UNIVERSITY OF WATERLOO

Faculty of Mathematics

Project Title

AWAKE

Team Member

Andrew Hua x27hua

Eric Liu e44liu

Yukai Wang y3229wan

Miranda Xie m52xie

Richella Li j2333li

Yixing Gu y232gu

GitHub Link

https://github.com/ErastoRuuuuuush/CS446.git

The final project successfully implements the three functional properties described in the proposal, which are the village, the map of "adventure" (dungeon), and the battle view. MVVM architecture and repository architecture are used for the implementation to support the functional properties.

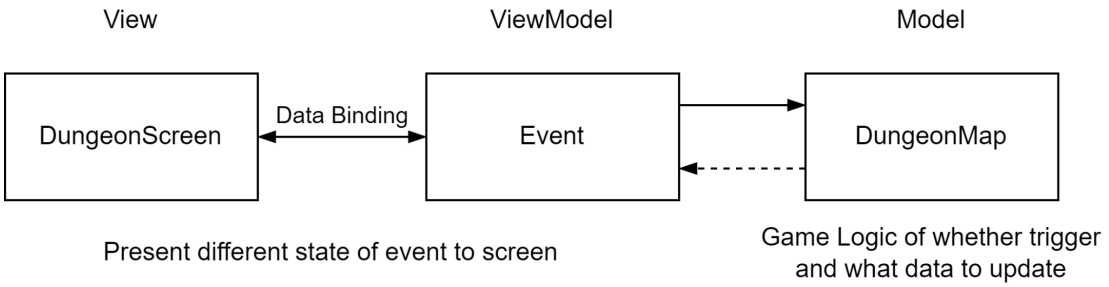
1. MVVM architecture

In the dungeon, a map consisting of different event cards is displayed and most of the event is shown in a state of "unexplored". The map containing event cards is the View component, where UI outlook is decided and receives user input on the screen. This is implemented in the class DungeonScreen in file "com.cs446.awake/ui/DungeonScreen.kt".

Inside the DungeonScreen, the event cards, which have multiple states, are the ViewModel that binds to the View and provides the state of the view component in the Dungeon. The card has a back side indicating "unexplored" and a different front side image indicating a different event triggered. The event state includes "next level", "battle", "item", and "empty" events. The event cards are implemented in the Event class in file "com.cs446.awake/model/Event.kt" as the ViewModel.

When receiving user input, the View component will pass the information to the Model eventually, which holds the actual data of the map event objects, in the project's case, the event cards. The class is implemented as DungeonMap in file "com.cs446.awake/model/DungeonMap.kt". When the View component receives user input, it passes information about which card is clicked on the screen to ViewModel to handle the event. The ViewModel will pass info to the model, which determines whether the card clicked is a valid map explore the movement of the map, and data passing back to ViewModel trigger the corresponding event card to change its state. The state change of cards then will be displayed back on the View. Finally, the View renders the screen again to display the change user made.

A diagram of the Dungeon View is displayed below:

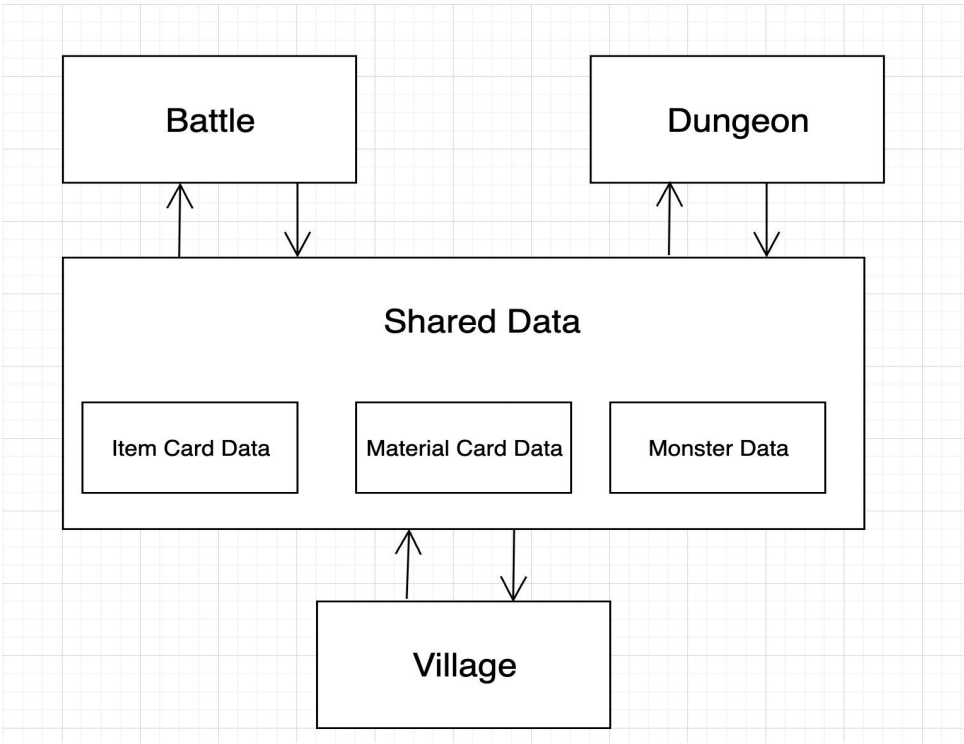


With the implementation of MVVM, different component of the screen can work independently and the change in one component won't affect the implementation of other components. Different developers can work on different component without conflicting each other. If you want to add a new view, for instance, draging instead of clicking, we could simply add a new Screen class that implements the new way of interacting with the users. The model, which is the DungeonMap class will not need to be changed.

The MVVM architecture are also used in Village view. In Village view, for the function properties of bagpack, modify item and merge item, and handbook, we create different View (models). They are class VillageScreen, class BackpackScreen and class MergeScreen. They use the same ViewModel and Model class. The ViewModels are class ItemCard and class MaterialCard, the Models are class ItemCardData and class MaterialCardData. Each <class> are implemented in the corresponding file named "<class>.kt"

Our project also fits the functional properties of the Battle View. Class BattleScreen act as UI frontend and game logic control. Class Enemy , class Player, and class State as game core data. These class can communicate with each other by passing Card class info in between. Besides the new logic of deck reuse, the game logic correctly achieved the proposal idea of Battle View.

2. Repository Architecture



2.1 Overview

Our project supports the data-saving functionality and communication among different maps through designing a shared-data system in:

- com.cs446.awake/model/SharedData.kt
- com.cs446.awake/model/Data.kt
- com.cs446.awake/model/CardData.kt
- com.cs446.awake/model/ItemCardData.kt
- com.cs446.awake/model/MaterialData.kt
- com.cs446.awake/model/MonsterCardData.kt

where SharedData mainly implements saving and loading to local .json files, ItemCardData, MaterialCardData inherited from CardData, CardData and MonsterData inherited from Dara

control saving and loading in games.

The specific implementation of Data is introduced in the later design part.

The SharedData implements two functionalities, `readJson` (for loading progress of the game), and `dumpJson` (for saving progress of the game). The system connects our three maps together and stores our user data in case of crashing and quitting.

`readJson` first checks if a local path to a previously saved .json file exists (storage, dungeon level and battle history) and loads information from these files at the beginning of the game.

`dumpJson` saves storage, dungeon level and battle history in some local files. We have several saving points while quitting the game, entering the dungeon, entering the next level dungeon, winning and losing the battle.

Besides, SharedData also contains player information, the current enemy (if any), the player backpack and a list of possible monsters and materials.

2.2 Functional properties

To achieve the functionalities stated in the initial proposal, the shared data system is used in three main maps of our game.

1. Village View
    a. The first function is storage. The shared data supports the view to display the materials and items that player is carrying. The shared data contains 'storage' and 'backPackItem', which hold items the player has in the village and items player brings into the dungeon. The village part can just access these fields to display items.
    b. The village allows players to use the crafting system as we described in our proposal. The 'MergeScreen' provided the UI to the player and handle the interaction. The new crafted items are added to the shared data, so other models can have access to it. All the information on dungeon and player is also provided from the SharedData.Using the repository architecture can benefit our system. It mediates cooperation between components.
2. Dungeon View
    a. When a player encounters a battle, it reads `monsterInfo` from SharedData according to the current dungeon level to generate an enemy and the battle event.
    b. When a player collects materials in the dungeon, it reads `materialInfo` from SharedData to generate a MaterialCard.
    c. When a player wants to open the backpack of rewarded materials or weapons, the system also requests it from SharedData.
3. Battle View
    a. The dungeon needs to pass the enemy to battle. In order to achieve this, the dungeon will update the enemy variable in the SharedData to be the newly created Enemy object for this specific battle. The battle will then read corresponding monster information from sharedData.
    b. After winning the battle, the system adds the rewarded MaterialCard to the SharedData.

