TEAM 10: PROJECT ARCHITECTURE BUILD 2

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

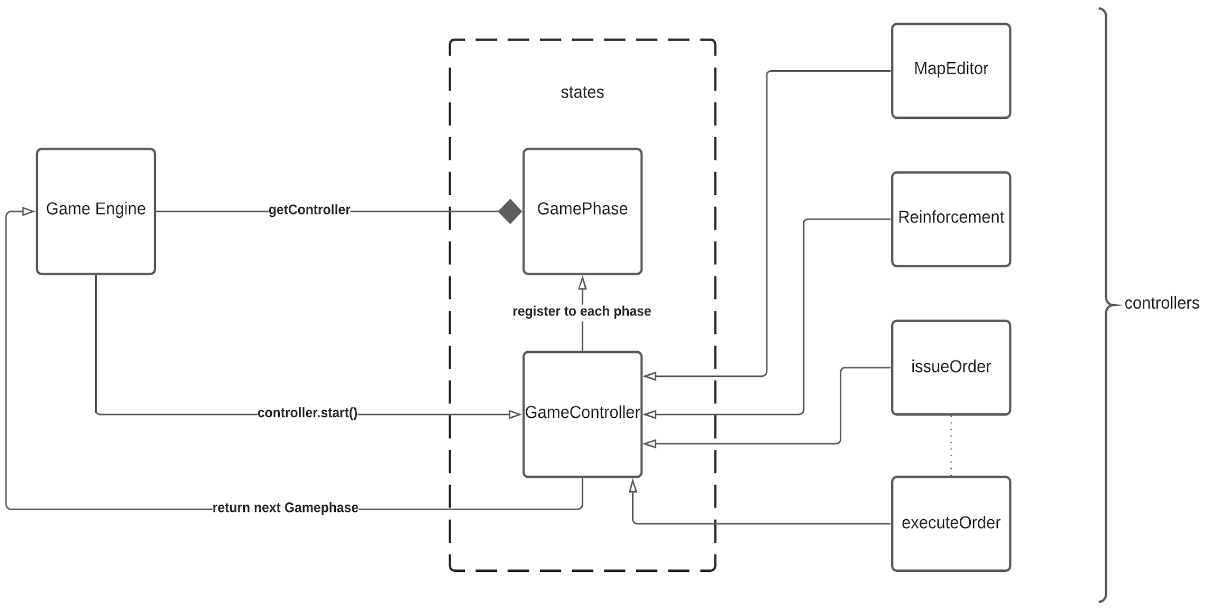
The GamePhase enum is responsible for holding the set of next possible states from each state mentioned above.

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

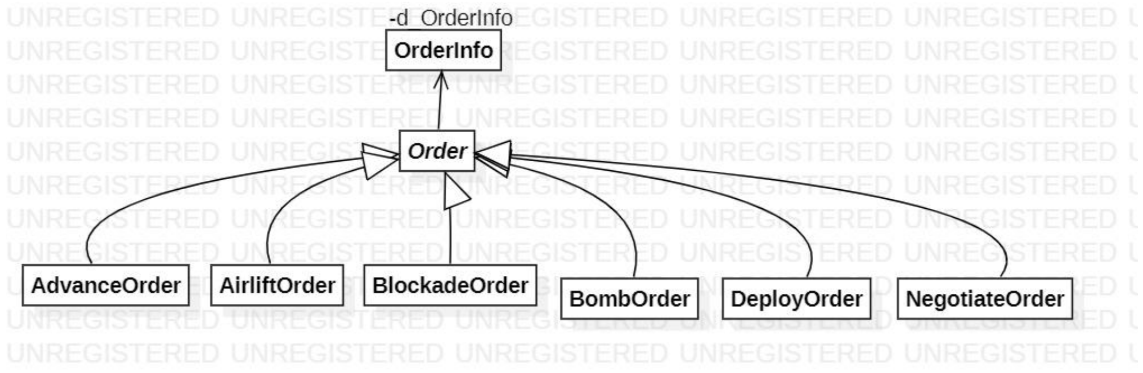
GameController Class:

