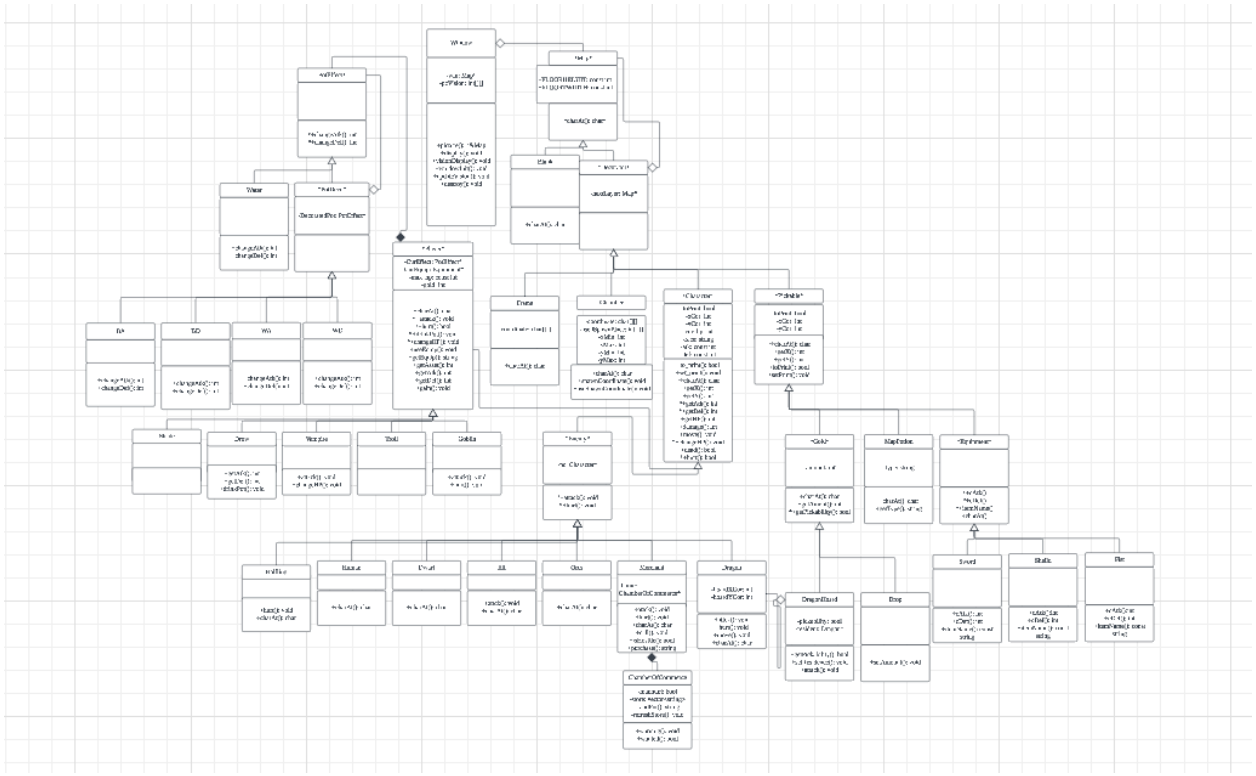
Mingyi Su, Jeannie Shi, Donghui Yu

Overview:

In our project, we have employed the decorator pattern throughout almost every class. By adopting the decorator pattern, we achieve convenient object spawning, smooth object movement, and efficient window display. (For a more detailed view of the decorator pattern, please refer to the UML section.)

