Advanced Programming Practices SOEN 6441- Winter 2024

RISK GAME

PROJECT BUILD 1

Submitted By :-

Sanjay Bhargav Pabbu- 40265895

Blesslin Jeba Shiny Augustin Moses - 40266442

Susmitha Mamula- 40277873

Poojitha Bhupalli- 40232528

Piyush Gupta- 40294464

Mahfuzzur Rahman - 40293992

Objective

To create a Java application that complies with the "Warzone" version of the Risk game's rules, map files, and command-line play capabilities.

The game is divided into three phases: launch, main gameplay, and execute, which manages the whole process. Users of the game can import pre-existing maps, make new ones, and modify existing ones to add or remove continents, nations, nearby nations, and players.

Players in the main game loop would be assigned countries to issue and carry out orders.

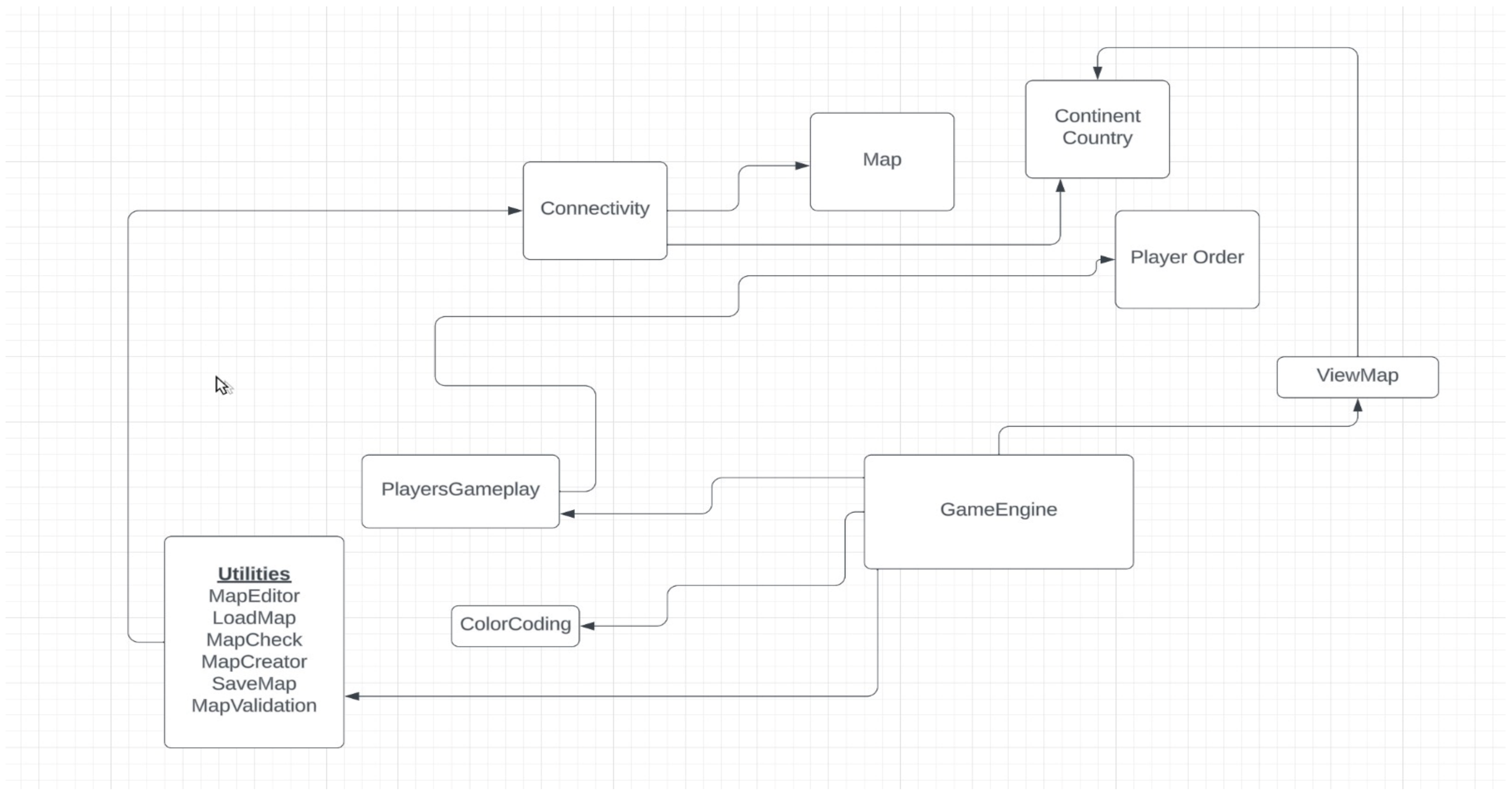
Until they place all of their armies, players issue deploy orders in a round robin fashion. In the execute phase, player orders to deploy armies to a country's CountryID are carried out.

