

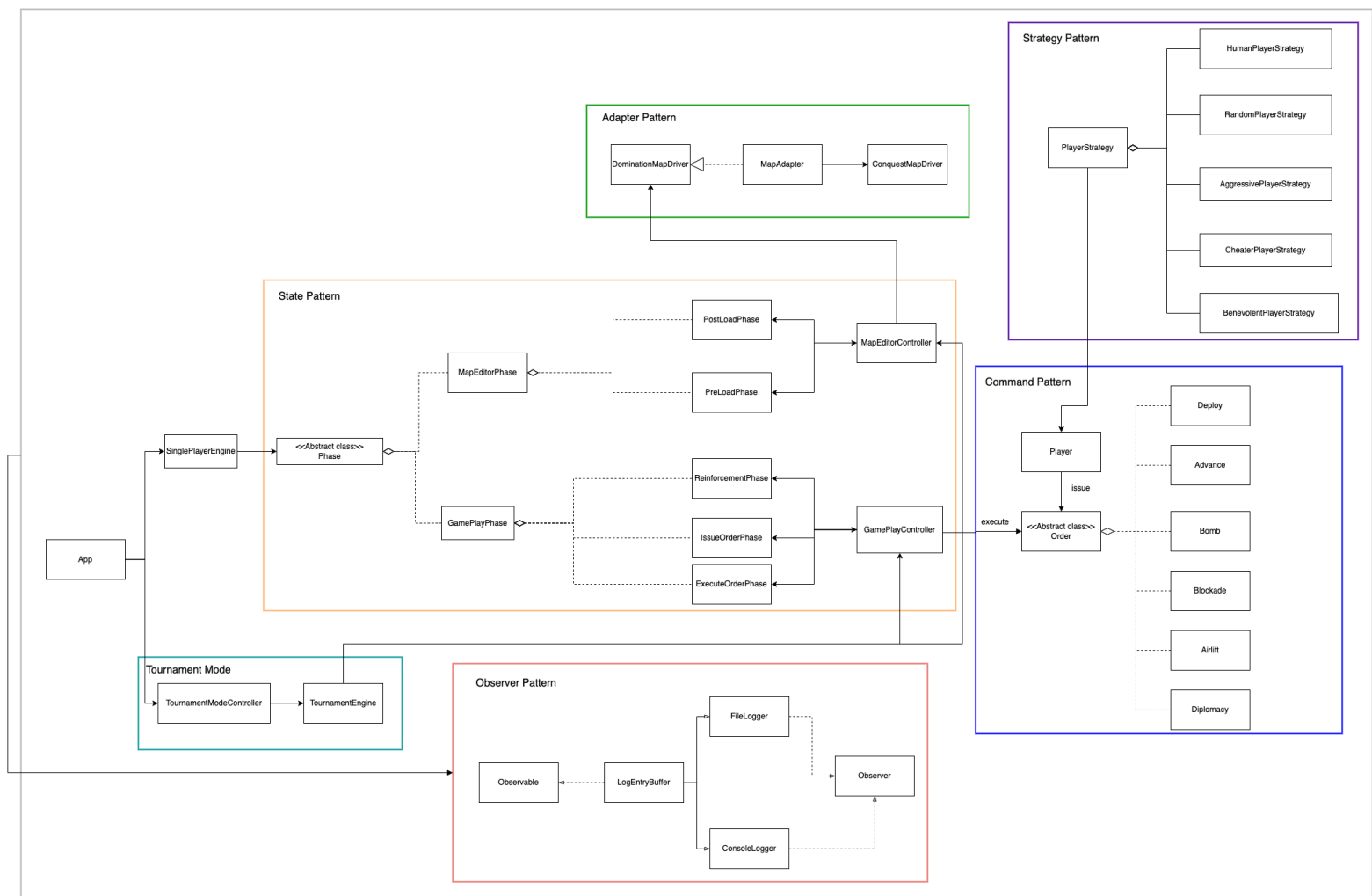
SOEN 6441 Advanced Programming Practices

