SOEN 6441 - Advanced Programming Practices

Architectural Design

Project - Risk Game (Build #2)

Fall 2023

Submitted by:-

Rishi Ravikumar
Anuja Ajay Somthankar
Yusuke Ishii
Nimisha Mavjibhai Jadav
Abhigyan Singh

Submitted to:-

Dr. Amin Ranj Bar

Summary of Build 2

- State Pattern for representing and navigating the game phases. The State Pattern is used within the game to model and manage various game phases (e.g., InitMapPhase, IssueOrderPhase), allowing for a clean and organized transition between them.

- Command pattern to create and execute out the orders given by the players. The Command Pattern is used to encapsulate player actions as commands, making them easier to execute, and manage, and thus increasing flexibility and extensibility.

- The Observer Pattern to generate game logs that notify and update interested observers (e.g., players, loggers) when significant game events occur, ensuring real-time record-keeping.

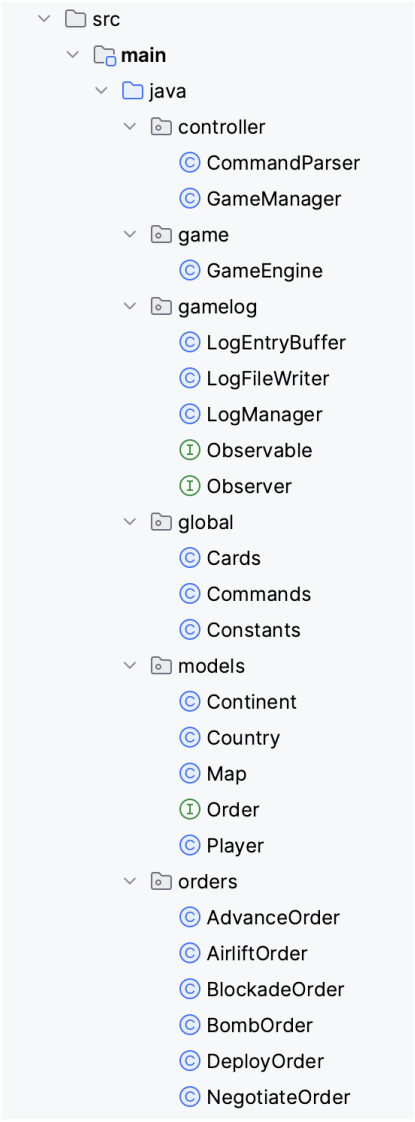
- Continuous Integration checks on the repository have been updated. Continuous Integration (CI) checks have been improved and integrated into the repository to validate code changes automatically, ensuring that new code is consistent, buildable, and passes relevant tests before being merged.

- The codebase is rigorously tested with JUnit5, a popular Java testing framework, to verify the correctness and reliability of individual units of code, thereby promoting software quality and robustness.

- The codebase is extensively documented using JavaDoc, providing comprehensive and easily accessible documentation to assist developers and maintainers in understanding the structure, purpose, and usage of the code.

- Implemented a test suite which entailed grouping unit tests logically based on criteria such as functionality, modules, or components. This enables efficient test execution and reporting for specific areas of your codebase.

