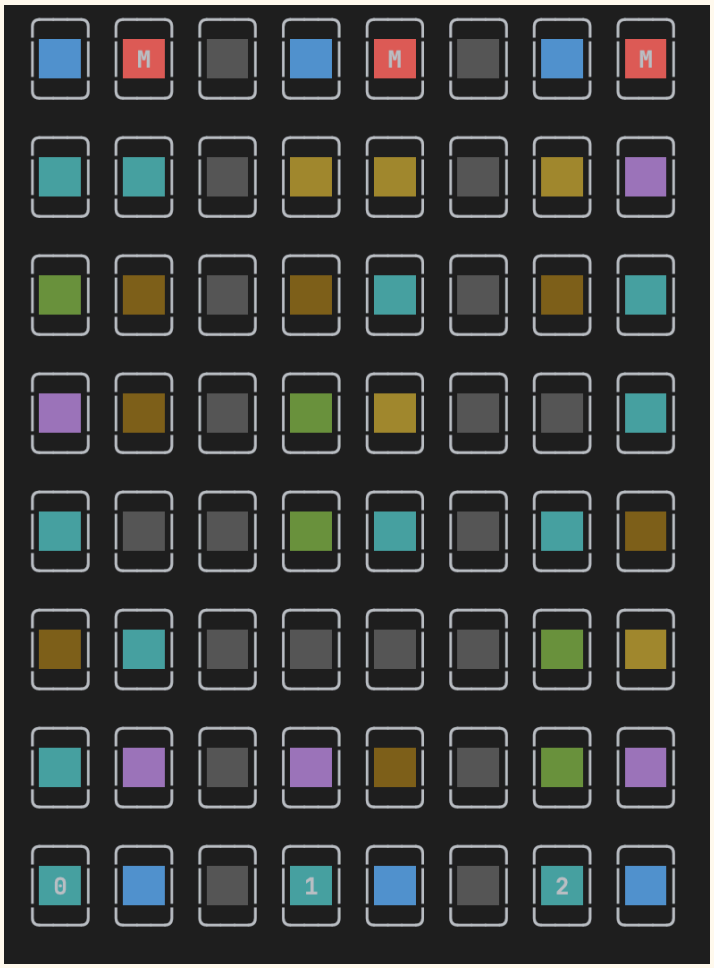
CS611
Object Oriented Software Principles and Design

Legends of Valor

Final Project - Design Documentation



Team Members

Sanjana
Priyanshu Bansal
Mengxing Wang

Project Overview

This final project is an integration of two games under a single executable program:

- **Legends: Monsters & Heroes** (a classic turn-based game focussed on exploration and combat)
- **Legends of Valor** (a lane-based board game focussed on positioning, terrain effects and reaching the enemies nexus)

The project provides a unified entry point that allows the user to select which game to play, while reusing shared entities and infrastructure such as heroes, monsters, party formation, and common I/O utilities.

Primary Design Goals

- **Integration :** Cleanly combine Monsters and Heroes (MH) and Legends of Valor (LoV) without duplicating logic.
- **Separation :** Keep UI, control flow, game rules, and world state independent to enable reuse across MH and LoV modes.
- **Extensibility :** Use of inheritance, interfaces, commands, and rules to allow new heroes, monsters, spells, tiles, and game modes with minimal changes.
- **Rule Driven Game :** Add rules based on the games to allow smooth working of the game.
- **Consistency :** Ensure shared mechanics like combat, spells, potions, etc and behave uniformly in both the games.

Scalability

This final project scales in terms of features, game modes and content without requiring much structural changes to the existing code. At the top level, the use of a unified launcher and a common GameMode interface allows additional game modes to be added without modifying existing logic.

At the gameplay level, scalability is achieved through separation of concerns and abstraction. Core entities such as heroes, monsters, items, and parties are shared across both games, so

expanding hero classes, monster types, spells, or items automatically benefits all modes. At the same time, each game owns its own world class, ensuring that mode-specific rules can grow in complexity without affecting the other game.