ARCHITECTURAL DIAGRAM

Group W10 – Build 3

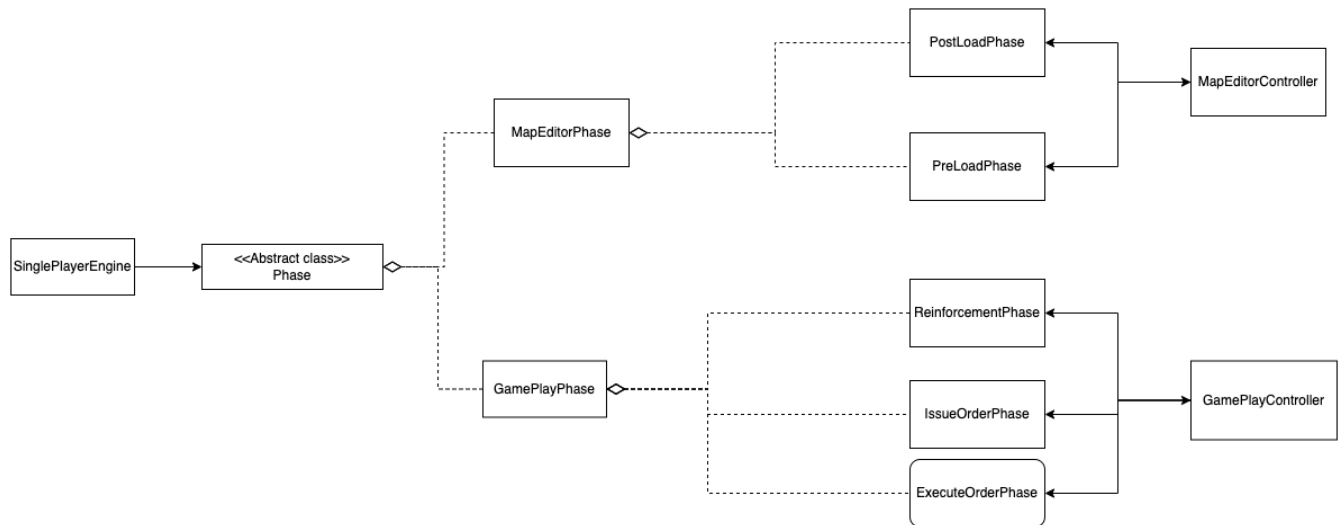
Group members:

- Omnia Alam
- Yajing Liu
- Sherwyn Dsouza
- Darlene Nazareth
- Duy Thanh Phan
- Md Tazin Morshed Shad



As shown in the architectural design diagram, our project is divided into six main parts: state pattern, command pattern, observer pattern, strategy pattern, adapter pattern, and tournament

mode. Specifically, the state pattern is to implement the phase of the application. The command pattern is to implement the orders. The observer pattern is to implement a game log file. The strategy pattern is to implement different computer player behaviors. Tournament mode allows games to run automatically. The details of these six parts will be shown in the following sections.



State Pattern

State pattern is to implement the phase of the application.

The state pattern contains the following classes:

- **Phase**: this class is an abstract class.
- **MapEditorPhase**: this class is an abstract class and extends the Phase class.
- **GamePlayPhase**: this class is an abstract class and extends the Phase class.
- **PreLoadPhase**: this class is a concrete class and extends the MapEditorPhase class.
- **PostLoadPhase**: this class is a concrete class and extends the MapEditorPhase class.
- **ReinforcementPhase**: this class is a concrete class and extends the GamePlayPhase class.
- **IssueOrderPhase**: this class is a concrete class and extends the GamePlayPhase class.
- **OrderExecutionPhase**: this class is a concrete class and extends the GamePlayPhase class.