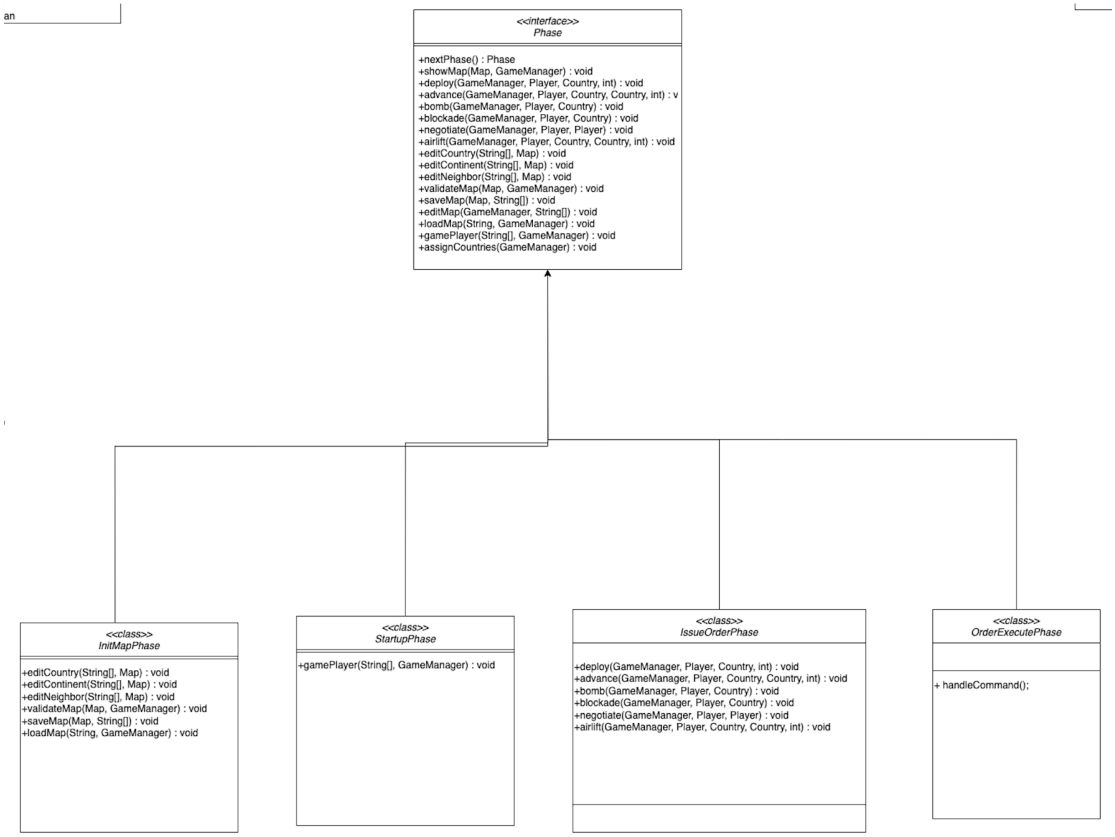
Package Structure



Design Description

1. *State Pattern*

To represent each phase of the game, we created a new class called Phase. The GameEngine class will serve as the context class, coordinating and transitioning between phases. Each Phase class will encapsulate the phase's specific behavior and logic.



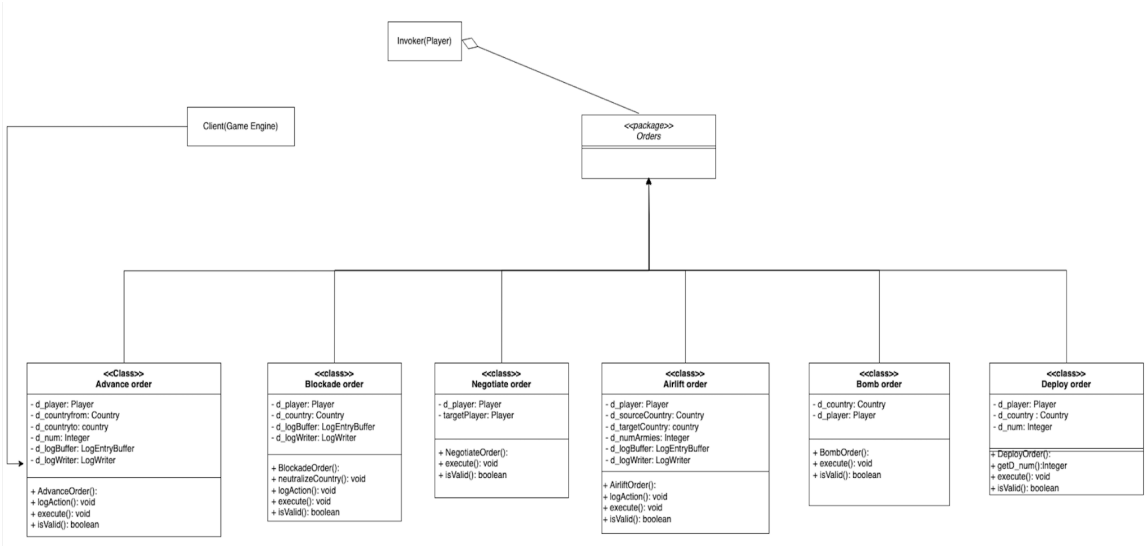
- ➔ Context is GameEngine class.
- ➔ State is Phase class.
- ➔ ConcreteSate consists of InitMapPhase, StartupPhase, IssueOrderPhase, OrderExecutePhase classes.

1. The central coordinator who manages the current game phase and players is known as the GameManager.
2. The game begins with the InitMapPhase, during which players can create and edit the game map, add and remove players, and assign countries.

3. The game moves to the IssueOrderPhase once the map is complete and players are assigned countries.
4. During the IssueOrderPhase, each player issues orders in turn, and the game progresses.
5. Players can issue orders such as armies deploying, armies advancing, and other strategic actions.
6. The ExecuteOrderPhase processes player orders and updates the game state.
7. Following the execution of orders, the game returns to the IssueOrderPhase in preparation for the next player's turn.
8. This cycle continues until a win condition is met, at which point the game is over.