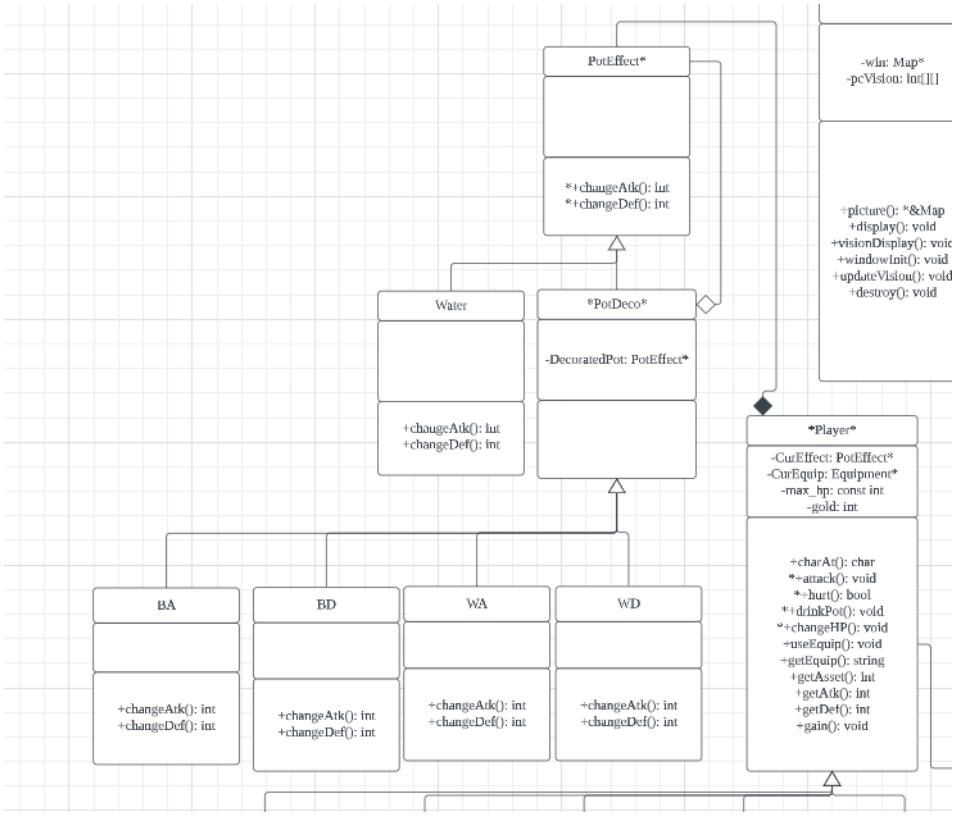
To show our project, we have organized it into five major modules(in the design section), each contributing to a crucial aspect of the game's functionality. These modules encapsulate the core mechanics of generating game elements, handling object movements, ensuring effective object visibility, managing combat interactions, and enabling item collection.

UML:

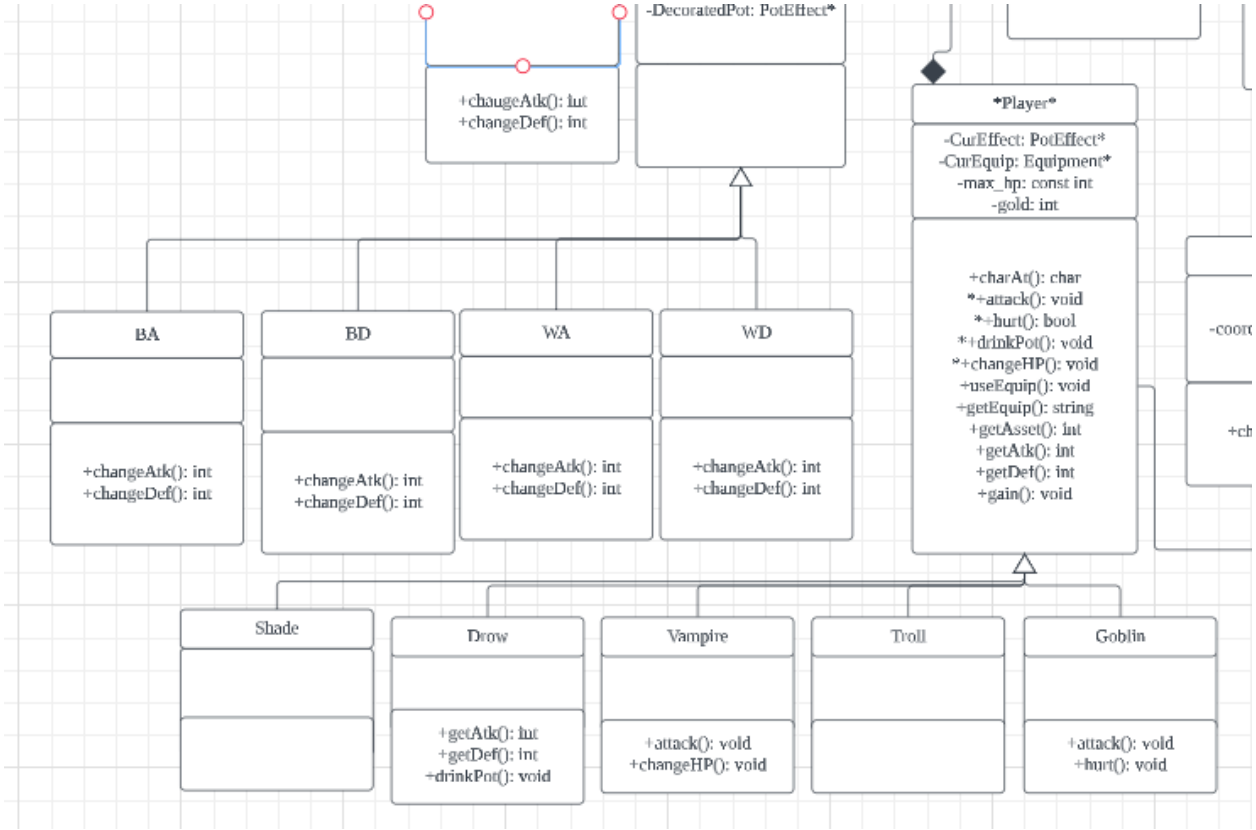


Top left:

Mingyi Su, Jeannie Shi, Donghui Yu

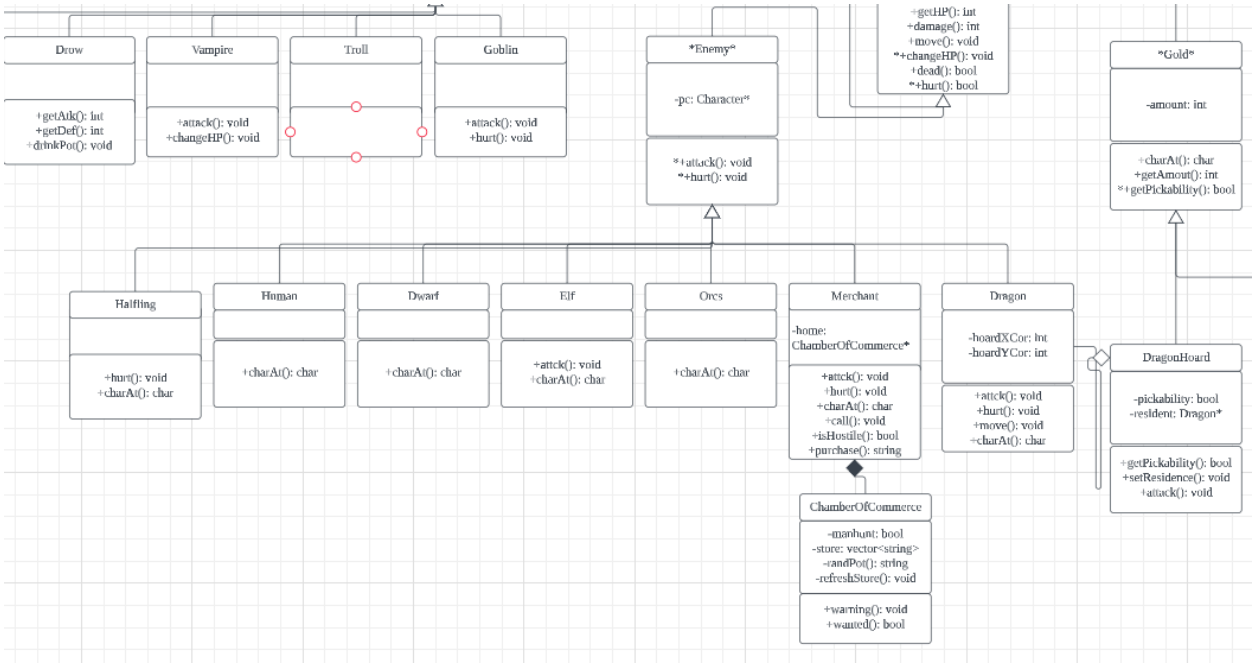


Bottom Left:

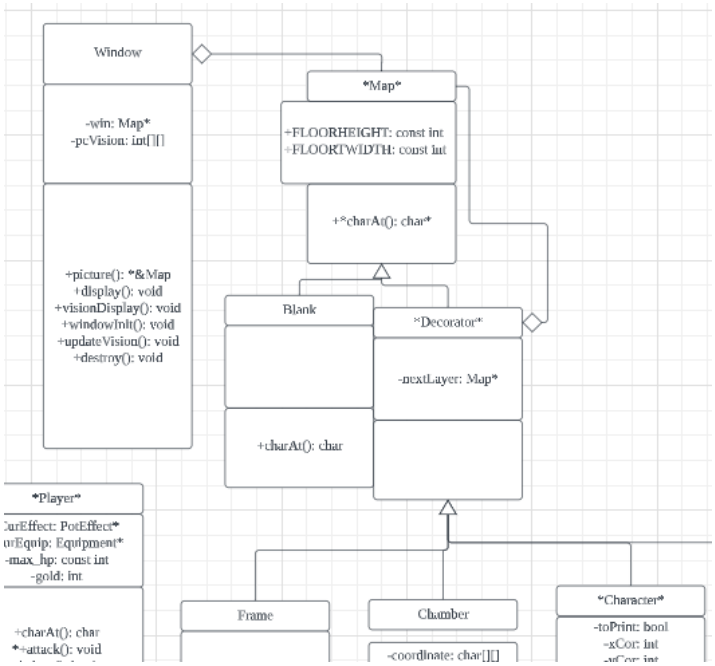


Mingyi Su, Jeannie Shi, Donghui Yu

Bottom:

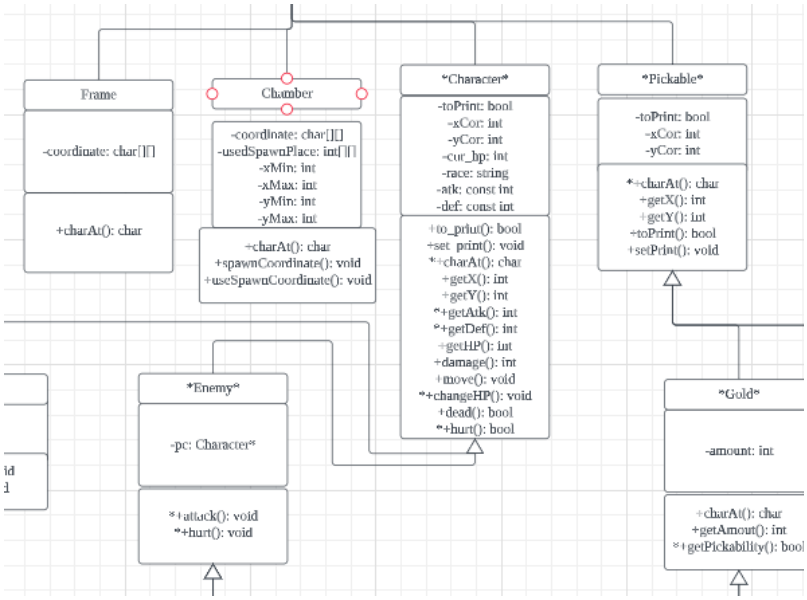


Top:

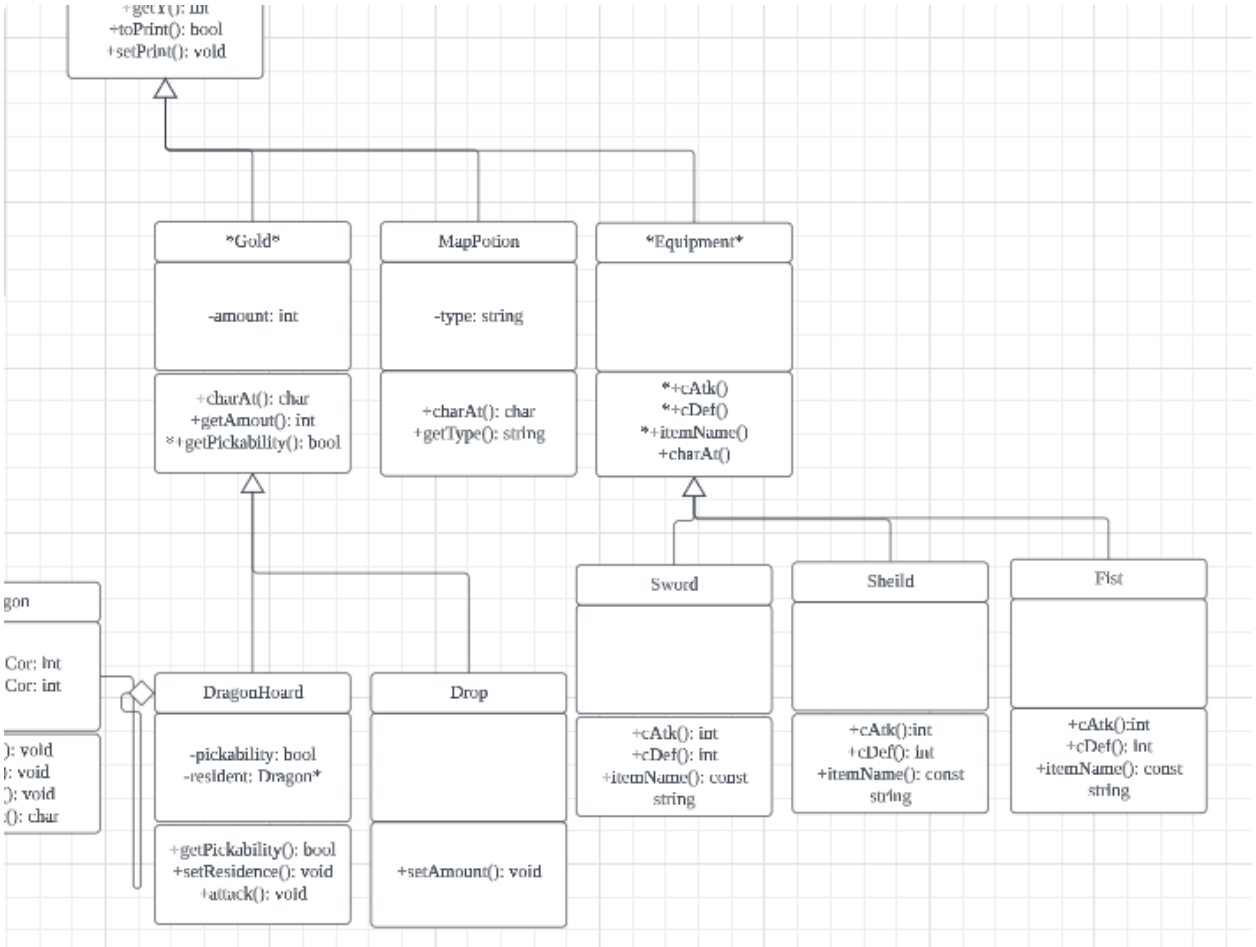


Center:

Mingyi Su, Jeannie Shi, Donghui Yu



Right:



To view a more clear version of this UML, please go to the whole pdf UML attached.

Mingyi Su, Jeannie Shi, Donghui Yu

Design:

Module 1: Generating

This module is responsible for generating the game's base elements, including the blank layer, frame layer, chamber layers, player, enemies, gold, and potions. We implement the decorator pattern for generating these layers, ensuring each layer builds upon the previous one to create a coherent game world. For example, the player and enemies are placed on the chamber layers to ensure they appear within the game environment. To facilitate object management, we use separate vectors for different types of objects, such as chambers, enemies, potions, and gold. Some objects, like dragon hoards, have specific spawning behaviors, like spawning a dragon within a 1-tile radius.