2.3 Non-functional properties

1. Robustness

   During the actual implementation, we use `readJson` and `dumpJson` to save and load game progress. The system reads the record while entering the game (in thevillage map), and saves the progress on local devices every time when entering the dungeon, entering next level dungeon, ending a battle and quitting the game. As we encounter battle frequently, the average saving time is around 3 minutes, which is much less than 5 minutes in initial our proposal.

2. Reliability

   To ensure reliability, once the game data changes, which involves weapon in storage and user backpack, materials in storage and user backpack, dungeon level. As stated in our proposal, our system will immediately modify the data stored in SharedData. Moreover, every time the player logs in to the game, we read data from the last record.

3. Privacy

   Consistent with the initial design, the game does not have the ability to share player information. As a stand-alone game that does not require an internet connection, the game means that it has access to read and write internal player saves, but other than that, the game does not require any other permissions from the player to keep the game running. We use "Gdx.files.local" to save directly in `dumpJson`, without the actual path to files, which also ensures that the software does not make additional changes to the user's local files. In general, the game does not violate the user's privacy.

Additionally, at the beginning of the game design, we wanted to make the game easy to operate and understand, and to make the game quick to play in 10 minutes. In the actual implementation process, due to some technical constraints, we used the manual and the accompanying demo video to let players quickly get started with the game. In the 2-page manual, players can use the index to quickly locate the issues they need (the main blocks are villages, battles, and dungeons) and, for operations that are difficult to describe completely in words (such as dungeon exploration and battles), the demo video will cover them all. At the same time, the tutorial considers clarity while still retaining information for players to explore on their own (such as deck tactics, monster skills, etc.), which also ensures the playability of the game.

In conclusion, our project uses MVVM architecture to design the interaction between the user interfaces and the models, uses repository architecture for different maps to access the shared data and controls progress saving and loading.

Since our App is an offline game, all of these classes will reside on user's phone. This game is built by utilizing the gfx library, but does not require any external services for it to work.

We used the repository architecture, where the three dominant views (villiage, dungeon, battle) will read from the sharedData to decide their own behaviour, and update the shared information while needed. In the class diagram, we can see a MergeScreen, which reads the sharedData for storage items and materials that could be used in crafting, and update the storage when we consume some of them to crate a new item.

There is also a BackpackScreen, which is opened when we want to access storage or gallary in the villiage view. It reads from the sharedData for the current storage and backpack, then display them accordingly. DungeonBagScreen displays the backpack in the Dungeon in a similar way. In addition, EnterDungeonScreen will create a dungeon by reading the current dungeon level that is stored in the sharedData. DungeonScreen will update this shared information when we proceed to the next level in the Dungeon. The DungeonMap also uses the sharedData to selcet monsters for the player to fight and materials to collect. The battleEvent will update the sharedData to inform battle what Enemy and Player it should use in a battle.