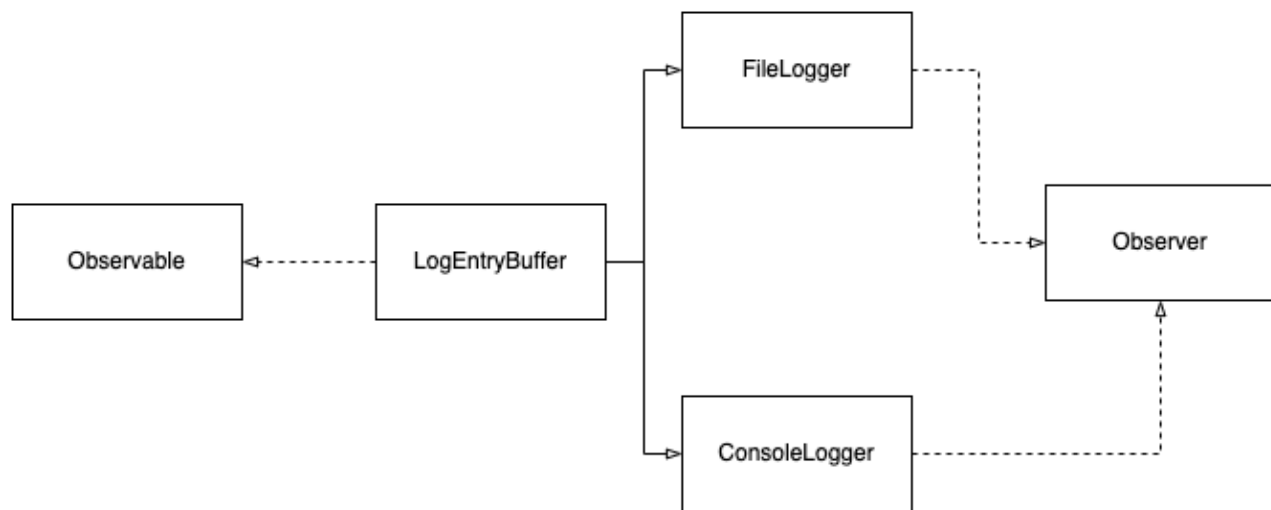
Advantages of using State Pattern:

1. It allows for cleaner, more modular, and maintainable code by separating different states into distinct classes and providing a structured way to transition between them.
2. Each command is separated by phases.
3. If the command does not belong to the phase, it does not execute within that phase and shows an invalid command as an error message.
4. It dictates the `SinglePlayerEngine` to run commands in a sequential manner for example: `editcountry` command does not run once we are in `GamePlayPhase`

Work-flow of the State pattern:

1. In the beginning, the SinglePlayerEngine will set phase to Pre-load.
2. Only loadmap command is valid in this phase.
3. After loading the map, the phase will automatically be set to Post-load.
4. In the post-load, only the map editor commands are valid.
5. Once the game map is ready, after editing, the next command will set the phase to PlaySetup, which is inside the GamePlayPhase.
6. In the Playsetup phase, we can add players and assign countries.
7. Once the assigncountries command is placed, the Phase will change to ReinforcementPhase, which is included in the GamePlayPhase.
8. In this phase, we can use reinforcement commands.
9. Then the phase will be set as IssueOrder Phase.
10. After issuing orders and committing, the phase will be set as ExecuteOrderPhase.