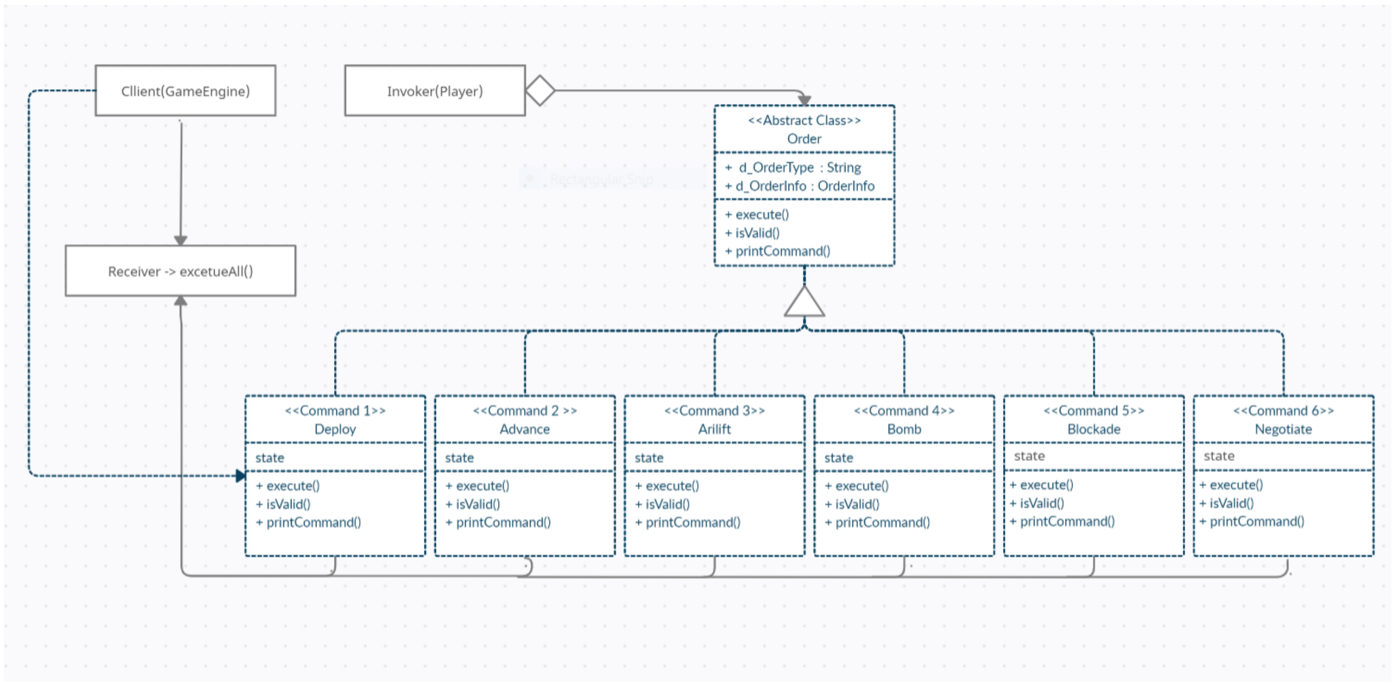
The interface which provides definition for all the controllers of the game phases.

The following diagram illustrates the *game flow*:



Command Pattern - GamePlay Orders





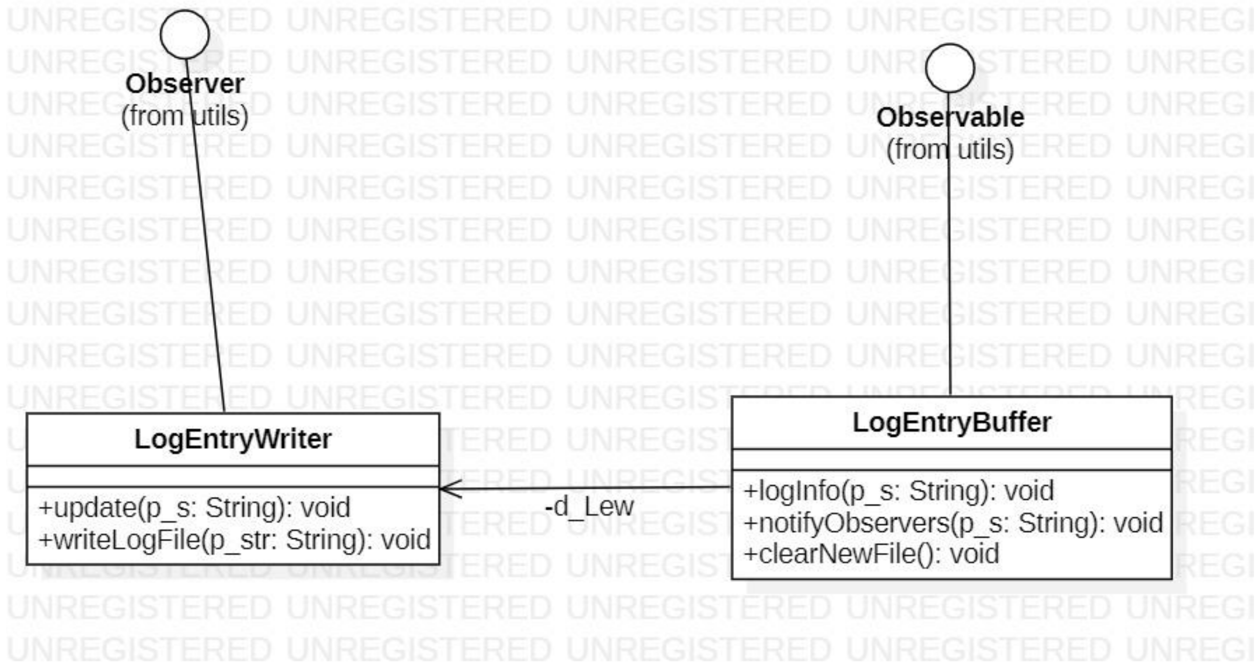
The Figure shows a UML diagram with the relevant classes that makeup how the player orders are issued and executed during gameplay. The idea is to use the Command behavioural pattern so that

