TEAM 18: ARCHITECTURAL DESIGN- BUILD 3
COURSE: SOEN 6441 (ADVANCED PROGRAMMING PRACTICES)
Instructor- Dr. Amin Ranj Bar

TEAM MEMBERS:

1) Jay Bhatt

2) Madhav Anadkat

3) Bhargav Fofandi

4) Meera Muraleedharan Nair

5) Reema Ann Reny

6) Mariya Bosy Kondody

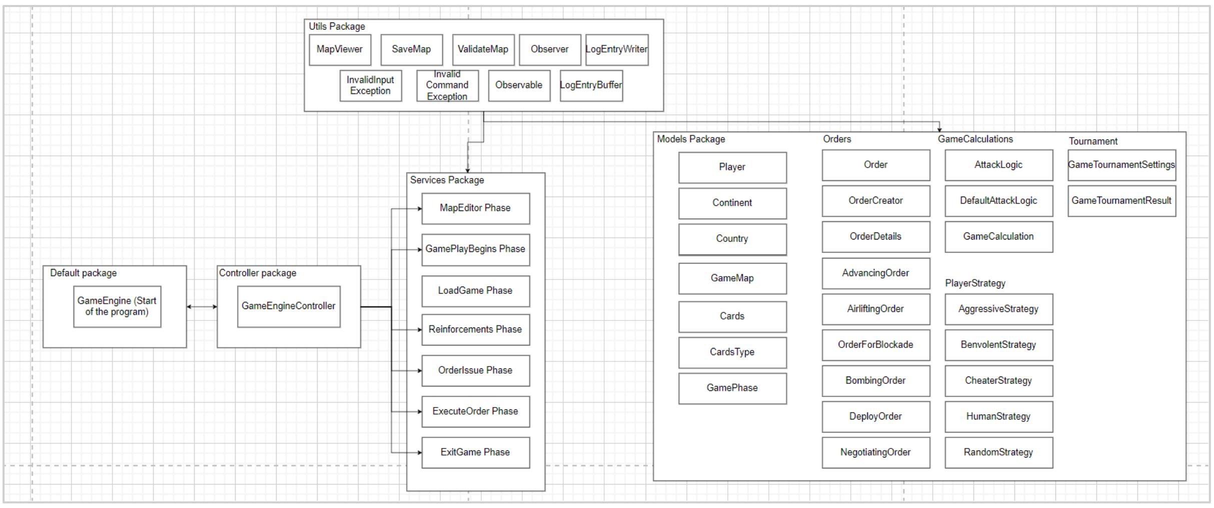
Project Title: Project Warzone Game

Overview

This document provides a concise overview of the architectural design of the Project Warzone Game. It describes how the State, Command, and Observer design patterns have been integrated into the system and how the system is organized into coherent modules.

Architectural Diagram:

The architectural diagram provides a visual representation of the game modules and their interactions.



GameEngine Controller:

- GameEngineController serves as the main controller and is responsible for managing the flow of different phases in the game.
- The start method is the entry point for the game controller. It accepts a GamePhase parameter that indicates the game's current phase. The process returns the phase of the game that will be played after the current one. The GamePhase is an object or enum that contains data regarding the game's current phase.
- The start method (Exception Handling) declares that it has the ability to throw an exception. This means that exceptions might be thrown when various game phases are being executed, and the concrete controller classes that implement this interface must handle these exceptions.
- It encourages modularity and permits flexibility in putting each phase's unique behaviour into practice, as needed for the operation of the game.

GamePhase Enum:

- GamePhase highlights the various stages of a Warzone Risk game. This enum is essential for controlling the flow and transitions between the various phases of the game, with each enum value denoting a different phase.

- Each enum value represents a specific game phase, which are of the following:
  - MapEditor - Handles Map Operations
  - StartUp - Maintaining the Gameplay Configuration
  - Reinforcement - Compute the Reinforcement armies
  - IssueOrder - Obtains each player's set of orders.
  - ExecuteOrder - This function carries out and verifies the orders that players submit.
  - ExitGame- Ends the game once all countries are conquered
- A utility method for obtaining the next game phase and validating it is the nextState method. After receiving a GamePhase parameter, it determines whether the supplied phase is present in the list of permitted future states for the active phase. It returns the phase supplied if it is valid; if not, it returns the current phase.

