Architecture Diagram Build III

For the fulfillment of SOEN 6441

Submitted To: Prof. Joey Paquet

Submitted By: Group 22

Team Members	
Student ID	**Name**
40289562	Inderjeet Singh Chauhan
40292088	Saphal Ghimire
40256283	Apoorva Sharma
40238072	Vibhor Gulati
40279035	Mohammad Zaid Shaikh

Introduction:

The architectural design and implementation of the Java-based strategy game Warzone, which was developed as a component of the SOEN 6441 project, are detailed in this report. Model-View-Controller (MVC) is a prevalent software development framework utilized by Warzone to organize code and decouple concerns. Furthermore, four fundamental design patterns are integrated into the game: Command, Observer, Strategy, and Adapter. These patterns are of utmost importance in the organization of the codebase and the improvement of the game's functionality. The purpose of this report is to furnish a comprehensive analysis of Warzone's architecture and the manner in which design patterns were implemented throughout the undertaking.

Design Patterns:

Design patterns are solutions to frequent issues that arise during the design and development of software that can be reused. These frameworks offer a methodical strategy for addressing particular challenges, fostering the reuse, maintainability, and scalability of code. By utilizing design patterns, software developers can effectively resolve recurring issues and adhere to established standards in the field of software design.

When considering the Warzone initiative, the following four design patterns have been implemented:

1. **Command Pattern:** By encapsulating a request as an object, the pattern of command enables clients to parameterize queued queries. This pattern decouples the sender of a request from its receiver, enabling extensible and flexible management of commands. In Warzone, the Command pattern is employed to manage player actions such as reinforcements, attacks, and fortifications, facilitating the implementation of undo/redo functionality and supporting a robust command execution system.

2. **Observer Pattern:** The Observer pattern defines a one-to-many dependency between objects, where changes in one object initiate updates in its dependent objects. This pattern fosters loose coupling between components, facilitating efficient communication and synchronization. In Warzone, the Observer pattern is utilized to notify participants of changes in the game state, such as territory ownership updates, reinforcements, and phase transitions, ensuring timely updates and a synchronized gaming experience.

3. **Strategy Pattern:** The Strategy pattern defines a family of algorithms, encapsulates each algorithm, and makes them interchangeable. This pattern enables algorithms to vary independently from clients that use them, promoting flexibility and enabling runtime selection of algorithms. In Warzone, the Strategy pattern is employed to implement various gameplay strategies, such as reinforcement strategies, attack strategies, and

fortification strategies. By encapsulating these strategies, Warzone facilitates dynamic strategy selection based on game conditions and player preferences.

4. **Adapter Pattern:** The Adapter pattern permits incompatible interfaces to work together by converting the interface of one class into another interface that clients expect. This pattern promotes interoperability and facilitates integration between disparate systems. In Warzone, the Adapter pattern is used to adapt external APIs or systems, facilitating seamless interaction with external services such as map generation utilities or player data repositories. By encapsulating the interaction logic within adapters, Warzone ensures compatibility and facilitates the incorporation of external functionalities into the game system.

These design patterns play a crucial role in shaping the architecture of Warzone, contributing to its modularity, extensibility, and maintainability. By adhering to established design patterns, Warzone demonstrates a structured and scalable approach to software design, augmenting its overall quality and usability.

Project Usage:

1. Adapter Pattern:

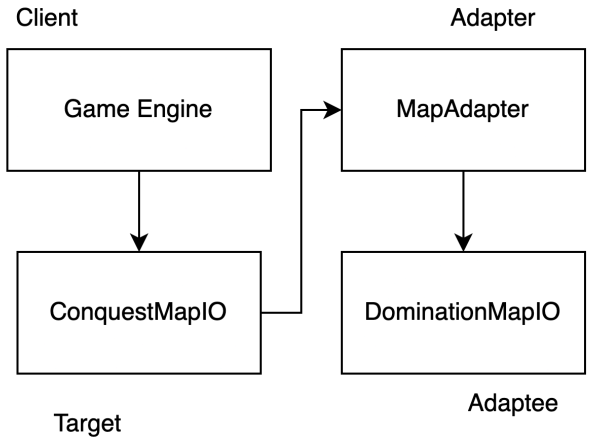

Figure: Adapter Pattern

In the Warzone project, the Adapter pattern plays a pivotal role in seamlessly integrating the Domination map format into the existing game engine, which predominantly supports the Conquest map format. The Conquest map input/output (IO) functions as the target interface, representing the standard input/output operations expected by the game engine. On the other hand, the Domination map IO functions as the adaptee, encapsulating the specific input/output operations required for the Domination map

