TEAM 10: PROJECT ARCHITECTURE BUILD 3

COURSE: SOEN 6441 (WINTER 21)

INSTRUCTOR- JOEY PAQUET

TEAM MEMBERS:

1. Dhananjay Narayan
2. Madhuvanthi Hemanthan
3. Prathika Suvarna
4. Neona Pinto
5. Surya Manian

State Pattern - States of the Warzone Game

The idea is to use the *State behavioural pattern* so that

1. It makes our application relatively easy and dynamic to modify the logic dictating the rules between phases.
2. The state pattern makes the application more testable as each state's business logic is kept in separate models which get invoked by the respective controllers.
3. The Controller of the next state is returned only upon successful execution/force stop of the current state.
4. Even if the force stop of a state happens, it does not move to the next state if the pre-requisites are not satisfied(Creating GameMap, Adding Players etc).
5. In this way we can prevent the application from getting crashed or implemented in the wrong direction.

The following diagram describes how the points stated above are attained:

GameEngine Class:

This is the starter class, which acts as an intermediate between different states of the game.

It is responsible for setting up the start phase of the game.

It uses the GamePhase Enum class to get the next possible states and their controllers for the *successfully executed current state.*

It is responsible for the invocation of the respective next state's controllers via controller.start() method.

GamePhase Enum (state management):

This is the enum class holding the different possible states in the warzone game.

The game is basically split up into following states:

MapEditor - Holding Map Operations

StartUp/GamePlay - Holding Gameplay Setup

Reinforcement - Calculating Reinforcement armies

IssueOrder - Gets the set of orders from each player

ExecuteOrder - Executes and validates the orders provided by players

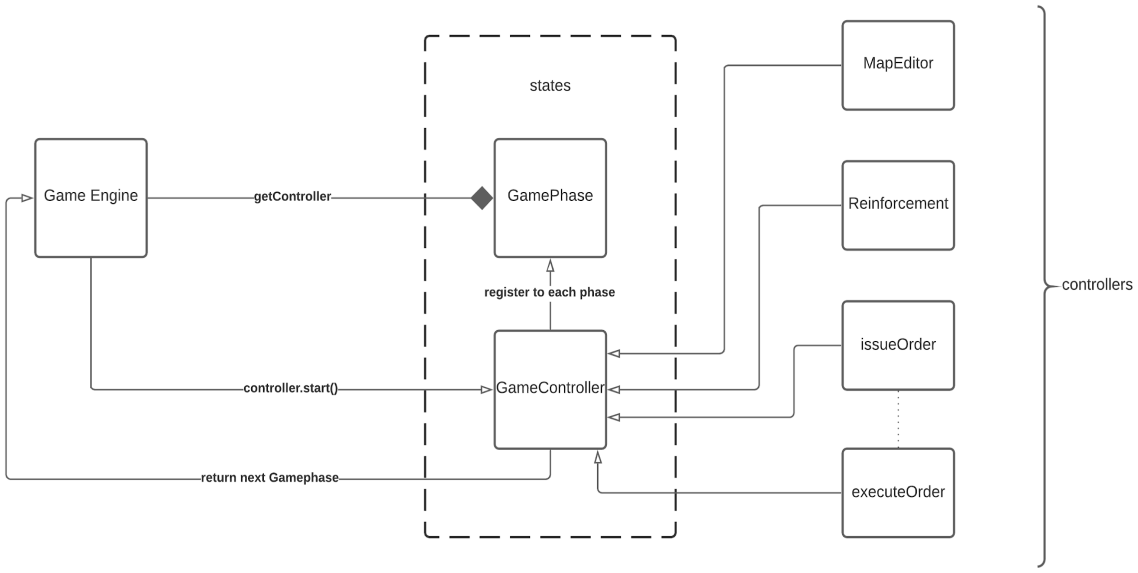
The GamePhase enum is responsible for holding the set of next possible states from each state mentioned above.

Also it is the one which is returning the controller class of each phase.

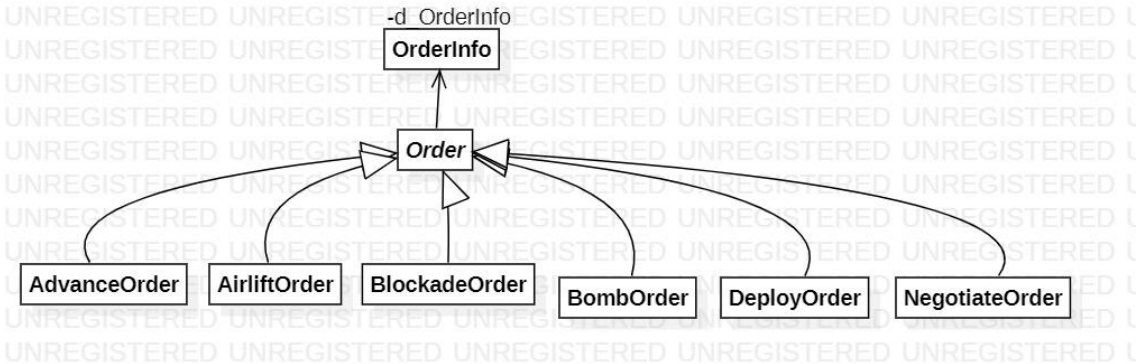
GameController Class:

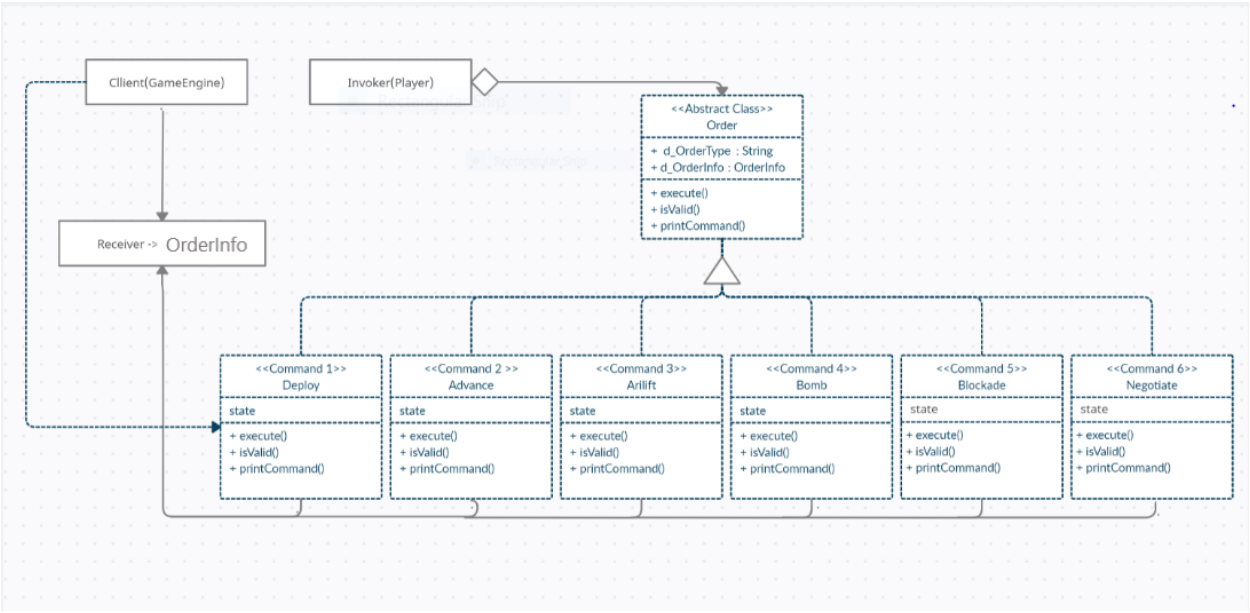
The interface which provides definition for all the controllers of the game phases.

The following diagram illustrates the *game flow*:



Command Pattern - GamePlay Orders





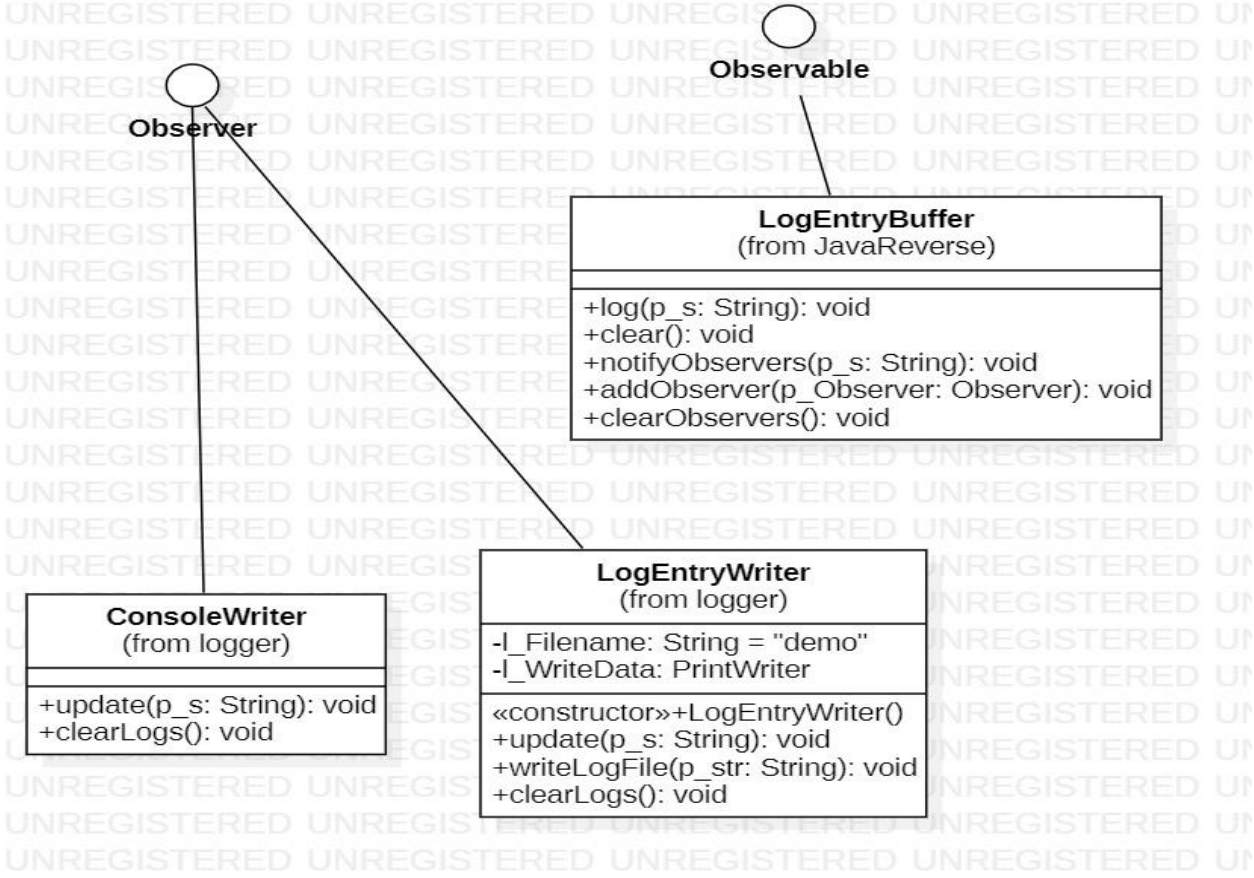
The Figure shows a UML diagram with the relevant classes that makeup how the player orders are issued and executed during gameplay. The idea is to use the Command behavioural pattern so that

1. It is relatively straightforward and dynamic to add/modify the processing of order.
2. The Orders can be created and stored for execution at a later time during gameplay.
3. The application is testable as order logic is kept abstract and consistent.
4. It separates the object that invokes the operation from the object that actually performs the operation.
5. It makes it easy to add new commands, because existing classes remain unchanged.

The following describes Command pattern implementation in the diagram:

1. The IssueOrder Controller is invoked which sets up the environment and invokes the issue_order() method for each player.
2. The issue_order() is invoked by the Player where the commands are read by the Player in round robin fashion. On entering wrong commands the user is asked to reenter the commands. Currently the IssueOrder Controller is defined as the datasource to get the order command from the user. This is with respect to the MVC pattern, i.e. the controller gets user input from the view.
3. Once all the orders have been created in the IssueOrder Controller, the game engine will invoke the ExecutionOrder start() method as per the state pattern.
4. Each player invokes the next_order() to get the orders and execute() method to execute the orders in round robin fashion.