BUILD 1

Flow of Execution

Load	Load Map and Show Map
Edit	Edit Map
Add	Add/remove Players
Assign	Assign countries and reinforcement to players
Deploy	Deploy all the reinforcement – Issue order phase
Execute	Execute Phase

Architecture Diagram



MVC Architecture

Our project follows MVC architecture to organize the structure of the game

Model : Includes data for Country, Continent, Player, Order and Map containing state and behavior that can be accessed across entire architecture of the project.

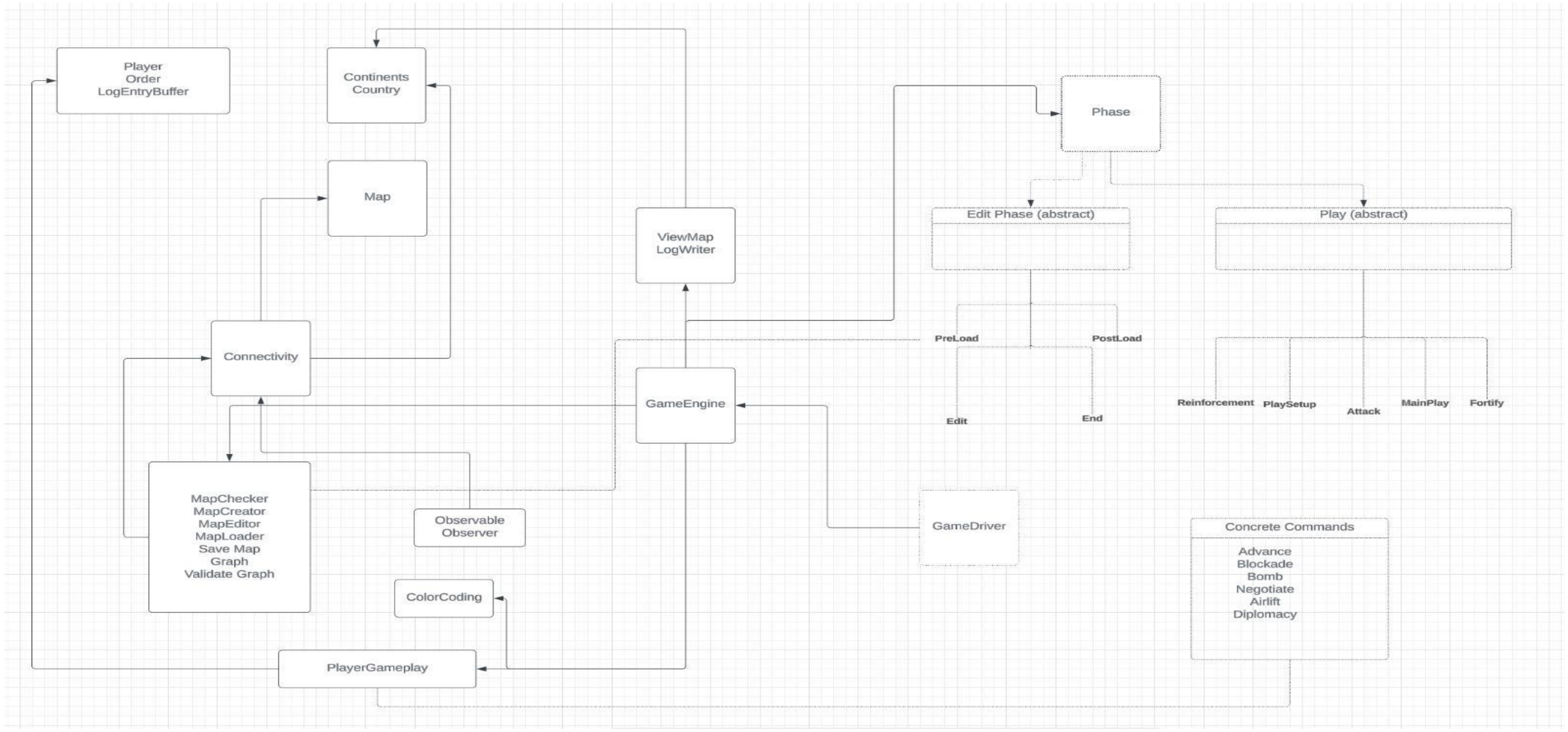
Utilities: Contains methods to perform startup, main game loop and execute phase.