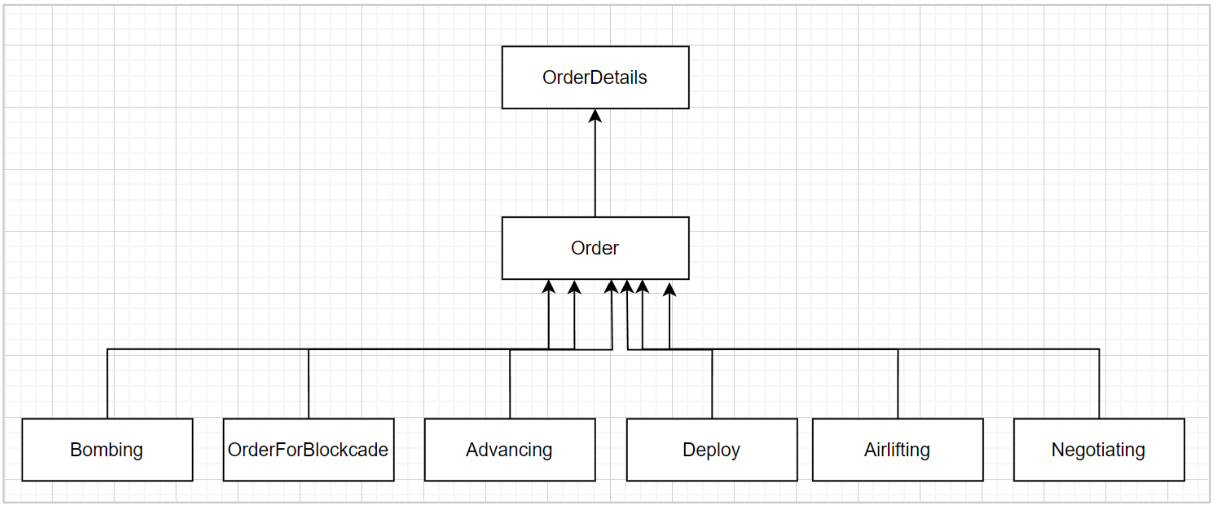
Design Patterns used:

State Pattern:

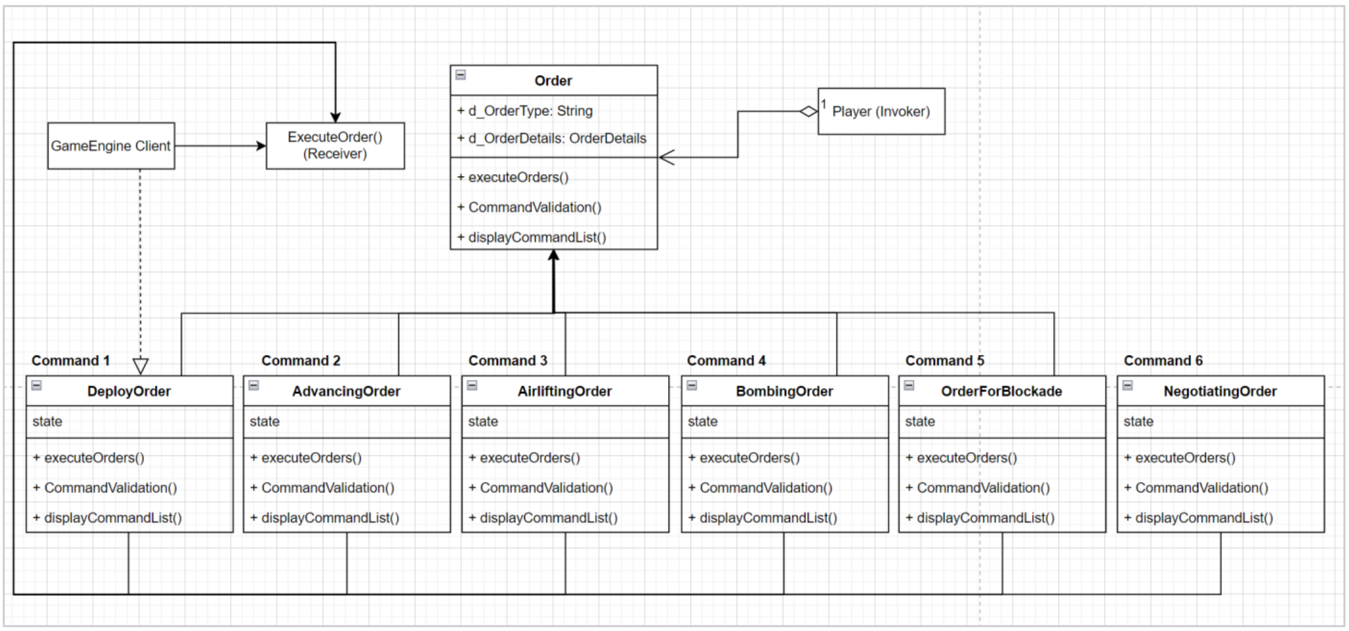
- State design pattern facilitates seamless transition between different states.
- It manages the various operational modes of the game, including "Error," "Active," and "Idle."
- Using the state pattern makes our application more flexible by allowing us to quickly and dynamically change the rules that dictate when one phase ends and another begins.
- In this code, the GamePhase enum represents different states of a game and the possibleStates() method defines a set of allowed next states for each state.
- Every state has its own controller as well, and the getController() function returns the controller that corresponds to that particular state.
- The State Pattern is characterized by this structure. When an object's internal state changes, the State Pattern enables it to change how it behaves.
- The nextState() method contains the state transition logic, and each state in this code corresponds to a distinct game behavior.
- When a transition is requested, the method returns either the current state or the new state depending on whether the requested next state is permitted based on the defined possible states.
- Also implemented 2 game modes: Single Game Mode and Tournament Game Mode.
- Tournament Game Mode has methods for initializing tournament settings, parsing commands, setting up games, and displaying results.
- Single Game Mode uses user input for map and player strategies, initializing a game, and displaying the winner.
- The application's testability has been enhanced through the implementation of the state pattern.
- To ensure a controlled transition, the controller for the subsequent state is only supplied after the current state has been successfully executed or terminated forcefully.