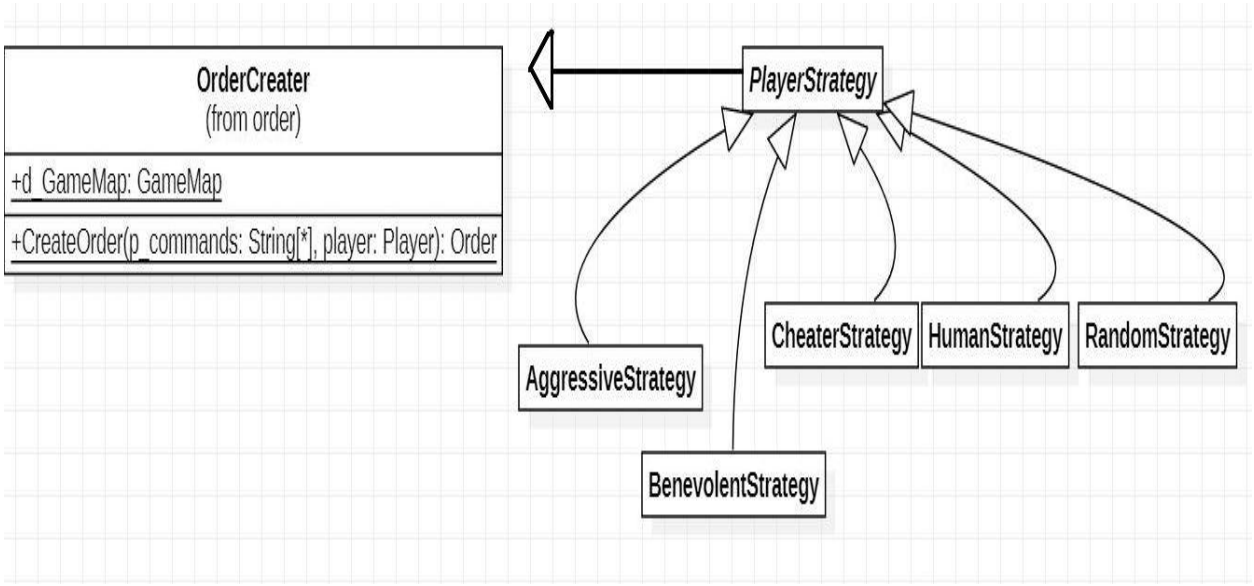
Observer Pattern



- The above diagram shows the overview of how the Observer pattern is implemented in our project.
- Observable and Observer are the interfaces for LogEntryBuffer and LogEntryWriter respectively.

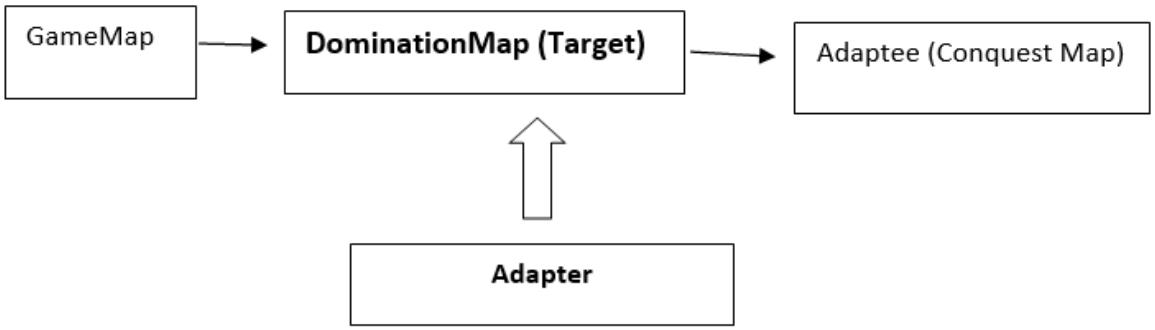
- The LogEntryBuffer object is filled with all the information of the actions happening in the game.
- It gets all the actions from log() function which takes the string parameter. It then passes the string to the observers and the observers can do the required operations with this received notification.
- LogEntryWriter and ConsoleWriter are the observers which receives the notification from LogEntryBuffer. In this project, LogEntryWriter is used to update the actions of the game to a log file. The function writeLogFile() takes the action as a string and then writes to a log file in the append mode.
- So whenever a particular action has to be logged, we create an object of LogEntryBuffer and the writing to part is taken care of by the observer. Similarly, the ConsoleWriter prints the action to Console.

Strategy Pattern



- The strategy pattern has been implemented to change the behaviour of the player in the gameplay.
- The OrderCreater is the context class that uses the behaviour of the player.
- Each strategy has a createCommand() method that created the order for the player based on the different types of strategies.
- There are 5 player strategies that have been implemented - Aggressive, Benevolent, Cheater, Human and Random. Each strategy of the player exhibits a different behaviour.
- The Aggressive player plays aggressive moves in order to win the game.
- The Benevolent player makes moves in a defensive way to protect territories and never makes an attack.
- A Cheater player manipulates the game that will allow him to conquer the neighboring countries without actually playing valid game moves.
- A human player requires user interaction in order to progress in the game.

Adapter Pattern



- Adapter pattern helps in implementing a bridge between two objects that have similar functionalities but used in a different manner.
- It lets two different classes work together, that could not otherwise due incompatible interfaces.
- The GameMap is the client that reads from the Target.
- In our implementation, we have two classes that read two different types of maps.
- The target is the domination map while the adaptee is the Conquest map reader.
- We have implemented an adapter class that acts as a bridge between the conquest and the domination map readers.
- The object of adaptee is created in the adapter class.
- The adapter overrides the method of the Target class and returns the object of the Conquest map type.
- This way both conquest and domination maps can be read without actually changing the logic of either.
- So now, GameMap can read from both Domination and Conquest map formats through the adapter pattern.