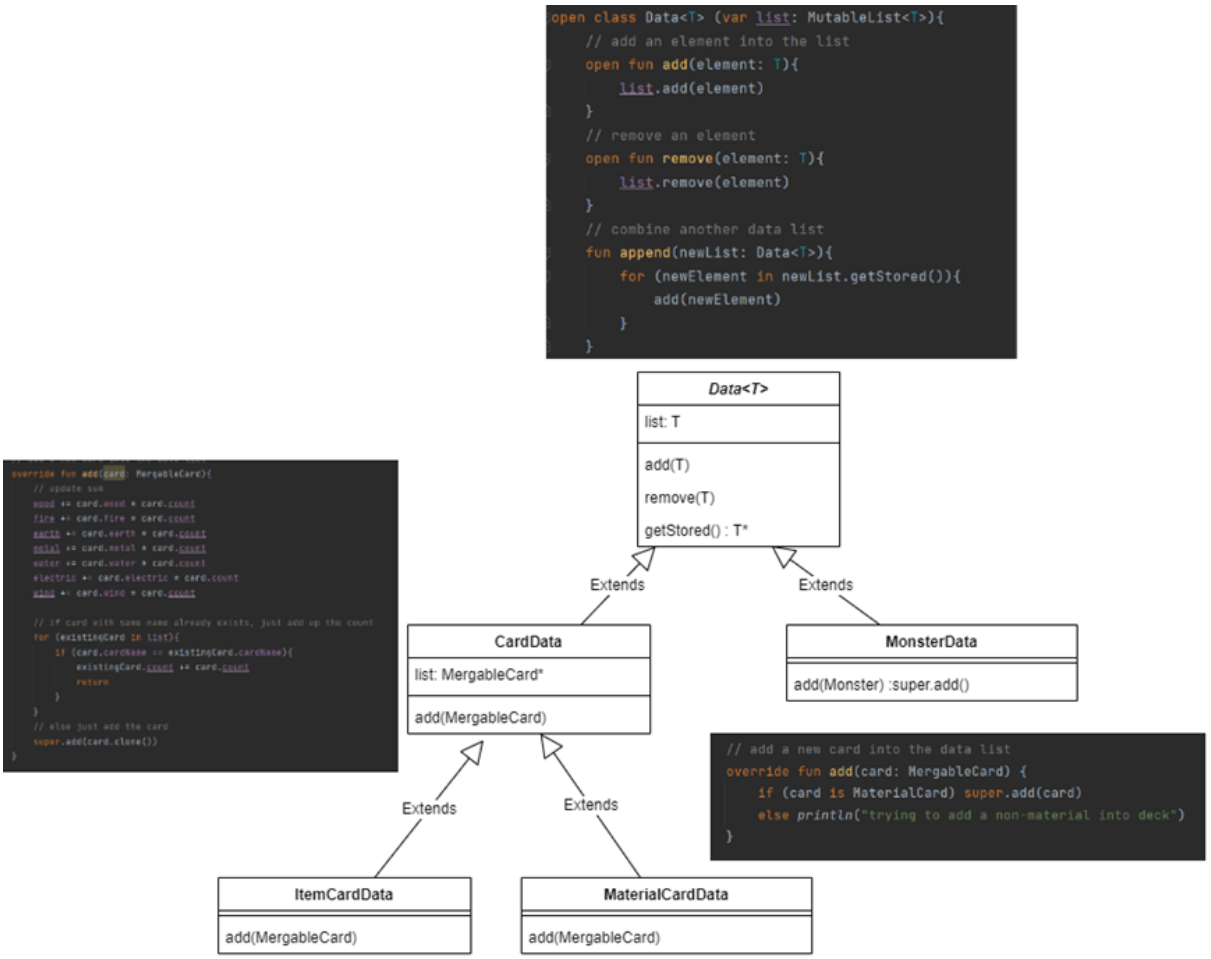
As shown below in the class diagram, there are a lot of abstractions in the design of our system: Character abstracts the classes Player and Enemy, who have different ways of action in their round of battle. MergableCard abstracts the ItemCard and MaterialCard classes, both of which could be used in the merge system, and have the seven element properties. MergableCard is then further abstracted with the ActionCard into the parent class Card, which represents any card that this game has, and keeps track of certain fields that are shared by all cards (i.e. name, count, etc.). An Event class abstracts all the events that might happen in the dungeon, including a CollectEvent, a BattleEvent, or simply an empty Event that does nothing.

The predominant abstraction and inheritance happens for the Data class and its children. A Data class is basically a list of elements with certain type. It provides some common operations on this list, including adding, removing, or selecting an element, and many other operation. One of its child class is MonsterData, which simply stores a list about Monster objects. Another subclass is the CardData class, which contains a list of mergable cards, and keeps track of more properties while also providing more functionalities. The CardData is further inherited by ItemCardData and MaterialCardData which deals with the specific types of MergableCard.

We could of course choose to break these specific classes apart and let them stand on their own, but by abstracting them we can reuse some common features and reduce coupling. In particular, this inheritance structure allows us to make use of many design patterns.

The most commonly used pattern in this project is the template pattern. For Instance, the Data class provides the append(Data<T>) method, which should combine the list in the input Data into the current list. The implementation for this function in Data will iterate through all the elements returned by the getStored() method of the given input, and call add() method on each of them. The subclasses, ItemCardData, MaterialCardData, MonsterData all implements different add() method, but there is no need to specifically define the append()

function for any of them, as they share the template created in the parent class. Notice that if we define more subclasses that have different new ways of adding new elements, the append method will still work.