Module 2: Moving

In this module, we handle the movement of enemies and the player. Enemies move one tile in a random direction if the player is not in their vicinity. We achieve this by checking for available spots around the enemy and randomly selecting one to move towards. However, dragons guarding dragon hoards do not move. For the player, we enable movement based on user input, ensuring they can move to empty spots and pick up treasures. We prevent movement into blocked areas or encounters with live enemies or potions.

Module 3: Displaying

This module controls the visibility of objects in the game window. Many classes have a "printability" function that determines whether an object should be displayed. For instance, when an enemy is slain or a potion is picked up, their printability is set to false, hiding them from the window. This display state also affects the game's detection mechanism. The Window class plays a central role, managing the display of layers based on their printability.

Module 4: Combating

In this module, we handle combat actions between the player and enemies. Both Enemy and Player have attack and hurt function. The attacker’s attack function calculates the damage and passes it to the defender’s hurt function, which is responsible for reducing the hp. During the game loop, the player can perform actions such as moving, picking up potions, or attacking nearby enemies. The algorithm checks for the availability of the desired object before executing the corresponding action. For example, attacking an enemy involves looking for the pointer to the enemy at that position and then passing its pointer to the player's attack function. After the player's turn, enemies take their actions, either moving or attacking the player. The player's hurt function will be called only when the player is within one radius block of that enemy.

Module 5: Picking

This module governs the player's ability to pick up potions and gold in the game. Map potions and gold are subclasses of the Pickable class, forming a decorator pattern. When the player interacts with pickables, they become unseeable but still exist in the game world. Additionally, potions have their own decorator pattern. A MapPotion object spawns on the map, providing information about the type of potion. When picked up, the corresponding real potion from the real potion class is activated, boosting or weakening the player's status. The player holds a pointer to this real potion class for potion usage.

Our core design technique is using decorator patterns. Every object that appears on the map employs decorator patterns. Using these patterns helps us easily implement the generating, moving, detecting functions. Thus, by properly using the convenient functions provided by decorator patterns, we implement our project.

Challenge:
Communication is a big challenge in our collaboration. Firstly, we all have different coding and naming styles, making it harder to read and understand each other's code. Even though we work on different parts, we often unintentionally create similar functions. Also, we frequently end up editing the same file at the same time, especially the main file, and then we have to spend a lot of time combining our changes.

Resilience to Change:

The decorator pattern serves as the backbone of our object management, allowing us to hold and layer various game elements effortlessly. Specific classes, such as Enemy and Pickables, act as the foundation for grouping related objects. To introduce new objects under these categories, all that is required is to create a subclass of the corresponding base class. For instance, the DLC expansion introduces Equipment as a Pickable.

Additionally, we use the general Decorator pattern to automatically let new objects have essential functionalities. The charAt() function, responsible for displaying objects on the game window, is inherited by any object added to the Decorator pattern. This ensures new objects can be effortlessly showcased within the game.

Furthermore, our design approach treats each object as a layer on the map. This modular organization simplifies the addition, removal, and modification of objects. New entities can be seamlessly incorporated into the existing layers, promoting extensibility and maintainability.

Answer to question:

Mingyi Su, Jeannie Shi, Donghui Yu

Question: How could you design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional races

Answer: Each race is a derived class(subclass) of the parent class, the Character; the Character has two direct children classes, which are Player and Enemy. Adding a player race is simply adding a subclass under the Player which contains the ability and stats(def, atk and hp) for that race. Adding an enemy race is similar.

The Character class serves as the base (parent) class for all races, with two direct child classes, namely Player and Enemy. Each specific race is implemented as a subclass of either Player or Enemy, inheriting their attributes and functionalities. To add a new player race, you can create a new subclass under the Player class and define its unique abilities(implemented as overridden functions) and stats such as defense, attack, and hp. Adding a new enemy race follows a similar process.

