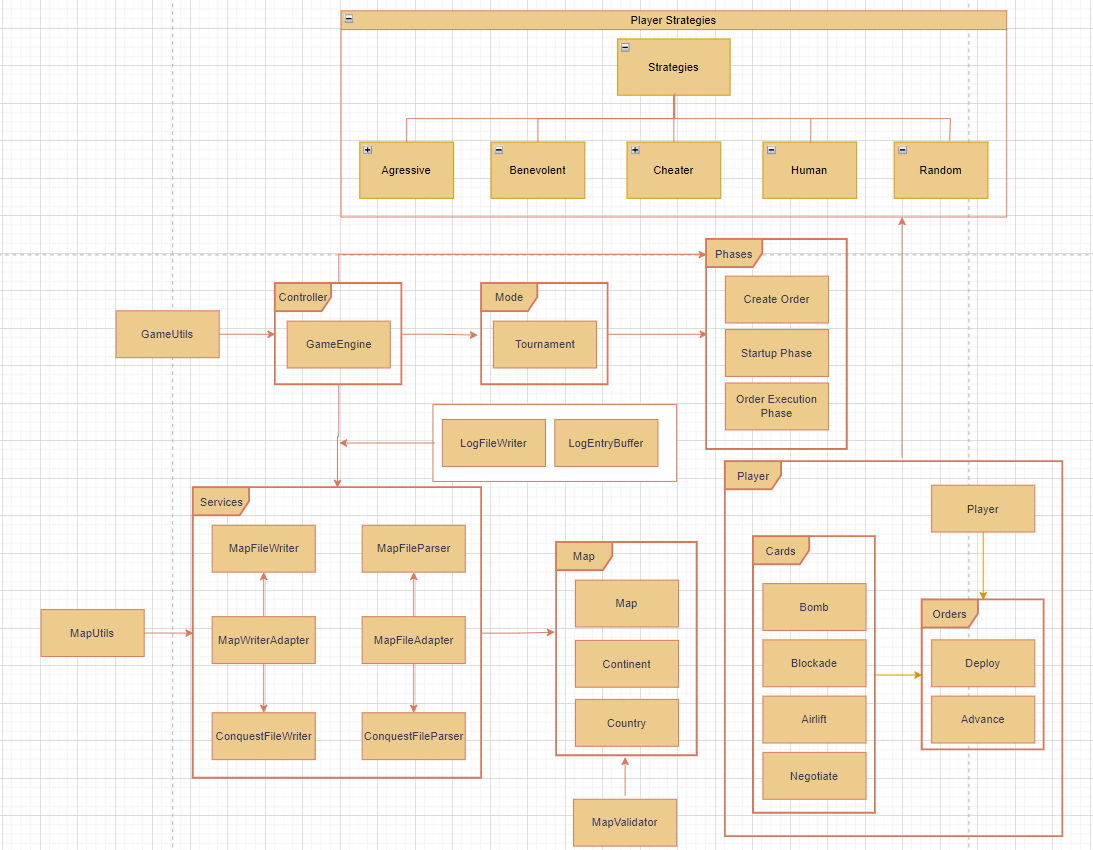
# Team U6:

# Architecture Diagram Document



* Game Engine:
  + Manages the flow of the game, processing commands, and updating the game state
* Player Classes:
  + AggressivePlayer, BenevolentPlayer, CheaterPlayer, HumanPlayer, RandomPlayer
  + Each player class contains attributes specific to the player type
* Map Classes:
  + Country, Continent, Map.
  + Represents the geographical structure of the game.
* Order Classes:
  + Order, Advance, Deploy, Airlift.
  + Implements the Command Pattern to handle different game orders
* Card Classes:
  + Bomb, Blockade, Airlift, Diplomacy.
  + Represents the cards and their effects on the game.
* Phase Classes:
  + IssueOrderPhase, StartupPhase, OrderExecutionPhase.
  + Represents different phases of the game using the State Pattern
* LogEntryBuffer:
  + Observer Pattern implemented here
  + Records and notifies observers (views) about various stages in the game.
* File I/O:
  + MapWriterAdapter, MapFileAdapter.
  + Implements the Adapter Pattern to write/read map files in different formats.
* Controller:
  + GameEngineController.
  + Controls the main command-handling logic.
* Services/ Utils:
  + GameUtils, MapUtils
  + Manages various game-related services, such as map editing, player management, and game state save/load
* Resources:
  + Map Files.
  + Stores and manages all map files.
* PlayerBehaviorStrategy:
  + AggressivePlayerStrategy, BenevolentPlayerStrategy, CheaterPlayerStrategy, RandomPlayerStrategy.
  + Implements different strategies for player behaviors using the Strategy Pattern.

## Detailed Explanation for implementation of the required design patterns:

## State Pattern

State pattern is implemented through the use of Phases in our design. The State pattern allows an object to alter its behavior when its internal state changes. Here, the Phase class represents the various states of the game, and each concrete state subclass provides implementations for the commands specific to that state.

Key elements of the State pattern implementation in this code:

* Abstract State Class:
  + Phase is an abstract class representing the state. It contains abstract methods for common commands like loadMap, showMap, editCountry, etc.
* Concrete State Classes:
  + Concrete subclasses of Phase represent specific states of the game (e.g., Preload, PlaySetup, etc.).
  + Each concrete state provides specific implementations for the abstract methods, defining the behavior associated with that state.
* Context (GameEngine):
  + The GameEngine class maintains a reference to the current state (Phase) and can transition between states using the next() method.
  + Various commands in the GameEngine delegate their behavior to the current state's implementation.
* State Transitions:
  + The next() method in each state transitions the GameEngine to the next state, implementing the logic for moving from one state to another.