format. To bridge the divide between these incompatible interfaces, a MapAdapter is introduced. This adapter translates the queries and responses between the game engine and the Domination map IO, ensuring compatibility and interoperability. As a result, the game engine can effectively employ the Domination map format without necessitating any modifications to its existing functionality or architecture. Through the Adapter pattern, Warzone obtains flexibility and extensibility in supporting multiple map formats, enhancing the versatility of the game and enriching the player experience.

2. Command Pattern:

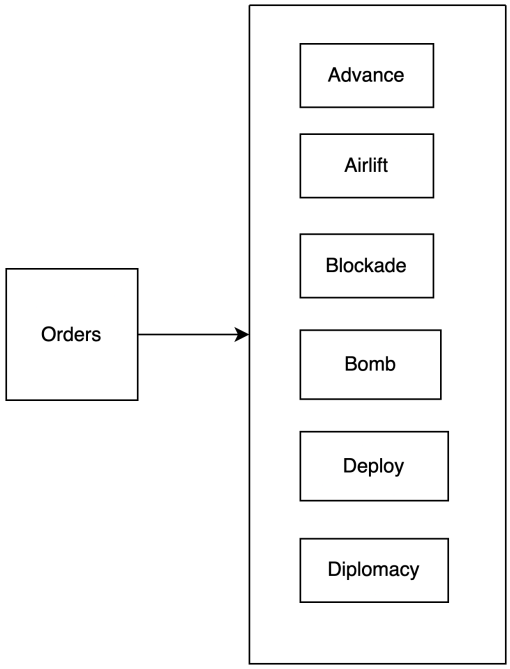

Figure: Command Pattern

In the Warzone project, the Command pattern functions as a fundamental component of the game's architecture, enabling the execution of player actions through a structured and flexible approach. Specifically, the Command pattern is implemented to manage the six distinct orders—Advance, Airlift, Blockade, Bomb, Deploy, and Diplomacy—each encapsulated as a command object. When a player issues an order, such as deploying armies to a territory or initiating a blockade, a corresponding command object is created and executed. This decouples the sender of the command (the player) from the receiver (the game engine), enabling for seamless integration and extensibility. Moreover, the Command pattern facilitates additional functionalities such as undo/redo capabilities,

command records, and asynchronous execution, enhancing the overall gameplay experience. By leveraging the Command pattern, Warzone accomplishes modularity, scalability, and maintainability in handling player actions, providing a robust framework for strategic decision-making within the game.

