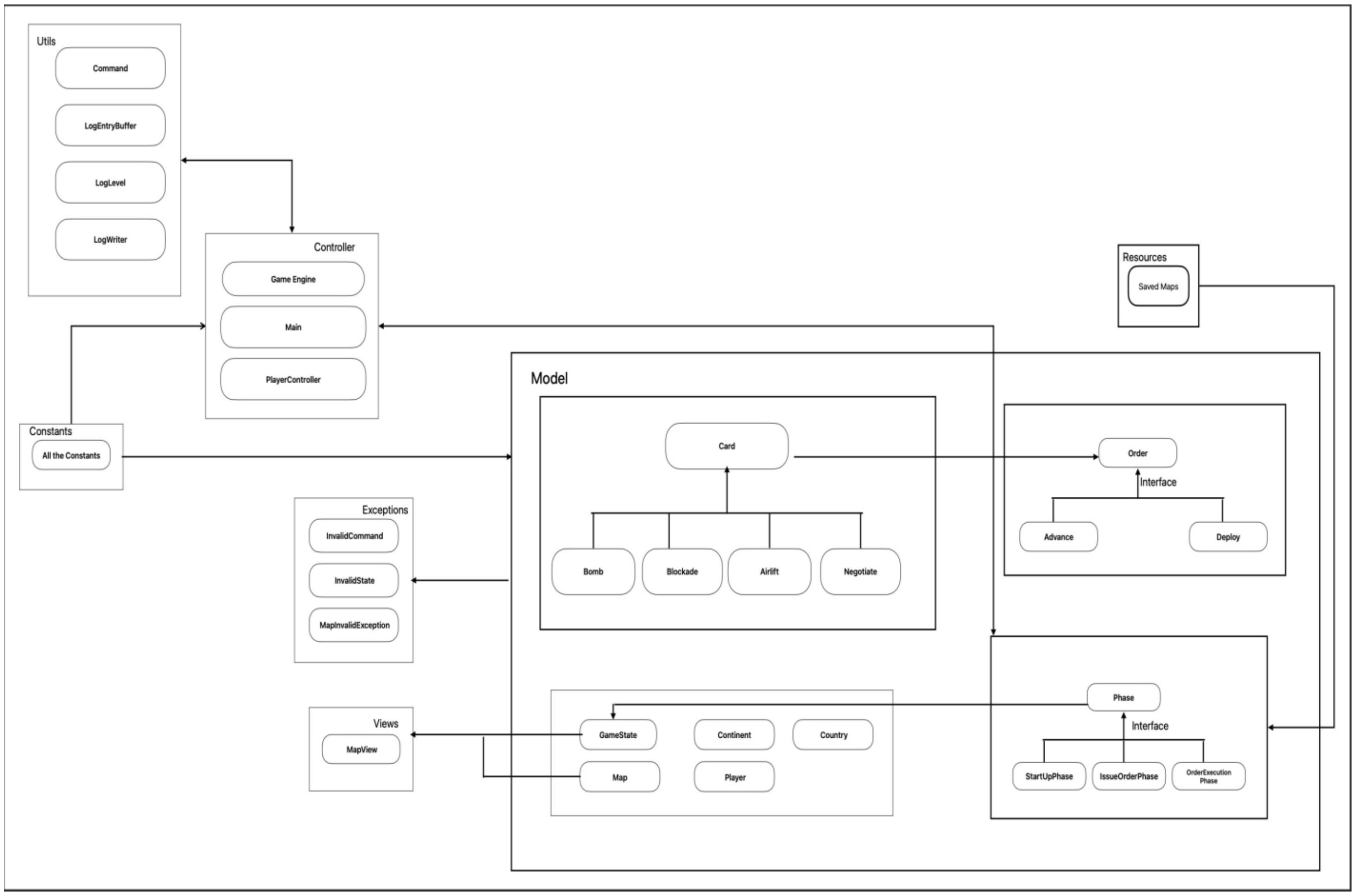
PROJECT ARCHITECTURE BUILD 2

Architecture Diagram:



State Design Pattern

The State Design Pattern offers a structured approach to managing the behavior of an object based on its internal state. Here are some key use cases for implementing the State Design Pattern:

1. The state pattern allows you to easily modify the logic between game phases by adding or modifying state classes.
2. Each state's business logic is contained within separate state classes, making the application more testable and maintainable.
3. The game controller returns the controller of the next state upon successful execution or force stop of the current state.
4. It ensures that the application doesn't move to the next state if the prerequisites are not satisfied, preventing crashes or improper execution.

Game Engine Class:

1. This class serves as the central hub for the game, functioning as a mediator between various game states.
2. Its role includes initializing the game's initial phase and maintaining a record of the current game state.
3. This class relies on the Game Phase class to determine the available next states and their corresponding controllers after successfully executing the current state. Additionally, it is responsible for triggering the controllers of the next state via the l_game.getCurrentPhase(). initPhase()   method.

Phase Class -:

This class manages the game's state through various phases and serves as the parent class for all the phases within the game.

The game is essentially divided into the following stages:

1. Startup Phase: This phase involves setting up the gameplay.
2. Issue Order Phase: In this stage, the game gathers orders from each player.
3. Execute Order Phase: During this phase, the game executes and verifies the orders provided by the players.

Command Pattern -:

The Command Pattern offers a behavioral design pattern that is used to encapsulate a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations. Here are some key use cases for implementing the State Design Pattern:

1. Adding or modifying order processing is both straightforward and adaptable.
2. Orders can be generated and saved for later execution during gameplay.
3. The application's testability is enhanced by maintaining abstract and consistent order logic.
4. It distinguishes the initiator of an operation from the entity executing the operation.
5. The system facilitates the seamless addition of new commands without necessitating changes to existing classes.

The following describes Command pattern implementation:

1. The Player class act as the invoker in the command Pattern. It sets up the environment and invokes the issue_order () method for each player.
2. Each player acts as a client that creates and issues commands. The issue_order() method in the player is responsible for reading and validating commands in a round-robin fashion. If a wrong command is entered, it prompts the user to re-enter the command. In this context, the player represents the sender of the command.
3. The init() method of the OrderExecutionPhase class is invoked by the game engine, and it acts as the receiver of the commands. It executes the commands as per the state pattern.
4. Each player invokes the next_order() method to retrieve the orders, and then they execute these orders using the execute() method. This follows the Command pattern's structure, where the command (order) is separated from its execution, allowing for flexibility and decoupling between the sender and the receiver.

Observer Pattern :-

The Observer Design Pattern is a behavioral design pattern that is used to establish a one-to-many dependency between objects, so that when one object (the subject) changes its state, all its dependents (observers) are automatically notified and updated. This pattern is used to achieve loose coupling between objects, making it a valuable tool in software design for several reasons:

1) The Observer pattern helps in reducing the coupling between the subject (the object being observed) and its observers.
2) By separating the subject and observer classes, you can create a more modular and reusable code.
3) Observers can be added or removed dynamically at runtime. This allows for a flexible system where you can adapt to changing requirements without modifying the subject.
4) Observer patterns are often used for logging and monitoring systems, where different components or modules can observe and log events or activities without being tightly integrated.