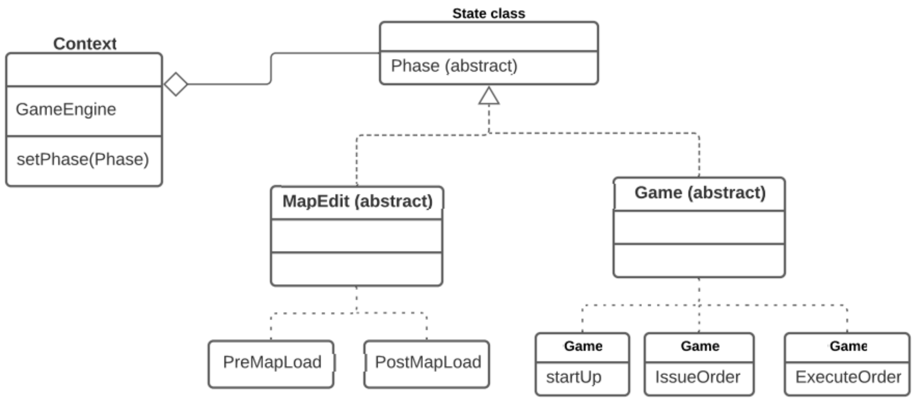
Refactoring

In this Build 2, apart from adding new features to Build 1, we have also implemented the following:

- State Pattern
- Command Pattern.
- Observer Pattern
- Various orders for the players, including:
  - Advance
  - Airlift
  - Bomb
  - Blockade
  - Diplomacy

State Pattern



In this context, the GameEngine class serves as the context class, while the Phase class acts as the state class. To introduce the state pattern into our current codebase, we've reorganized our existing MapEdit and Game classes into distinct phases.

Specifically, the MapEdit class has been subdivided into the following phases:

- PreLoad

- PostLoad

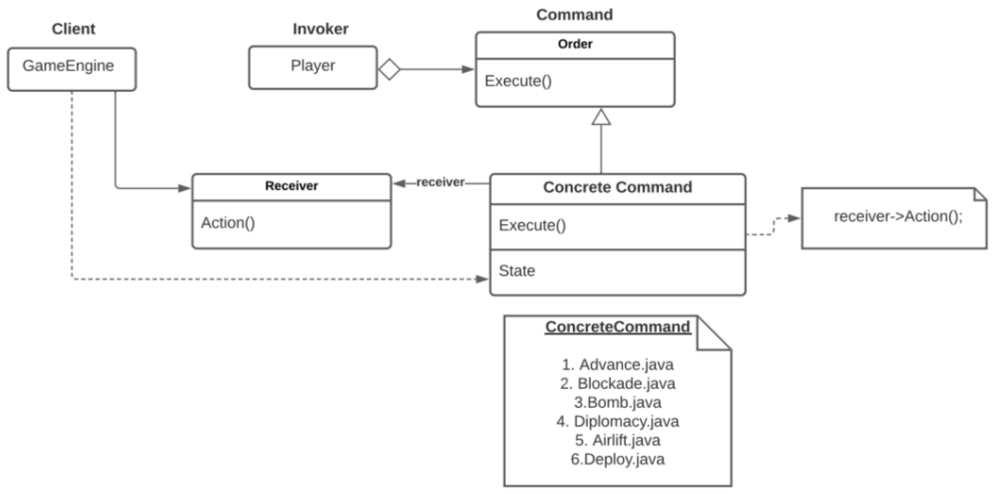
Likewise, the Game classes have been segmented into the following phases:

- PlaySetup

- CreateOrders

- ExecuteOrders

Command Pattern



In this scenario, the GameEngine class serves as the client, the Player class acts as the invoker, and the Order class represents the command. To implement the command pattern within our current codebase, we've introduced six distinct concrete commands to handle various orders. These concrete commands are as follows:

- Advance.java
- Bomb.java
- Blockade.java
- Airlift.java
- Deploy.java
- Diplomacy.java

The orders are generated when a player executes its issue_order() method. These orders are then executed when the GameEngine retrieves the player's orders from the Players using the next_order() method and subsequently executes the orders by invoking the execute() method of the Order class.

Possible Refactoring Targets

Listed below are the potential refactoring targets in our code:

1. Implement State Pattern

2. Implement Command Pattern

3. Move all functions related to Game in GameUtils class.

4. Execution and creation of orders is moved from GameEngine class into two separate concrete phase classes namely CreateOrder and Execute order which are called in a loop from the GameEngine class.

[GameEngine.java, CreateOrder.java, ExecuteOrder.java]

4. For the command pattern we had already implemented and abstract class name Order with execute() function in the first build and extending it we implemented the Deploy concrete class, for this build we added 2 more functions namely printOrder() and isValid() that would be used by all command concrete classes to print the order details under execution and check the validity of its execution respectively. [Order.java]

5. The map and player list from the GameUtils class had to be made static since they have been changed from many different classes [GameUtil.java]

6. We have also implemented the Observer pattern through a Logwriter that updates the log with status of the gameplay or map editor phase and maintains the record into the textfile. We have used the Observer class from java.utils to implement Observer pattern for the same.

Refactoring

1. Implementation of the State Pattern

The integration of the State Pattern allows us to effectively manage the phases across our application, covering both map editing and gameplay. The gameplay phases include Game Startup, Game Issue Order, and Game Execute Order. The GameEngine class operates as the context class for the State Pattern, with the State class represented by a new class named Phase.

Tests: We've established comprehensive testing procedures to ensure the accurate execution of commands within their respective phases. Importantly, we prevent the execution of invalid commands within specific phases.

2. Implementation of the Command Pattern

This implementation leverages the Command Pattern to handle orders efficiently. The Command class corresponds to the Order class, the Invoker Class aligns with the Player, and the Client class pertains to the GameEngine. Orders are instantiated as players execute their issueOrder() method, and these orders are subsequently retrieved from the Players using the nextOrder() method. The execution of these orders is achieved by invoking the execute() method of the Order class. We've created six concrete implementations of the abstract Order class, including Advance, Airlift, Blockade, Bomb, Deploy, and Diplomacy.

Tests: We've designed a series of individual tests to evaluate the validity of each command type.

3. Optimized Organization of Game-Related Methods in GameUtils

Our approach to declutter the GameEngine involves migrating all game-related methods to the GameUtils class. This includes functions such as Reinforcement armies' assignment, random number generation for assigning countries to players, and awarding cards to players upon battle victory. These functions have been refactored into distinct functions in the GameUtils class which is calling the methods from the orders.

Test: Thorough testing covers aspects like Random Number Generation and Reinforcement army calculation.