2. *Command Pattern*

The Command Pattern is applied in the GameEngine context to manage player orders as distinct commands. Each order is contained within an Order class, and the Player class serves as the invoker, queuing orders as they are issued by the player. The client class is the GameEngine, which retrieves and executes orders via the next_order method. When the Order objects are executed, they carry out the corresponding actions.

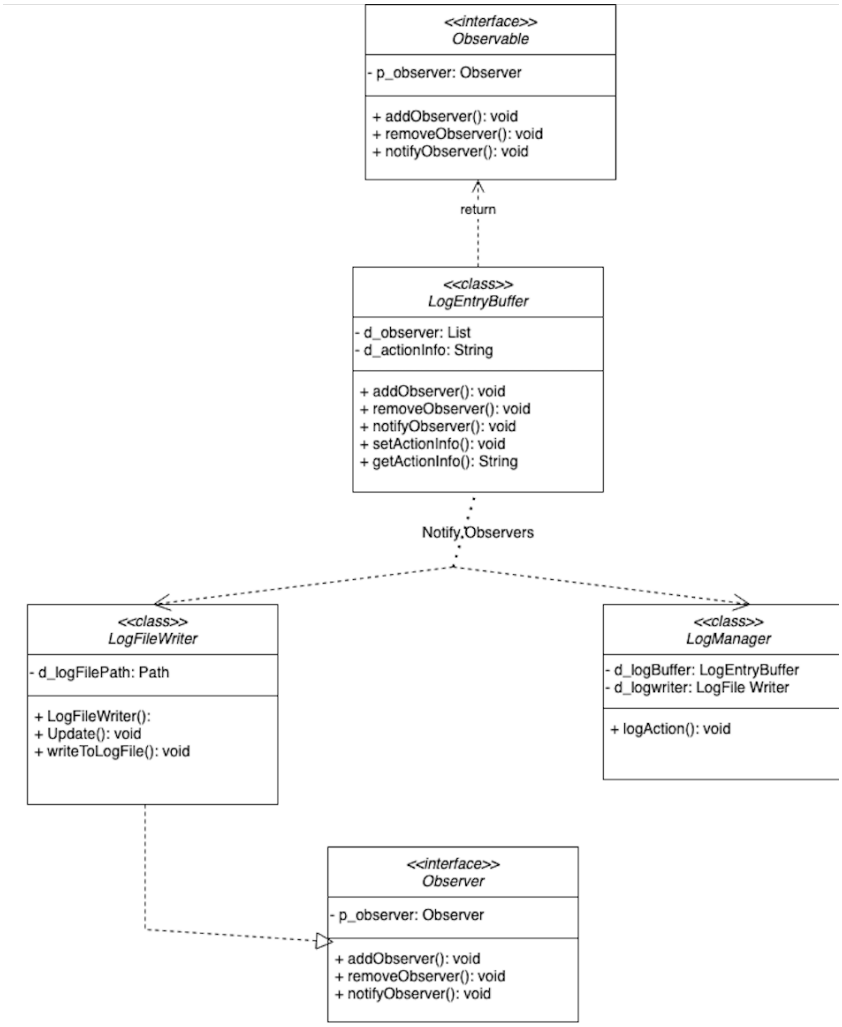


- ➔ Invoker is a Player class.
- ➔ Client is a GameEngine class.
- ➔ Command is Order class.
- ➔ ConcreteCommand are AdvanceOrder, BlockadeOrder, NegotiateOrder, AirliftOrder, BombOrder, DeployOrder classes.

1. The issue_order() function is called by the Player, and the commands are read in a round robin fashion. When incorrect commands are entered, the user is prompted to re-enter them.
2. Once all orders have been created, the game engine will call the execute() method from the Order interface in accordance with the state pattern.
3. Each player uses the next_order() method to get the orders and the execute() method to execute them in a round robin fashion.

3. Observer Pattern

The Observer pattern is used in the context of implementing a game log file to establish a one-to-many relationship between objects, allowing the state of one object (the LogEntryBuffer) to be observed and monitored by multiple other objects (the log file writer). The LogEntryBuffer class serves as the subject (Observable) and stores information about game actions. When this information changes, it alerts its observers (the log file writer) so that the changes can be captured and recorded in a log file.



1. When an action needs to be logged (for example, during gameplay), the LogManager is used to do so.
2. By calling setActionInfo, the LogManager updates the LogEntryBuffer with the action information.
3. The LogEntryBuffer updates its observers (in this case, the LogFileWriter).
4. The LogFileWriter receives the update, extracts the action information, and appends it to the log file that was specified when it was initialized.