Observer Pattern

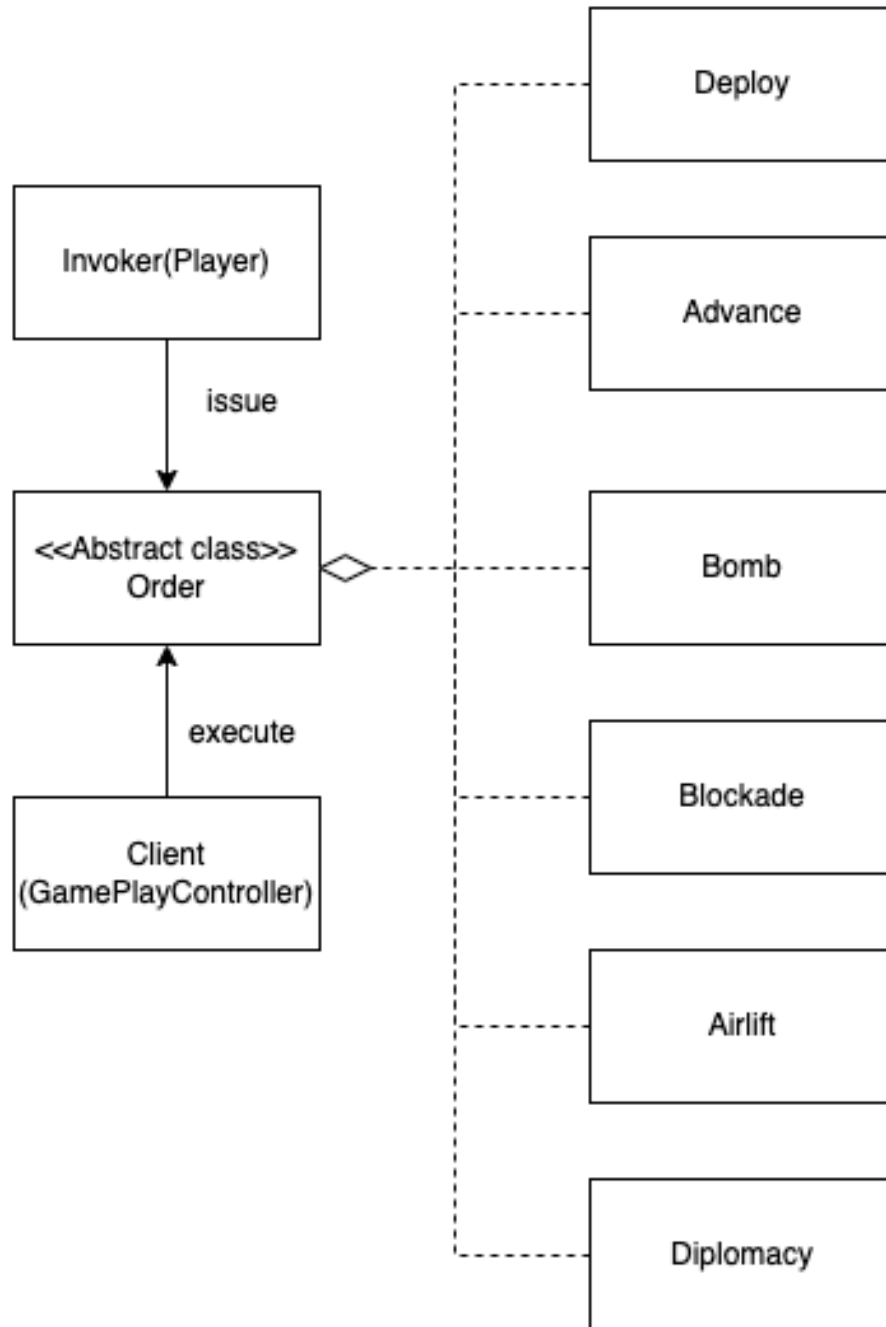


Observer pattern is to implement a game log file.

This observer pattern contains the following classes:

- **Observable**: This is an interface.
- **Observer**: This is an interface
- **LogEntryBuffer**: This is a concrete class that implements **Observable** interface.
- **FileLogger**: This is a concrete class that implements **Observer** interface.
- **ConsoleLogger**: This is a concrete class that implements **Observer** interface.

Command Pattern



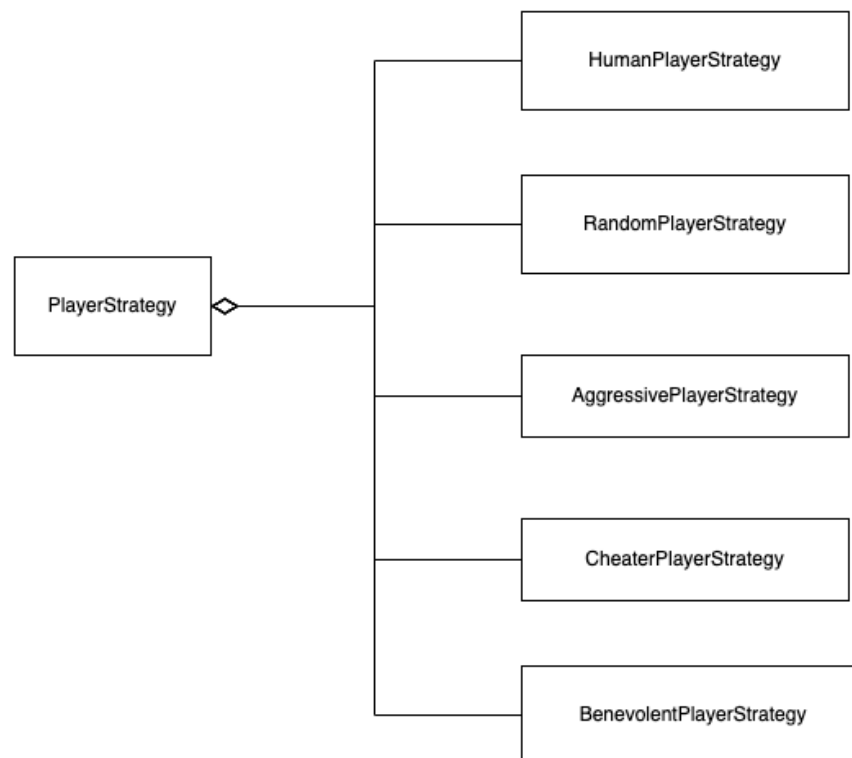
This command pattern is to implement the Orders.