- Importantly, the application remains robust. The next state won't happen unless certain requirements are met, like finishing the GameMap and adding players, even in the case of a forced state termination. This security measure keeps the application from crashing or acting differently than it is supposed to.

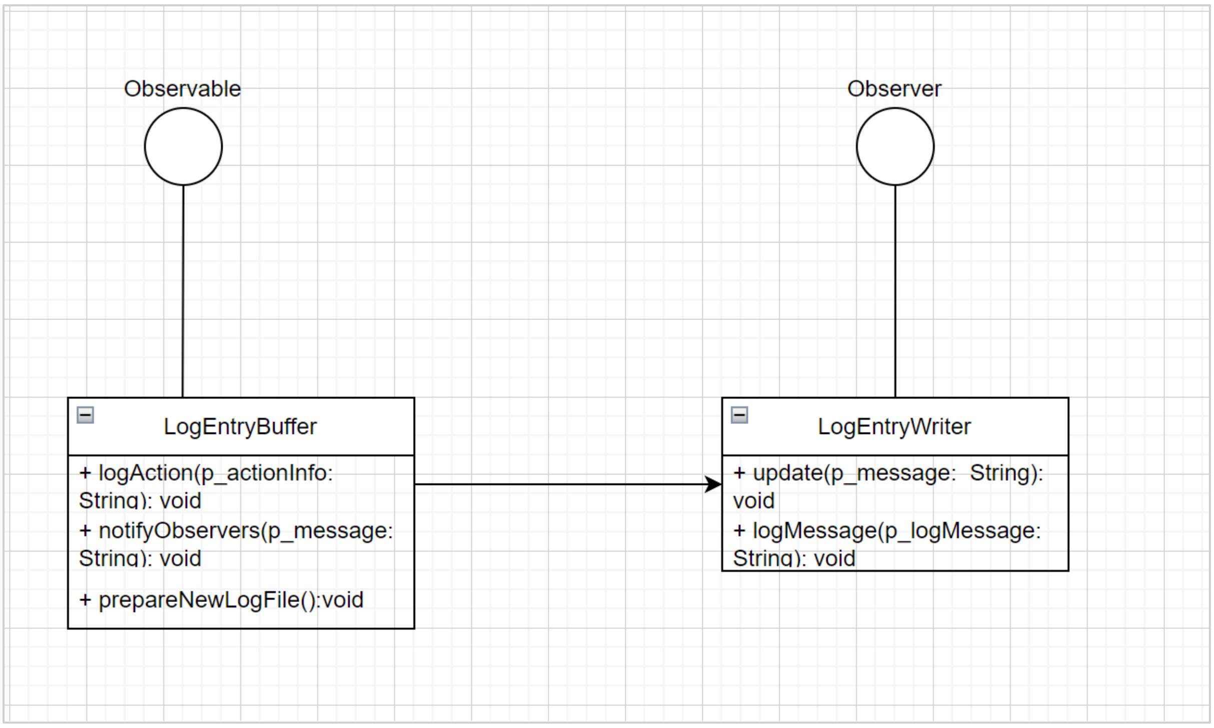
Command Design Pattern:



- Commands from the user are accepted by this module, which then encapsulates them as command objects and sends them to the relevant subsystems for execution.
- Each command in this code is handled by the start method of the OrderIssue. This class contains the logic for handling various commands, structured similarly to the concrete command classes in the Command pattern.
- The OrderIssue class serves as the invoker, or client, answering various commands from the players.
- To perform the designated game actions, the OrderIssue class interprets the commands and communicates with players, the GameMap, and other elements.
- The readFromPlayer method reads commands from the players and processes them in the start method.
- The start method, which processes the logic for handling various command types (such as "deploy," "advance," "bomb," etc.), is where commands are executed.
- The figure below represents the flow in which player issues orders and how they are executed during the gameplay:



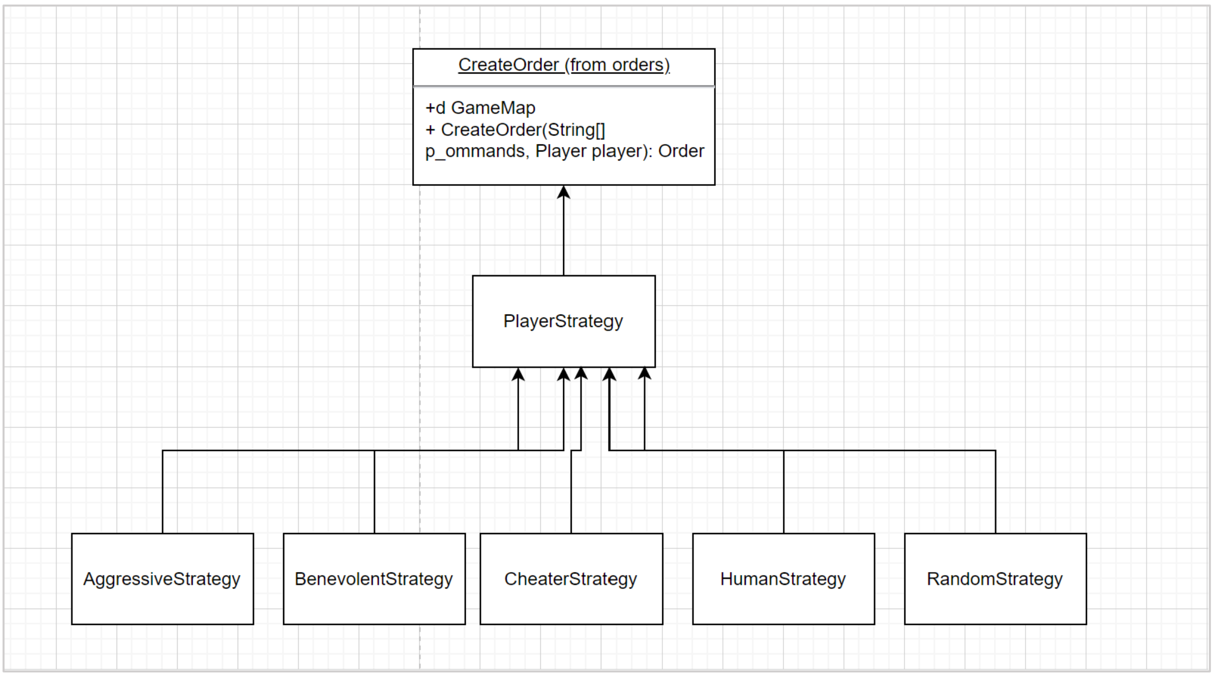
Observer Design Pattern:


Observable
Observer
LogEntryBuffer
+ logAction(p_actionInfo: String): void
+ notifyObservers(p_message: String): void
+ prepareNewLogFile():void
LogEntryWriter
+ update(p_message: String): void
+ logMessage(p_logMessage: String): void

- The Observer design pattern enables different components to subscribe to particular events and we receive log files when changes occur.
- The above diagram depicts an overview of the implementation of Observer Pattern in this project.
- In this instance, the observable subject is the LogEntryBuffer that notifies its observers when some state and data changes.

- The Observer interface, which comprises the update method for getting updates from the observed subject, is implemented by the LogEntryWriter class.
- To get updates from the subject (observable), the update method is used. The data to be logged is passed in as the p_message parameter. The update method uses the logMessage method to log the message when an update is received.
- To receive updates, LogEntryWriter, a concrete observer class, registers with the observed subject. This observer logs information to a log file when updates happen.
- To summarize, the LogEntryWriter class serves as an observer, recording the messages it receives from the observed subject (LogEntryBuffer) and acting as an observer by listening for updates. This is in line with the Observer design pattern, which allows observers to respond appropriately to changes in a subject's state after being informed of such changes.