View: Used to show the existing or updated map to the users in console. ShowMap displays Country, Continent, Neighboring countries and Total number of armies in tabular format.

Controller: Acts as an intermediary between Model and View which takes user inputs via the View, processes it by interacting with Model and then updates the view to reflect the result. GameEngine is the center of the game that controls the entire functionality of the game

BUILD 2

Architecture Diagram



State Pattern

The game program structure has been modified to adopt state pattern as mentioned in the table below:

PHASE	COMMAND
Any phase	showmap
Edit:Preload	loadmap, editcountry, editcontinent, editneighbor, validatemap
Edit:PostLoad	savemap
Play:PlaySetup	gameplayer, assigncountries
Play:MainPlay:Reinforce	deploy
Play:MainPlay:Attack	advance, bomb, airlift, blockade, negotiate
Play:MainPlay:Fortify	fortify
End	end game
Any	next phase

State Pattern

State pattern for the game is designed to perform in a way described below:

- **Showmap** command can be used at any phase for to view the modified map at regular intervals.
- **Preload** phase allows user to load the map, perform map edits such as editcountry, editcontinent, editneighbor and validate the map.
- In order to move to **PlaySetup , Postload** phase wants user to save the map.
- In **PlaySetup** phase, user can add/remove players and assign countries.
- Once the countries are assigned in PlaySetup phase, game goes to next phase which is **Reinforce** where it allows users to deploy armies to the countries for every added player in round robin fashion.
- Next phase is **Attack** where players can advance/attack or use their cards. If winner is not declared, the game gets passed to fortify phase
- **Fortify** phase routes user back to Reinforcement to deploy newly added troops.
- **End** Phase ends the game with a thank you message to the user. This phase is set on exit or when the winner is declared.

Observer Pattern

The project uses observer pattern for logging so that whenever a command is executed, or an order is issued or executed , a LogEntryBuffer object is filled with information about the effect of the action:

- The Observable class is LogEntryBuffer. It notifies observers to update its state whenever the file is changed.
- Observer LogWriter writes the content of the LogEntryBuffer to a log file when it is changed(when notified by observer).
- This allows users to see all the actions happened during the game along with the phases that was executed in the game.

Command Pattern

The project utilizes the Command pattern to encapsulate user commands as objects, thereby abstracting the command processing from the command sender to promote loose coupling between game components.

This involves creating an "Order" class, serving as a command class to encapsulate user commands as objects, with an "execute()" method defining the action to be taken when the command is invoked.

The "Player" class acts as the invoker, executing commands from "Order" and managing their execution. It serves as an intermediary between the client and the command objects.

As for the client class, "GameEngine" is responsible for configuring commands to determine when and how to execute them. It retrieves orders from players using the "next_order()" method and executes them by calling the "execute()" method of "Order" during the Reinforcement phase.

Cards Implementation

When a player successfully conquers a country through a successful attack, cards are randomly assigned to them. The implementation of the bomb card is provided in the "PlayersGamePlay.java" file, which will be executed in the "Order" class following the command pattern.