This command pattern contains the following classes:

- **Player**: This is a concrete class to create orders by using the `issueOrder()` method.
- **GamePlayController**: This is a concrete class to execute orders by using `execute()` method of the **Order**.
- **Order**: This is an abstract class having an abstract `execute()` method.

- **Deploy:** This is a concrete class that extends the Order class and implements the concrete execute() method.
- **Advance:** This is a concrete class that extends the Order class and implements the concrete execute() method.
- **Bomb:** This is a concrete class that extends the Order class and implements the concrete execute() method.
- **Blockade:** This is a concrete class that extends the Order class and implements the concrete execute() method.
- **Airlift:** This is a concrete class that extends the Order class and implements the concrete execute() method.
- **Diplomacy:** This is a concrete class that extends the Order class and implements concrete execute() method.

Strategy Pattern:



The structure of the strategy pattern:

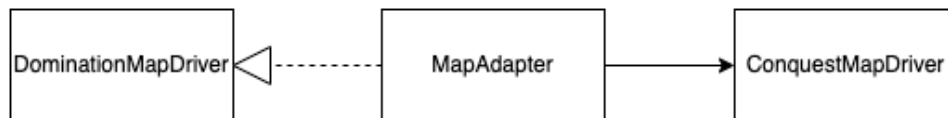
- PlayerStrategy is the abstract class with method issueOrder() for implementation classes to dictate the algorithm of issuing player orders.
- Since all the strategies are different from each other so each strategy class overrides the issueOrder() method from PlayerStrategy in order to implement different strategies.
- The Player class has an extra attribute to get or set the strategy name.

- Based on the input of the user the strategies are assigned to the player.

The strategy pattern contains the following classes:

- **PlayerStrategy**: This class is an abstract class that provides a framework for implementing different strategies for a player in a game.
- **HumanPlayerStrategy**: This class extends the PlayerStrategy class. This class serves human player and asks human to input new orders to issue orders.
- **RandomPlayerStrategy**: This class extends the PlayerStrategy class. This computer player strategy deploys on random country, attacks random neighboring countries and moves armies randomly between its countries.
- **AggressivePlayerStrategy**: This class extends the PlayerStrategy class. This class represents a strategy for a player in a game, where the player deploys armies on their strongest country, attacks neighboring countries with their strongest country, and moves armies to maximize force.
- **CheaterPlayerStrategy**: The CheaterPlayerStrategy class is a subclass of PlayerStrategy that implements a strategy where the player conquers all neighboring countries and doubles the armies in countries with enemy neighbors.
- **BenevolentPlayerStrategy**: The BenevolentPlayerStrategy class is a subclass of PlayerStrategy that implements a strategy where the player issues deploy and advance orders to their weakest country.

