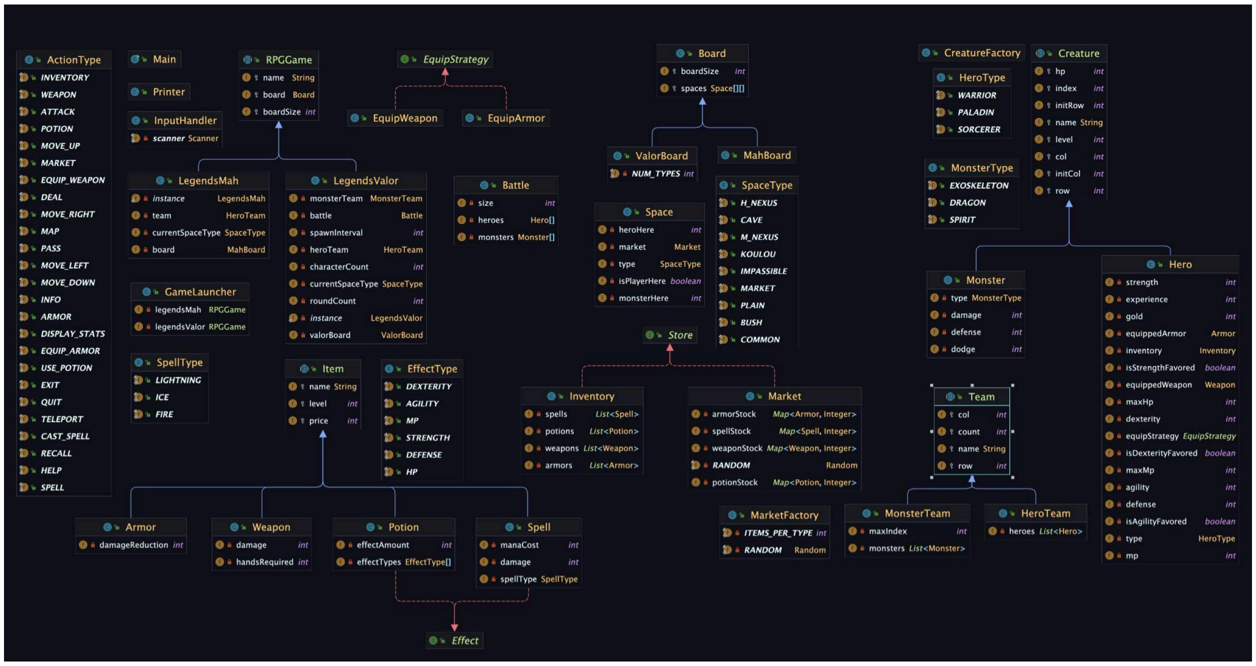
Design Documentation

Yifei Bao, Tingxuan Tang

I. Class Structure



In our design, there are 4 main components of the RPG game that are represented as folders of classes: creature, item, store and world. The creature folder contains classes all about game characters. The item folder contains classes about items in this game like potions, armors. The store folder contains classes all about storage like inventory and market. The world folder contains classes about the world board like board, space. Then it comes the game logic class folder game and other folders action, battle, interfaces, strategy, util. The game folder contains game classes like RPGGame, LegendsValor. The action folder contains classes about user action like ActionType, InputHandler. The battle folder contains Battle class for battle logic in the game. The interfaces folder contains interfaces like Effect, Store. The strategy folder contains strategy classes like EquipArmor, EquipWeapon. The util folder contains Printer class to print messages.

We chose Yifei's structure as the base structure and incorporated the strengths of Tingxuan's structure to make improvements. In Yifei's structure, similar classes are grouped into folders, making the structure more organized. This design extensively uses Enums to categorize different Heroes, Monsters, Items, and Spaces instead of using subclasses, as the differences between these types are tiny, which efficiently reduces similar and unnecessary subclasses and makes the program more clear. Additionally, user input actions are also converted to Enums to enhance the code readability. Input and output handling is separated into helper classes. The creation of Heroes, Monsters, and Markets is implemented using the Factory Pattern. All of these features are reasons why we chose Yifei's design.

