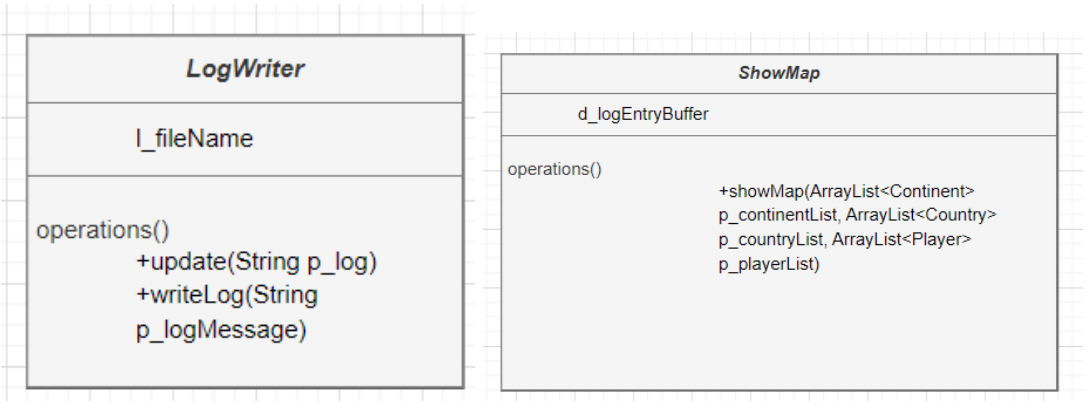
l. Observable: interface defines a standard way for objects to notify observers about changes by using the notifyObservers() method.

m. SaveGame: Saving the current game state to a file.

n. ConquestMapLoader: Responsible for loading the map by the user in conquest format.

o. ConquestSaveMap: Responsible for saving the map by the user in conquest format.

6. View



a. **LogWriter**: The LogWriter class is responsible for writing log messages to a log file. It implements the Observer interface, allowing it to receive log messages from observable objects. When it receives a log message through the update method, it writes the message to the log file using the writeLog method. The log file is stored in a directory named "logFiles", and each log message is appended to the file named "LogEntry.log". If the directory or file doesn't exist, it creates them.

b. **ShowMap**: The ShowMap class provides functionality to display the game map, including continents, countries, neighbors, army count, and country owner. It encapsulates the logic to construct and print the map details, facilitating clear visualization of the game map for the users.