3. State Pattern:

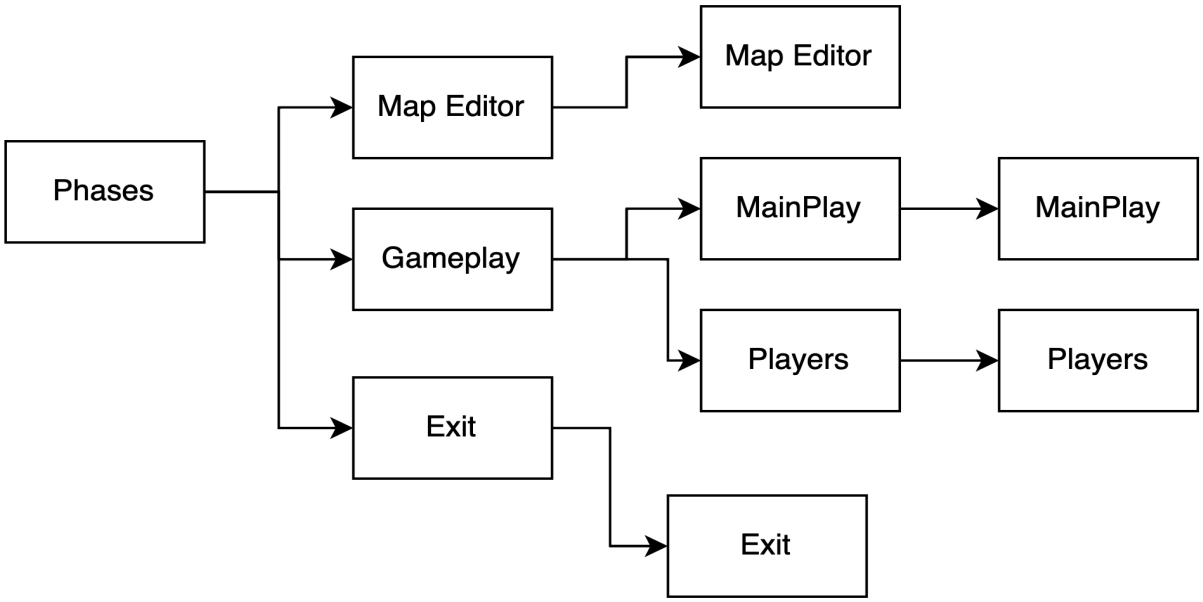

Figure: State Pattern

In the Warzone project, the State pattern is employed to manage the three distinct esources. By leveraging the State pattern, Warzone accomplishes modularity and flexibility in managing the game's lifecycle, allowing for seamless transitions between different phases while encapsulating the state-specific functionalitiephases of the game: MapEditor, Gameplay, and Exit. Each phase represents a distinct state of the game, with specific functionalities and behaviors associated with it. Within the MapEditor phase, players can construct or modify game maps using the MapEditor component, which encapsulates the logic for map editing operations. Transitioning to the Gameplay phase activates the MainPlay component, where the primary gameplay mechanics unfold, including territory conquest, troop deployment, and strategic decision-making. Additionally, the Players component within the Gameplay phase manages user interactions and orchestrates gameplay events. Finally, the Exit phase facilitates the graceful termination of the game session, assuring proper cleanup and closure of rs within dedicated components. This design approach promotes code maintainability,

scalability, and legibility, facilitating the development and evolution of the Warzone game system.

4. Observer Pattern

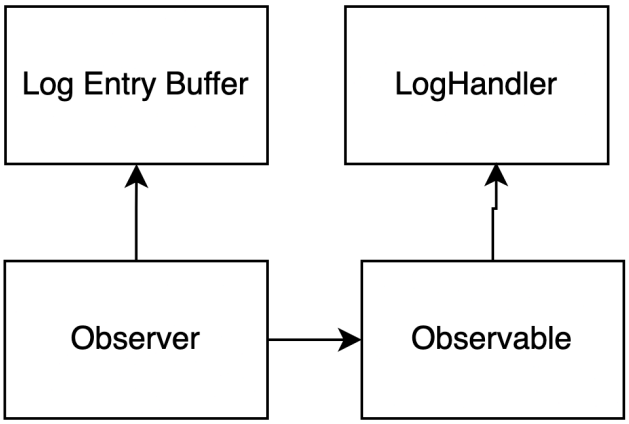

Figure: Observer Pattern

In the Warzone project, the Observer Pattern is effectively utilized to establish communication between objects while maintaining loose coupling. The Observable class serves as the subject, managing a list of observers and facilitating notifications upon state changes. As an interface, Observer defines the method for observers to respond to updates. LogEntryBuffer extends Observable, representing a concrete subject responsible for logging entries and notifying observers of modifications. LogHandler acts as a specific observer, reacting to updates from LogEntryBuffer by writing entries to a file. This design promotes adaptability and extensibility, allowing multiple observers to dynamically subscribe and unsubscribe without impacting the subject or other observers. By segregating responsibilities and minimizing dependencies, the Observer Pattern enhances maintainability and scalability within the system, facilitating seamless adjustments and enhancements as needed in response to evolving requirements or additional functionalities.