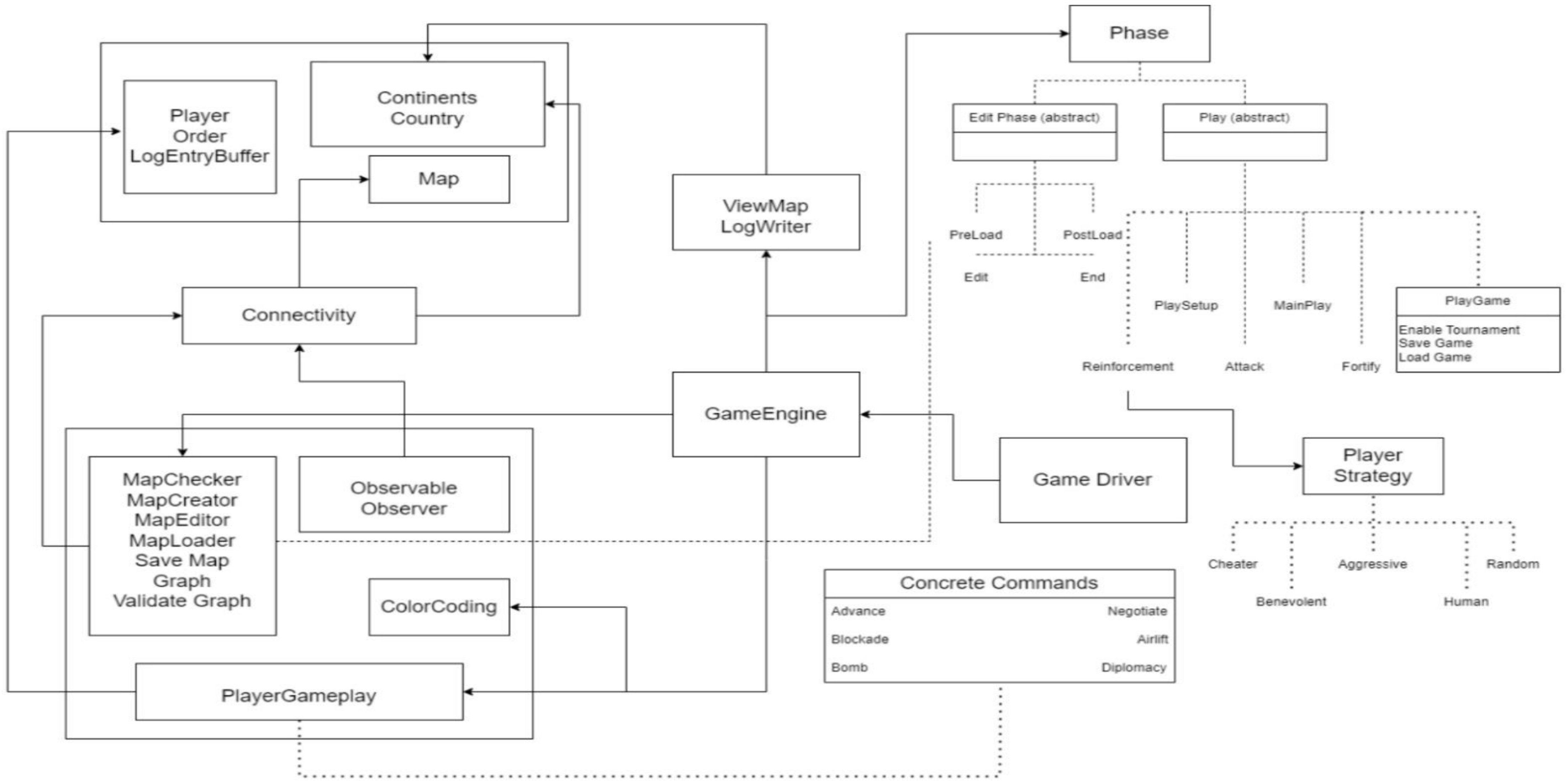
The cards that can be assigned include the following:

- Bomb: This card is used to eliminate half of the armies located on an opponent’s country adjacent to one of the current player’s countries.
- Blockade: This card triples the number of armies on one of the current player’s countries and converts it into a neutral territory.
- Airlift: It allows the player to move some armies from one of their countries to any other country.
- Negotiate: This card prevents attacks between the current player and another player until the end of the turn.

After executing the card for the player, it is removed from the player’s array of cards.

BUILD 3

Architecture Diagram



Strategy Pattern

PlayerStrategy (Interface/Abstract Class):

Objective: Defines the structure for various player strategies.

Action: issueOrder(GameMap map): Represents the player's move based on the selected strategy.

HumanPlayerStrategy (Concrete Class):

Objective: Implements the PlayerStrategy for a human participant.

Action: issueOrder(GameMap map): Enables user interaction for decision-making during gameplay.

AggressivePlayerStrategy (Concrete Class):

Objective: Implements the PlayerStrategy for a computer player with an aggressive approach.

Action: issueOrder(GameMap map): Prioritizes consolidating forces, attacking with the strongest country, and optimizing army movements.

Strategy Pattern

BenevolentPlayerStrategy:

Definition: Implements the PlayerStrategy for a benevolent computer player.

Function: issueOrder(GameMap map): Prioritizes protecting weaker countries by reinforcing them.

RandomPlayerStrategy:

Definition: Implements the PlayerStrategy for a random computer player.