Question: How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

Answer:
When creating a PC object, the player has to choose their desired race from the five available options. Consequently, only one player character object (race as well) will be created based on the player's choice. On the other hand, generating enemies follows a different approach. In the main() function, enemies are randomly allocated to a random chamber using a random function. (The class Chamber also provides a helper function that assists to provide an empty spot in it). Once the position allocation is done, their respective constructors are called to generate and initialize the enemy objects accordingly. The only similarity between spawning a player and an enemy is that they both need an random chamber number first, by random function. This way, the game ensures that the player character and enemy characters are created through distinct processes, catering to the player's choice while introducing unpredictability and diversity in enemy encounters.

Question: How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

Answer: All the enemy races have member functions for combat, including attack(), hurt(), and move(), and a pointer to the player, inherited from its parent class. During a combat, enemy's attack() is responsible for calculating the damage produced based on its atk and player's def, as well as some additional impacts or abilities between two specific races (For example, the effect

that Orcs does 50% higher damage to Goblins is calculated by Orcs's attack function), and then it calls the player's hurt() and pass the damage to that function. Player's hurt() will take an int input representing the damage, and will reduce the player's hp field. Enemy's hurt() has similar functionality, and the difference is that it will check if this enemy object is dead. If the enemy object is dead, it will increase the player's amount of gold, either 1 or 2.

Question: The Decorator and Strategy patterns are possible candidates to model the effects of potions, so that we do not need to explicitly track which potions the player character has consumed on any particular floor. In your opinion, which pattern would work better? Explain in detail, by discussing the advantages/disadvantages of the two patterns.

Answer: In my opinion, the Decorator pattern works better. The advantage of this pattern is the ability to create individual classes for each potion, making it easier to manage potion effects. Additionally, potions can be dynamically stacked, allowing multiple effects to be applied to the player simultaneously. Though recursive functions might be needed to calculate attack and defense each time, the number of consumed potions is strictly limited to ten, resulting in constant time complexity O(1) for calculating attack and defense. As for its disadvantage, implementing six classes may increase complexity, potentially leading to difficulties in maintenance and management. Nevertheless, with proper design, it remains manageable. On the other hand, the Strategy pattern offers more flexibility in switching between different potion effects during gameplay, but to keep track of previous changes, additional methods are required, leading to increased complexity. In conclusion, the Decorator pattern is superior due to its ability to combine potion effects effectively, providing a robust solution for managing various potion types.

Question: How could you generate items so that the generation of Treasure and Potions reuses as much code as possible? That is, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code?

Answer: Both of these two classes are subclasses of the parent class Pickable. The pickability function is implemented in the parent class, Pickable, through the setPrint(), toPrint(), and charAt() functions. Their unique features are implemented in the subclasses, such as when a player picks up a gold, the player can gain some gold.

Extra Credit Feature:

Mingyi Su, Jeannie Shi, Donghui Yu

We cooperate together to make a DLC called “A More Real Dungeon”. In this DLC, you can buy potions from merchants, have limited vision and use equipment such as swords, shields and your own fists. To activate this DLC, please type “./cc3k dlc” in the terminal.

Final Questions:

Q: What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

A: (1) Communicate with teammates about coding styles first. There is a high possibility of bugs appearing if teammates use a different coding style with you. (2) Using tools, especially github. Using github greatly increases our working efficiency. Every person can work on their parts and integrating all code together is also convenient. (3) Communicate. We usually communicate via discord or meeting each other in person in DP library. The good communication between us helps us achieve the completion and dlc completion of the final project.

Q: What would you have done differently if you had the chance to start over?
A:
(1)Donghui Yu: I would start the final documentation earlier instead of making the dlc.
ヘ(;´Д`ヘ)
(2) Mingyi Su: I am motivated to delve into more challenging and intricate OOP principles to create funny, entertaining and engaging features.
(3) Jeannie Shi: I would like to spend less time on deciding which design pattern to use for the potion part. I have spent a lot of time deciding between the decorator pattern and the strategy pattern.

Conclusion:

With solid design strategies, effective time management, and good communication, 'CC3K' and its DLC is completed on time. The strategic implementation of the decorator pattern ensures smooth basic gaming function and thus allows us to build more features to this basis. Throughout this project, we all learned how to effectively communicate with each other. We learned how to use tools such as github and discord. We also learned teamwork organization skills.