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


Observer
(from utils)
Observable
(from utils)
LogEntryWriter
+update(p_s: String): void
+writeLogFile(p_str: String): void
-d_Lew
LogEntryBuffer
+logInfo(p_s: String): void
+notifyObservers(p_s: String): void
+clearNewFile(): void
UNREGISTERED

- 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 logInfo() 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 is the observer which received 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.
- The observer pattern can be used for other possible tasks by adding more functions after it gets the action in the update() function in LogEntryWriter.

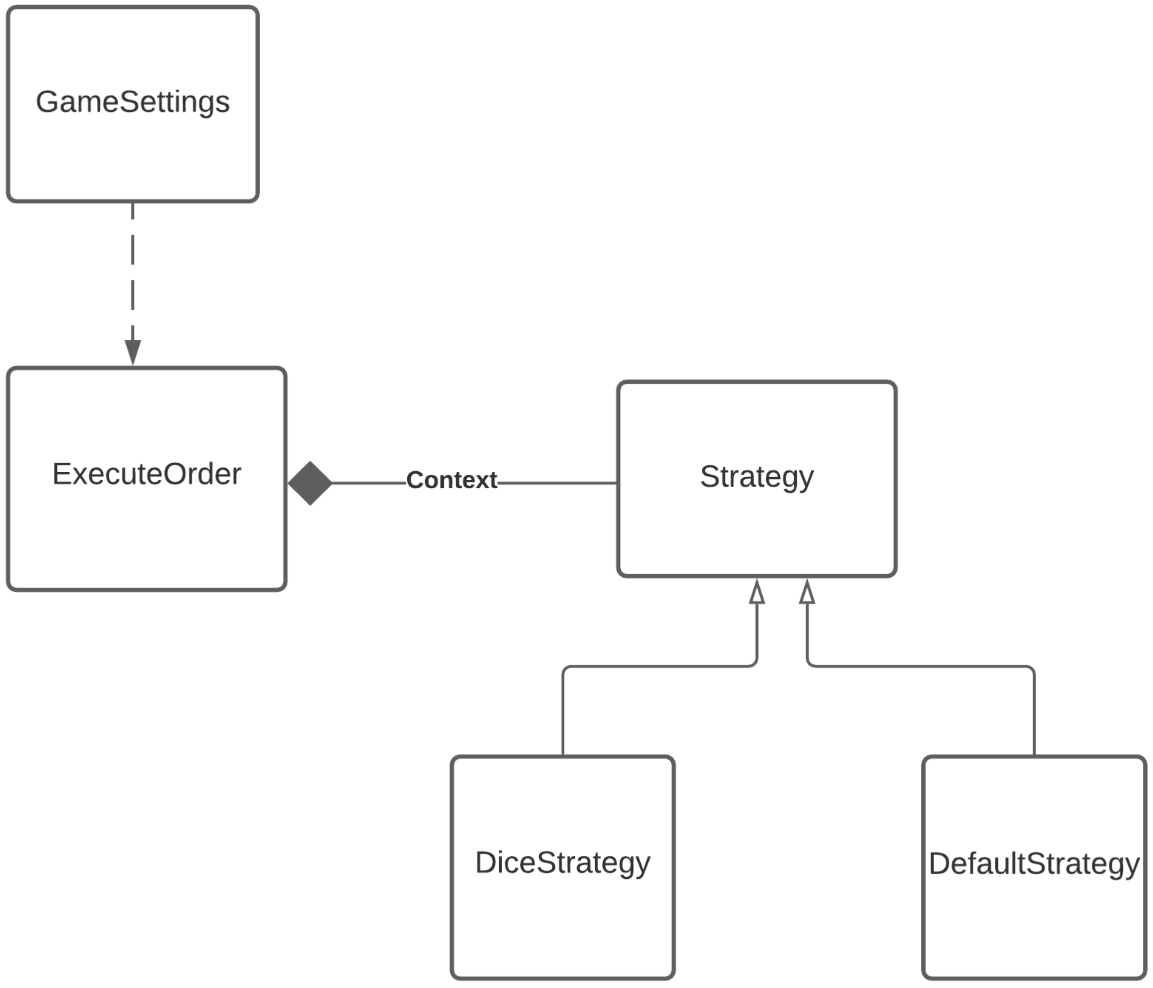
Strategy Pattern(For Game):

We tried to implement the strategy pattern deciding the Game Attack logic.

There are basically two strategies allowed in this build:

- DefaultStrategy - Calculates with default value
- DiceStrategy - Calculates with random value

The following diagram illustrates how the game settings is made based on strategy decision:


GameSettings
ExecuteOrder
Context
Strategy
DiceStrategy
DefaultStrategy