Within each game mode, scalability is further supported by rule-driven and effect-based designs. Movement rules, tile effects, and combat behaviors are encapsulated in dedicated classes, making it easy to extend or replace mechanics without rewriting the world logic.

This structure allows the code to be modular and robust, open to changes.

Extendability

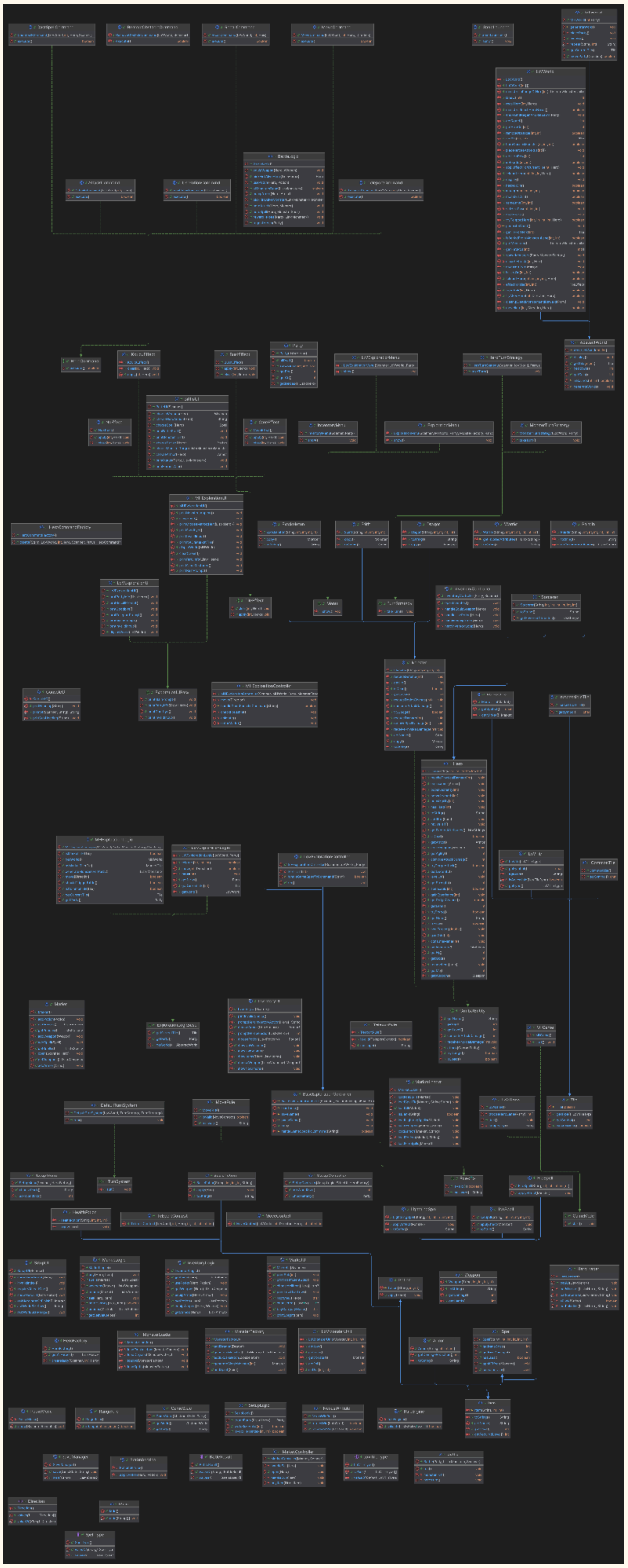
The final project makes sure that extension is possible without completely modifying the existing stable code. This is achieved through the consistent use of interfaces, abstract base classes, and polymorphism across the system. New behavior is added by extending abstractions rather than rewriting logic.

At the game level, extensibility is enabled by the GameMode interface. Any new game variant can be introduced by implementing this interface and plugging it into the launcher, without altering the internal logic of MH or LoV. This makes the architecture open to future expansion while remaining closed to regression.