Adapter Pattern:

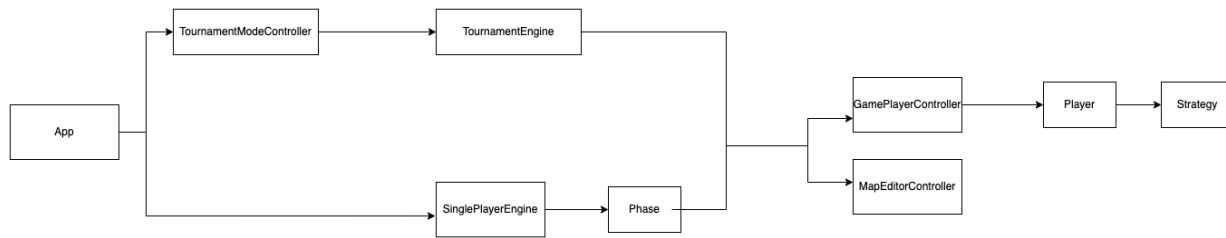


Adapter pattern supports users to read and save in Conquest game map format or Domination game map format.

This adapter pattern contains the following classes:

- **DominationMapDriver**: This class reads a map file and initialize into the Country and Continents. This is used to populate the GameMap.
- **MapAdapter**: This class acts an adapter between the DominationMapDriver and ConquestMapDriver classes, allowing for the translation of game maps between the two formats.
- **ConquestMapDriver**: The ConquestMapDriver class is responsible for loading and saving Conquest game maps in the Conquest file format.

Tournament Mode



We also developed a different mode - tournament mode in this build. Tournament mode is a different mode that can automatically run on different maps between the chosen computer player strategies. Users can choose single player mode or tournament mode at the beginning of games. If users choose the single player mode, the game will be handled by SinglePlayerEngine. If users choose the tournament mode, the game will be handled by TournamentModeController and TournamentEngine.

Project Structures

- ✧ **App:** This class initializes the application and asks users to choose single game mode or tournament mode.

Engines

- ✧ **SinglePlayerEngine:** The SinglePlayerEngine class is responsible for managing the game flow, handling user input, and executing commands in the Risk game.
- ✧ **TournamentEngine:** The TournamentEngine class is responsible for managing and running a series of games on different maps using a set of player strategies.

Controllers

- ✧ **MapEditorController:** This class holds the logic to update the game map.
- ✧ **GamePlayController:** This class is responsible for managing the players, issuing orders, executing orders and their interactions in a game.
- ✧ **TournamentModeController:** The TournamentModeController class is responsible for handling user input and executing commands in a tournament game mode.

Models

- ✧ **GameMap:** This class contains data and functions to retrieve and manipulate game map properties such as countries and continents
- ✧ **Continent:** This class represents a continent in a game, with properties continent ID, name, list of countries, list of neighboring countries, and bonus points.
- ✧ **Country:** This class represents a country in a game, with properties country ID, name, continent ID, army count, and owner.

- ✧ **Player:** This class represents a player in a game, with properties name, countries owned, orders, and leftover armies.
- ✧ **CardType:** This class is to define the cards can be used and give random cards when it is needed.
- ✧ **MatchResult:** The MatchResult class represents the result of a match, including the player's name, game count, and map.
- ✧ **ConquestGameMap:** The ConquestGameMap class represents a game map consisting of countries and continents, and provides methods to add countries and continents to the map.
- ✧ **DataStorage:** The DataStorage class is responsible for storing data for saving and loading the game.
- ✧ **Phase:** This class is an abstract class that represents a phase in a Risk game and provides methods for various game commands.

Phases

- **MapEditorPhase:** This class is an abstract class that extends the Phase class and provides default implementations for various methods related to map editing in a game.
- **GamePlayPhase:** This class is an abstract class that extends the Phase class and provides default implementations for various methods related to game play in a game.
- **PreLoadPhase:** This class is a subclass of MapEditorPhase that handles the pre-loading phase of a game, allowing the user to load a map file and transition to the next phase.
- **PostLoadPhase:** This class is a subclass of MapEditorPhase that represents the phase after loading a map in a game editor, allowing for editing and saving of the map.
- **PlaySetupPhase:** This class is a subclass of GamePlayPhase that handles the setup phase of the game, including creating and removing players, assigning countries, and transitioning to the main play phase.
- **ReinforcementPhase:** This is a subclass of GamePlayPhase that handles the reinforcement phase of the game.
- **IssueOrderPhase:** This class is a subclass of GamePlayPhase that represents the phase of the game where players issue orders.
- **ExecuteOrderPhase:** This class is a subclass of GamePlayPhase that represents the phase of the game where player orders are executed.