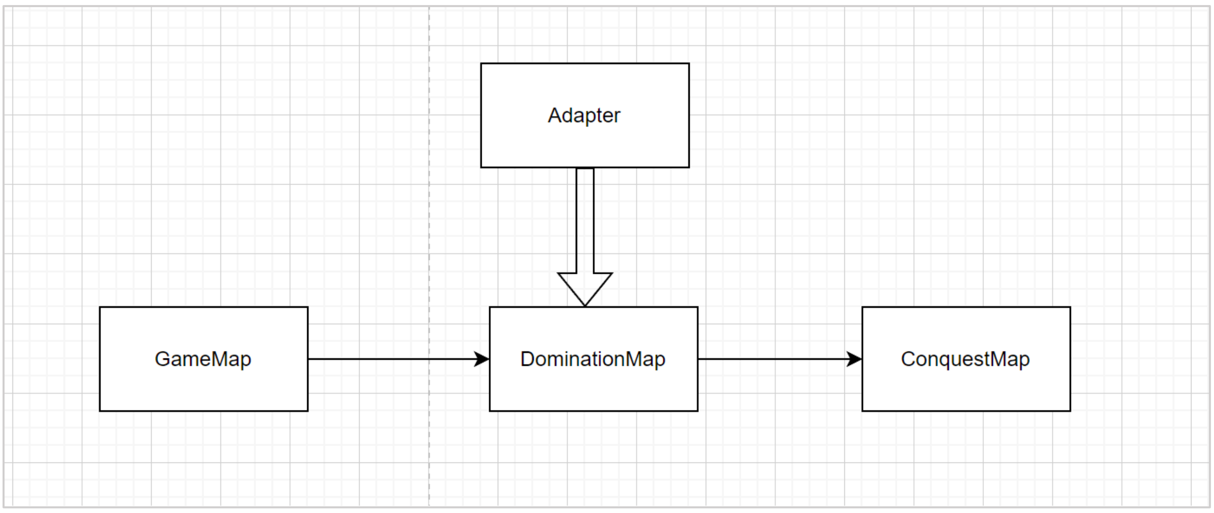
Strategy Pattern:



- Components:
  - **PlayerStrategy**: Abstract class defining the common interface for player strategies.
  - **Concrete Strategies**: Classes such as AggressiveStrategy, BenevolentStrategy, CheaterStrategy, HumanStrategy, and RandomStrategy; implementing specific player strategies.
- The strategy pattern has been utilized to encapsulate and interchange player strategies seamlessly.
- PlayerStrategy serves as the abstract strategy class, and concrete strategies implement the createCommand method.

- The createCommand() function in each strategy determines the player's order depending on the various kinds of tactics.
- Factory method (getStrategy) for obtaining specific strategy instances.
- Concrete Strategy Classes implement specific strategies (HumanStrategy, RandomStrategy, etc.) Every player strategy displays a distinct behavior.
  - Aggressive: To win the game, the aggressive player employs aggressive maneuvers.
  - Benevolent: The Benevolent player defends territories by making defensive plays and never attacks.
  - Cheater: The Cheater player manipulates the game to conquer adjacent countries without making any legitimate game moves.
  - Human: In order to advance in the game, a human player needs to interaction with other users.

Adapter Pattern:



- Using an adapter pattern makes it easier to create a link between two objects with comparable features but distinct uses.
- We have built an adapter class that serves as a bridge between the readers of the Conquest and Domination maps; the target is the Domination map, and the adaptee is the Conquest map reader.
- Class Hierarchy:
  - Adaptor: The Adapter Pattern is being implemented by this class. By extending the DominationMap class, it implies that it want to modify ConquestMap's capabilities to conform to the interface that DominationMap requires.
  - ConquestMap: It appears that this class requires adaptation because of its conflicting interface.
  - DominationMap: The target interface to which the class Adaptor is attempting to adapt.

- Adapter Methods:
    - readMap: This feature is included in the modified user interface. To provide the implementation for the readMap method in the DominationMap interface, it invokes the readMap method of the ConquestMap instance (d_adp).
    - saveMap: This function is also a component of the modified interface. To provide the implementation for the saveMap function in the DominationMap interface, it invokes the saveMap method of the ConquestMap instance (d_adp).
- The adapter class creates an object. The adapter replaces the target class method and returns the Conquest map type object.
- In this manner, it is possible to read maps of dominance and conquest without actually altering the logic of either.
- Hence, with the adapter design, GameMap can now read from both Domination and Conquest map formats.