Architecture Diagram:

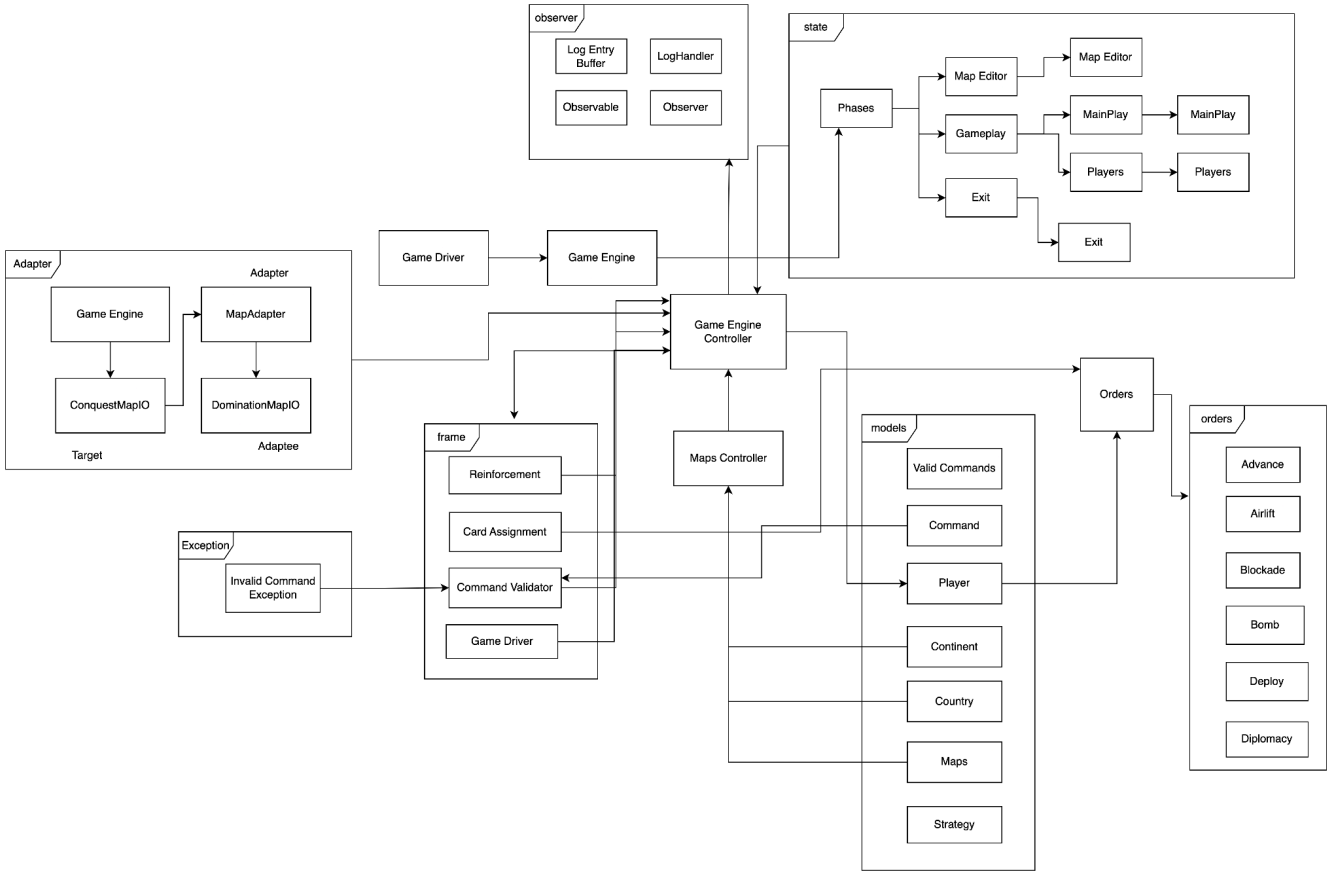

Figure: Architecture Diagram