Strategies

- **PlayerStrategy:** This class is an abstract class that provides a framework for implementing different strategies for a player in a game.
- **HumanPlayerStrategy:** This class extends the PlayerStrategy class. This class services human player and asks human to input new orders to issue orders.

- **RandomPlayerStrategy:** This class extends the PlayerStrategy class. This computer player strategy deploys on random country, attacks random neighboring countries and moves armies randomly between its countries.
- **AggressivePlayerStrategy:** This class extends the PlayerStrategy class. This class represents a strategy for a player in a game, where the player deploys armies on their strongest country, attacks neighboring countries with their strongest country, and moves armies to maximize force.
- **CheaterPlayerStrategy:** The CheaterPlayerStrategy class is a subclass of PlayerStrategy that implements a strategy where the player conquers all neighboring countries and doubles the armies in countries with enemy neighbors.
- **BenevolentPlayerStrategy:** The BenevolentPlayerStrategy class is a subclass of PlayerStrategy that implements a strategy where the player issues deploy and advance orders to their weakest country

Utils

- ⊄ **CommandInterpreter:** This class provides methods to extract the main command and argument list from a given command input.
- ⊄ **Constants:** This class contains various constant values used throughout the code.
- ⊄ **Reinforcements:** This class contains methods to reinforce armies for each player.
- ⊄ **GamePlayHelper:** The GamePlayHelper class provides a static method to get a list of countries owned by other players that are neighbors of the countries owned by a given player.
- ⊄ **SaveLoad:** The SaveLoad class is responsible for saving and loading the game.
- ⊄ **Observable:** This is an interface for implementing Observable with notify Observers, attach and detach functions.
- ⊄ **Observer:** This is an interface used to implement the Observer design pattern, which allows objects to be notified of changes in the state of another object (observable object).

Loggers

- **LogEntryBuffer:** This class is a singleton implementation that acts as a buffer for log entries and notifies observers when new log entries are added.
- **FileLogger:** This FileLogger class implements the Observer interface that writes data to a file.
- **ConsoleLogger:** This class implements the Observer interface in Java that prints the given data when the update method is called.

Maps

- **MapEditor:** This class operates on the current game map, adding and removing game map elements.
- **MapDisplay:** This class populates and formats a table to display information about countries, continents, players, and neighboring countries on a game map.

- **MapValidator:** This class provides methods to validate a game map by checking for various conditions such as empty map, non-existent continents or neighbors, self-referencing neighbors, inaccessible countries, and disconnected continents.
- **DominationMapDriver:** This class reads a map file and initialize into the Country and Continents. This is used to populate the Gamemap.
- **MapAdapter:** This class acts as an adapter between the DominationMapDriver and ConquestMapDriver classes, allowing for the translation of game maps between the two formats.
- **ConquestMapDriver:** The ConquestMapDriver class is responsible for loading and saving Conquest game maps in the Conquest file format.

Commands

- ∉ **Order:** This class is an abstract class for order. It defines the basic behavior (execute) of an order.
- ∉ **Deploy:** This class is the deploy order class. It extends the Order class. It defines the specific executing behavior for deploying.
- ∉ **Advance:** This class is the Advance order class. It extends the Order class. It defines the specific executing behavior for advancing.
- ∉ **Bomb:** This class represents a player's order to bomb a specific country, reducing its army count by half.
- ∉ **Blockade:** This class represents a blockade order in a game, where a player blocks a country and triples its army count while removing it from their owned countries.
- ∉ **Airlift:** This class represents an order to move armies from one country to another in a game.
- ∉ **Negotiate:** This class is to negotiate with other players. In this turn, the two players can not attack each other.
- ∉ **Command:** This class is to define an enum for commands that can be used in the game.

Exceptions

- **ApplicationException:** This class is a custom exception class that wraps all checked standard Java exceptions and can be used to handle risk-game-specific errors in Java.