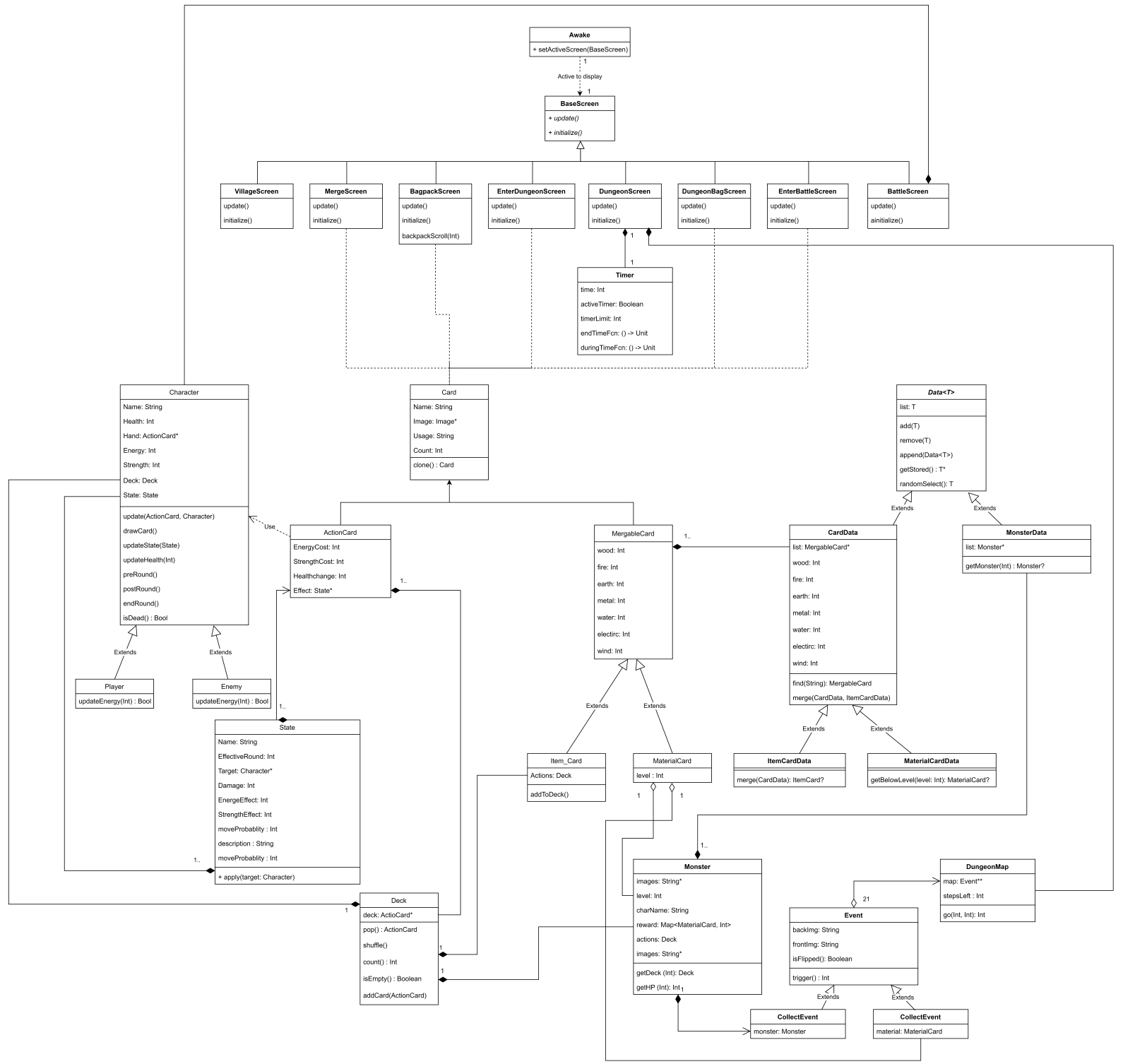
Another possilble approach of implementing the append function is just to overwrite the method for every sub-class, and let them to decide how to append another list to another. However, since all appending is essentially adding all elements in one list to another, we could combine the behaviour in a template, which reuses the code and is easier to maintain or accomodate changes in the future.