On the other hand, Tingxuan's design introduces great abstraction for the game and board class, and it also applies the Strategy Pattern. At the same time, it also allows players to choose heroes they like. We integrated the strengths of both structures and complemented their functionalities to create the current structure.

We mainly use inheritance and Singleton Pattern to implement the new game and make sure the previous game and structure can be modified as little as possible. For example, the specific board for Legends: Valor can be achieved by creating a ValorBoard class inheriting Board class, and the main game logic can be implemented in LegendsValor class inheriting RPGGame class. At the same time, Singleton Pattern ensures that both of the games have only one instance.

II. Scalability and Extensibility

Creature, Team, Board, Item, RPGGame are all basic classes designed to be easily extended. For example, in the two Legend games, boards are all extended from Board class. Classes in the creature folder are all directly used in the two games by certain extensions without modification and you can easily create a new creature by extending the creature class.

The EquipStrategy interface and its concrete implementations provide a flexible way to add new equipping strategies to the game. You can create a new implementation of the EquipStrategy interface to define a new way of equipping items to creatures.

The Effect interface and its concrete implementations provide a flexible way to add new effects to the game. You can create a new implementation of the Effect interface to define a new effect that can be applied to creatures.

Separate InputHandler and Printer classes allow for reusable methods for user action input and terminal output.

III. Design Pattern

1. Factory Pattern

There are two factories in our program, CreatureFactory and MarketFactory, which are used to create characters and markets respectively.

The CreatureFactory abstracts the creation of Hero and Monster objects. It encapsulates the logic required to create instances of Hero and Monster, loading their configurations from external files or initializing them with specific attributes based on their types.

The MarketFactory manages the creation and initialization of the Market object with appropriate Items. It ensures that the Market is consistently populated with items, hiding the complexity of loading and managing stock from other parts of the program.

Both CreatureFactory and MarketFactory centralize the object creation logic. At the same time, the program does not directly instantiate Hero, Monster, or Market objects, nor does it manually manage item inventories. This reduces the coupling between components and makes it easier to adjust object creation logic without impacting other parts of the code. And the factories can be reused across the program whenever new instances of Hero, Monster, or Market are needed. This avoids duplicate code and ensures consistency and reusability.

What is more, adding new types of heroes, monsters, or items (e.g., a new HeroType or MonsterType) requires changes only to the factories, not the client code. This makes the program easier to extend, which enhances the flexibility and scalability.

2. Strategy Pattern

We apply Strategy Pattern in achieving equipping items logic.

The EquipStrategy Interface defines a common interface for all concrete “equip” strategies, which allows for different ways of equipping items (e.g., weapons, armor) to be implemented without affecting the client code that uses the strategy.

EquipWeapon and EquipArmor are concrete implementations of the EquipStrategy interface. Each class provides its own logic for equipping a weapon or armor to a Hero. Then, the Hero class can use a EquipStrategy instance to dynamically decide how to equip an item. This enables flexibility in changing the equipping behavior at runtime by swapping out the strategy.

By using the Strategy Pattern, the Hero class does not need to change when new types of equipping strategies are added, like a new item. You can simply add another implementation of the EquipStrategy interface, which satisfies the Open Closed Principle.

In addition, the logic for equipping weapons and armor is encapsulated in their respective strategy classes, avoiding duplication in the Hero class.

3. Singleton Pattern

In order to ensure that a class has only one instance throughout the application’s lifecycle and provides a global point of access to that instance, in this program, we appay singleton pattern in LegendsValor and LegendsMah class to make sure there is only one instance of each game mode at any time.

Both LegendsValor and LegendsMah classes have a private static field instance to store the singleton instance and a public getInstance() method that returns the single instance, creating it if necessary. The instance field ensures that no matter how many times getInstance() is called, the same object is returned.

The Singleton Pattern avoids potential conflicts or inconsistencies caused by multiple instances. And the singleton instances are created only when they are first needed, optimizing resource usage.