By using the State pattern, the code achieves flexibility and extensibility in handling different states of the game. It allows the GameEngine to vary its behavior based on its internal state, facilitating easy addition of new states and modifications to existing ones without modifying the GameEngine class.

## Command Pattern

The Command Pattern is implemented through the use of Orders in our design. The Command Pattern is a behavioral design pattern that turns a request into a stand-alone object, allowing users to parameterize clients with queues, requests, and operations. In our case:

* Command (Order) Interface:
  + Order is the command interface, which declares the execute() method.
  + It also declares the printOrder() method to display information about the order and the isValid() method to check the validity of the order.
* Concrete Commands (Advance, Deploy, Airlift, etc.):
  + Subclasses of Order (e.g., Advance, Deploy, Airlift) provide specific implementations for the execute(), printOrder(), and isValid() methods.
  + Each subclass encapsulates a specific order and its behavior.
* Invoker (Player):
  + The Player class acts as an invoker, calling the execute() method on the concrete command (order) when needed.
  + The player can issue different types of orders by creating instances of the corresponding concrete command classes.
* Client (Player):
  + The client (Player) is responsible for creating and invoking the commands.
  + It can issue orders by creating instances of concrete command classes (e.g., AdvanceOrder, DeployOrder) and executing them.

## Observer Pattern

The Observer Pattern is implemented through the use of LogEntryBuffer and LogFileWriter in our design where it implements the Observer pattern using Java's built-in Observable class.

* Subject (LogEntryBuffer):
  + The LogEntryBuffer class extends Observable, indicating that it is the subject being observed.
  + It maintains a String d\_logMessage to store the log messages.
* Concrete Observer (LogFileWriter):
  + In the constructor of LogEntryBuffer, an instance of the LogFileWriter class is created and added as an observer using this.addObserver(l\_logWriter).
  + LogFileWriter is presumably a class that implements the Observer interface, responding to updates from LogEntryBuffer.
* Notify Observers:
  + The currentLog method in LogEntryBuffer is responsible for updating the log message and notifying observers.
  + The setChanged() method is called before notifyObservers() to mark that the state has changed.
  + notifyObservers() triggers the update method of all registered observers, notifying them of the change.
* Update Logic (currentLog Method):
  + The currentLog method sets the d\_logMessage based on the provided p\_logType and p\_messageToUpdate.
  + It uses a switch statement to handle different log types (command, order, phase, effect, start, end) and formats the log message accordingly.
  + After setting the log message, it marks the state as changed and notifies observers.

Overall, when the state of the LogEntryBuffer changes (when a new log message is set), it notifies its observers (in this case, the LogFileWriter). The LogFileWriter can then respond to the update by performing any necessary actions, such as writing the log message to a file.

## Adapter Pattern

The Adapter Pattern is implemented in our code using two classes: MapFileAdapter and MapWriterAdapter.

* MapFileAdapter
  + Target (MapFileParser):
    - MapFileParser is the target interface that the client code (the code using MapFileAdapter) expects to work with.
  + Adaptee (ConquestFileParser):
    - ConquestFileParser is the existing class that needs to be adapted to work with the MapFileParser interface.
  + Adapter (MapFileAdapter):
    - MapFileAdapter extends MapFileParser, acting as an adapter to translate calls from MapFileParser to ConquestFileParser.
  + Constructor:
    - The constructor of MapFileAdapter takes a ConquestFileParser as a parameter, initializing it with the name of the map.
  + Method Translation:
    - The parseMapFile method in MapFileAdapter translates calls to parseMapFile from MapFileParser to ConquestFileParser.
* MapWriterAdapter
  + Target (MapFileWriter):
    - MapFileWriter is the target interface that the client code (the code using MapWriterAdapter) expects to work with.
  + Adaptee (ConquestFileWriter):
    - ConquestFileWriter is the existing class that needs to be adapted to work with the MapFileWriter interface.
  + Adapter (MapWriterAdapter):
    - MapWriterAdapter extends MapFileWriter, acting as an adapter to allow using ConquestFileWriter as a MapFileWriter.
  + Constructor:
    - The constructor of MapWriterAdapter takes a ConquestFileWriter as a parameter, initializing it with the target map.
  + Method Delegation:
    - The saveMap method in MapWriterAdapter delegates the saveMap method to the corresponding method in ConquestFileWriter.

Both adapters in our code (MapFileAdapter and MapWriterAdapter) bridge the gap between the existing classes (ConquestFileParser and ConquestFileWriter) and the target interfaces (MapFileParser and MapFileWriter). They allow the client code to work with the target interfaces while using the functionality provided by the existing classes through adaptation.

## Strategy pattern

Our code implements the Strategy Pattern using PlayerStrategy, where different strategies for creating orders are encapsulated in separate strategy classes.

* Strategy Interface (PlayerStrategy):
  + PlayerStrategy is the strategy interface that declares the common methods, such as createOrder and getStrategyName, that all concrete strategies must implement.
* Concrete Strategies (AggressiveStrategy, BenevolentStrategy, CheaterStrategy, RandomStrategy):
  + Each concrete strategy class extends PlayerStrategy and provides its own implementation of the createOrder and getStrategyName methods.
  + Each strategy class encapsulates a specific behavior for creating orders.
* Context (Player):
  + The Player class contains a reference to a PlayerStrategy (d\_targetPlayer), allowing the player to switch strategies dynamically.
* Usage of Strategies:
  + The Player class delegates the responsibility of creating orders to its associated strategy by calling the createOrder method.
  + The Player class can also query the strategy name using the getStrategyName method.