Another design pattern that is used for the CardData class is Adapter. Notice that the constructor of CardData requires a list of MergableCard, while a list of ItemCard or a list of MaterialCard is required for its two subclasses. This is simply adapted by converting the specific list to the general type list in construction. A more interesting example will be the merge function of CardData and ItemCardData. The ItemCardData class only takes one parameter which is the given input list of cards to be merged, while the CardData calss takes two parameters, one being the input list, and the second being the possible item outcomes. However, the ItemCardData class knows that itself should be the possible outcome!

The strategy pattern is also used as well. Although Player and Enemy both inherits from the Character class, they uses different strategy in games. The Player will wait for the input from player, and the Enemy will simply use the first card in their hand. Different AI strategy might be added using this pattern without changing much of current code.

Although the factory pattern is not directly used, its idea and structure is seen parallel as how CardData-MergableCard, ItemCardData-ItemCard, MaterialCardData-MaterialCard pairs corresponds just like the factory-product pair for factory pattern.

A possible future change for this system is that we might want to have different kinds of items, i.e. weapons, armors, potions. In order to achieve this, we can simply create three new classes that inherit from the ItemCard, and add specific methods or properties that suit the subclass. We might create three subclass for ItemCardData as well. Notice that we will only have to change and adapt for the ones that are special for these types, and other behaviors like append() that uses these methods will change accordingly to match the desire behavior for this particular type, without explicitly being changed.


Awake
+ setActiveScreen(BaseScreen)
1
Active to display
1
BaseScreen
+ update()
+ initialize()
VillageScreen
update()
initialize()
MergeScreen
update()
initialize()
BagpackScreen
update()
initialize()
backpackScroll(Int)
EnterDungeonScreen
update()
initialize()
DungeonScreen
update()
initialize()
DungeonBagScreen
update()
initialize()
EnterBattleScreen
update()
initialize()
BattleScreen
update()
ainitialize()
1
1
Timer
time: Int
activeTimer: Boolean
timerLimit: Int
endTimeFcn: () -> Unit
duringTimeFcn: () -> Unit
Character
Name: String
Health: Int
Hand: ActionCard*
Energy: Int
Strength: Int
Deck: Deck
State: State
update(ActionCard, Character)
drawCard()
updateState(State)
updateHealth(Int)
preRound()
postRound()
endRound()
isDead() : Bool
Use
Card
Name: String
Image: Image*
Usage: String
Count: Int
clone() : Card
Data<T>
list: T
add(T)
remove(T)
append(Data<T>)
getStored() : T*
randomSelect(): T
Extends
Extends
ActionCard
EnergyCost: Int
StrengthCost: Int
Healthchange: Int
Effect: State*
1..
MergableCard
wood: Int
fire: Int
earth: Int
metal: Int
water: Int
electirc: Int
wind: Int
1..
CardData
list: MergableCard*
wood: Int
fire: Int
earth: Int
metal: Int
water: Int
electirc: Int
wind: Int
find(String): MergableCard
merge(CardData, ItemCardData)
MonsterData
list: Monster*
getMonster(Int) : Monster?
Extends
Extends
Player
updateEnergy(Int) : Bool
Enemy
updateEnergy(Int) : Bool
1..
State
Name: String
EffectiveRound: Int
Target: Character*
Damage: Int
EnergeEffect: Int
StrengthEffect: Int
moveProbablity : Int
description : String
moveProbablity : Int
1..
+ apply(target: Character)
Extends
Extends
Item_Card
Actions: Deck
addToDeck()
MaterialCard
level : Int
1
1
Extends
Extends
ItemCardData
merge(CardData): ItemCard?
MaterialCardData
getBelowLevel(level: Int): MaterialCard?
1..
Monster
images: String*
level: Int
charName: String
reward: Map<MaterialCard, Int>
actions: Deck
images: String*
getDeck (Int): Deck
getHP (Int): Int
1
DungeonMap
map: Event**
stepsLeft : Int
go(Int, Int): Int
21
Event
backImg: String
frontImg: String
isFlipped(): Boolean
trigger() : Int
Extends
Extends
CollectEvent
monster: Monster
CollectEvent
material: MaterialCard
1
Deck
deck: ActioCard*
pop() : ActionCard
shuffle()
count() : Int
isEmpty() : Boolean
addCard(ActionCard)
1
1
1