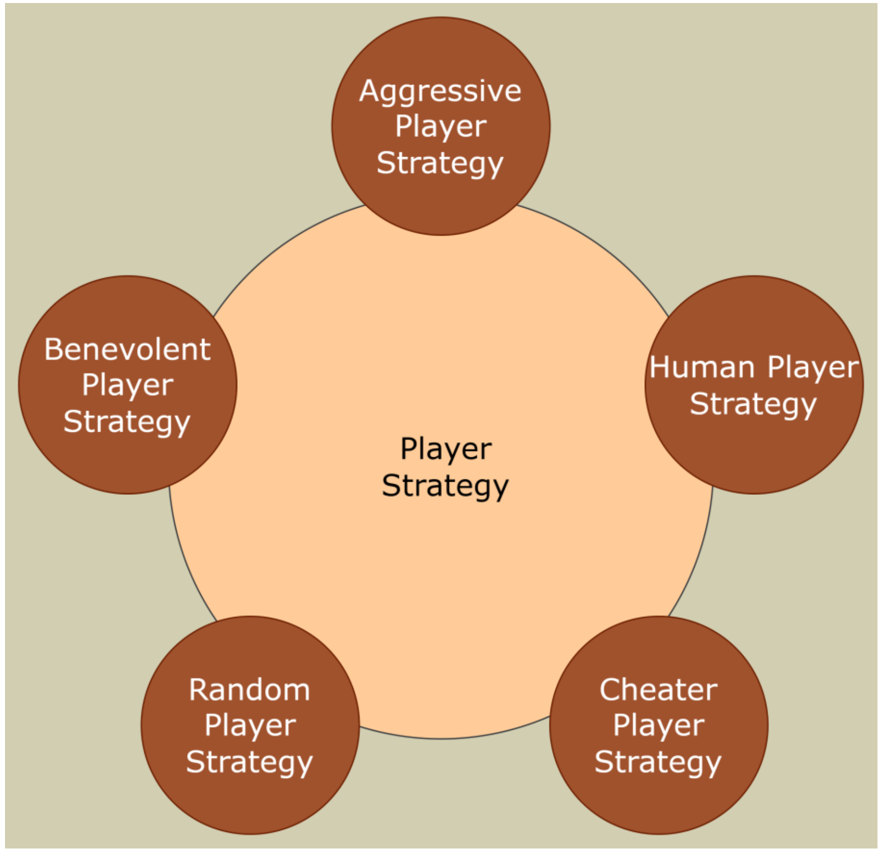
Function: issueOrder(GameMap map): Deploys armies randomly, attacks neighboring countries randomly, and moves armies randomly.

CheaterPlayerStrategy:

Definition: Implements the PlayerStrategy for a cheater computer player.

Function: issueOrder(GameMap map): Conquers all immediate neighboring enemy countries and doubles armies based on specific conditions, directly manipulating the map.

Strategy Pattern



- The strategy pattern facilitates:
- Flexibility
- Encapsulation
- Easy maintenance
- Run-Time Switching of Strategies:
- Code Reusability

Adapter Pattern

To implement the Adapter Pattern:

1. Define the Target interface that aligns with the client code's expectations when handling map files.
2. Create the Adaptee class, which corresponds to the existing "domination" file reader.
3. Implement the Adapter class, which adapts the functionality of the "conquest" file reader to conform with the Target interface.
4. Adjust the client code to instantiate either the original file reader or the adapter according to the user's preference.
5. Enable the user to specify the desired output file format when saving the map.

Adapter Pattern

The CheckMap() method prompts the user to input the map name, then determines its type, whether it's a domination map or a conquest map.

ConquestMapLoader is a class designed for loading conquest-type maps. It contains a method, loadMap(), utilized to load the selected map, whether it's pre-defined or user-created.

MapLoader class is responsible for loading and validating domination-type maps (Adaptee Class).

The SaveMap() function is employed to save the map. If it's a domination map, it gets saved as usual. However, for conquest maps, the conquestMapSaver() function is utilized.

Javadoc

Javadoc documentation on all the classes, data members, and member functions has been implemented to generate comprehensive and easily navigable documentation in HTML format.

Syntax:

```
/**
 * Javadoc comment
 *
 */
```

@param	provide information about method parameter or the input it takes
@see	generate a link to other element of the document
@version	provide version of the class, interface or Enum.
@return	provide the return value
@author	Describes an author
@throws	Throws exception if something goes wrong

Junit

Include over 50 test cases to cover essential scenarios in the game:

1. Evaluate the accuracy of the "assigncountries" functionality.
2. Verify the effectiveness of editing map commands, encompassing addition/removal of countries, continents, and neighbors.
3. Ensure precise allocation of armies to players.
4. Validate maps and continents as connected graphs.
5. Handle scenarios involving reading an invalid map file.
6. Confirm correctness of startup phase functionality.
7. Validate tournament phases to ensure proper progression.
8. Calculate reinforcement army numbers accurately.
9. Test various order validations upon execution, covering source/target validation for all orders, conquering countries, moving armies within conquered countries, and triggering end-game conditions upon conquering all countries.
10. Assess the accuracy of player strategies like aggressive and benevolent strategies.

Thank You