At the domain level, core hierarchies such as Hero, Monster, Spell, Potion, and TileEffect are designed for subclassing. New hero classes, monster types, spells, potions, or tile effects can be added by creating new subclasses that override well-defined hooks without changing existing classes. Similarly, rule based systems in LoV like movement rules, teleport rules, range rules, etc allow new constraints or mechanics to be added by introducing new rule classes rather than modifying world logic.

Therefore, the system follows an open closed design principle where functionality is growing through extension.

UML Diagram



Design Pattern Choices

Command Pattern - Commands Package

Motivation

Legends of Valor requires the user to repeatedly issue actions such as moving, attacking, casting spells, teleporting, recalling, using items, or removing obstacles. If all of these actions were implemented directly inside the controller using long chains of if/else conditions, the controller would become hard to read, hard to test, and difficult to extend. Therefore, we adopted the Command Pattern to represent each action as a self contained object. This allows the controller to remain small and uniform, supports adding new commands with minimal changes, and enables a clean approach to handling invalid commands most importantly, an invalid command should not automatically consume a turn.

Implementation in Code

The command system is defined by a shared interface game/exploration/commands/HeroCommand.java, where each command implements boolean execute(). The return value is a deliberate design choice: true indicates a valid command that consumes the turn, while false indicates an invalid command and allows the player to retry without losing the turn. Each concrete action is implemented as a separate command class in game/exploration/commands/ (for example, MoveCommand, AttackCommand, CastSpellCommand, TeleportCommand, RecallCommand, UsePotionCommand, and RemoveObstacleCommand). Each command stores only the context it needs such as references to LoVWorld, the hero index, the hero instance, and sometimes input utilities and delegates real rule enforcement to the correct receivers. To prevent the controller from becoming responsible for command mapping, we use HeroCommandFactory.java to translate raw user input into the appropriate command object. Finally, LoVExplorationController.java functions as the **Invoker**: it reads input, asks the factory for a command, calls execute(), and ends the turn only if the command returns true.

Why This Design Works Well

This Command Pattern design provides strong extensibility: adding a new action typically requires adding a new command class and a single mapping line in the factory, without rewriting controller logic. It also improves correctness and user experience by explicitly supporting **“invalid input → retry”** through the boolean execution result. In addition, it improves testability: each command can be instantiated and tested independently by calling execute() with controlled world/entity state, without running the entire game loop.

Strategy Pattern - Effects Package

Motivation

In Legends of Valor, special tiles (Bush/Cave/Koulou) provide different attribute bonuses to heroes. These tile effects share a common lifecycle (apply when entering, clear when leaving) but differ in their concrete behavior (which attribute changes and by how much). Implementing this using conditionals inside movement logic would tightly couple tile types to world code and quickly become unmaintainable as more tile types or effects are introduced. Therefore, we adopted the **Strategy Pattern** to represent each tile effect as a separate interchangeable behavior.

Implementation in Code

The strategy interface is world/effects/TileEffect.java, which defines apply(int heroIdx, Hero hero) and clear(int heroIdx, Hero hero). Concrete strategies are implemented as BushEffect, CaveEffect, KoulouEffect, and a NoEffect default strategy. The LoVWorld acts as the **Context**: it stores instances of these strategies, selects the correct one based on the current tile type, and ensures the correct timing of effect application and clearing. This keeps the movement logic clean and avoids scattering tile-specific code across the system.

Improvements

Introduced a shared CombatEntity interface to unify hero monster combat behavior. Removed duplicated combat logic by centralizing attack, dodge, and damage flow. Cleaned hero and monster responsibilities by pushing mode specific logic out of entities. Replaced conditional spell handling with polymorphic spell subclasses.

Contribution

- **Sanjana** - board and world layer for LoV, including world rendering, hero positioning, occupancy rules, and movement integration. Worked on improving monsters and heroes based on the feedback. Documentation work. Structure of the code.
- **Priyanshu** - combat layer, including attack and damage formulas, monster behavior logic, and hero abilities. Teleportation implementation. Structure of the code.
- **Mengxing** - systematic testing, identifying and fixing bugs and logical mistakes, expanding and validating the market system, and checking edge cases to ensure correctness across both game modes. Documentation work.