CS 246 Final Project

Xinhao Xu, Zian Wang, Joyce Dai

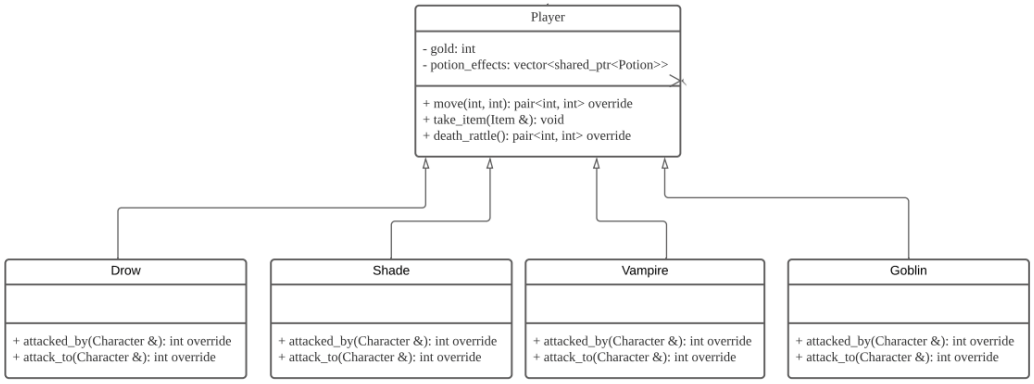
- **Breakdown of Project Timeline & Responsibilities**

Task	Responsible Team Member	DDL
Plan of Attack	All	11.30
General main structure	All	11.31
General Floor Rendering	Xinhao Xu	12.1
Class Chamber	Xinhao Xu	12.2
Class Cell	Xinhao Xu	12.3
Class Character	Zian Wang	12.4
Class Enemy	Zian Wang	12.5
Class Player	Zian Wang	12.6
Class Item	Joyce Dai	12.7
Class Gold	Joyce Dai	12.8
Class Potion	Joyce Dai	12.9
Specific Floor rendering with all components	Zian Wang	12.10
Text Display	Joyce Dai	12.10
Finalize main.cc	All	12.11
design.pdf	All	12.12
demo.pdf	All	12.13
uml-final.pdf	All	12.14

- # Project Specification Questions

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?

For each race to be easily generated, we can use inheritance to create an abstract base character class, then create an abstract player class that inherits from the character class, then each race will be a concrete class that inherits from the player class. Each time when a floor is generated, a player pointer will be passed into its constructor, so that the status of the player character can be remembered for all 5 floors. With this strategy, it will be fairly easy to add additional races. It can be simply done by creating another class that inherits from the player class with its own race features and attributes.

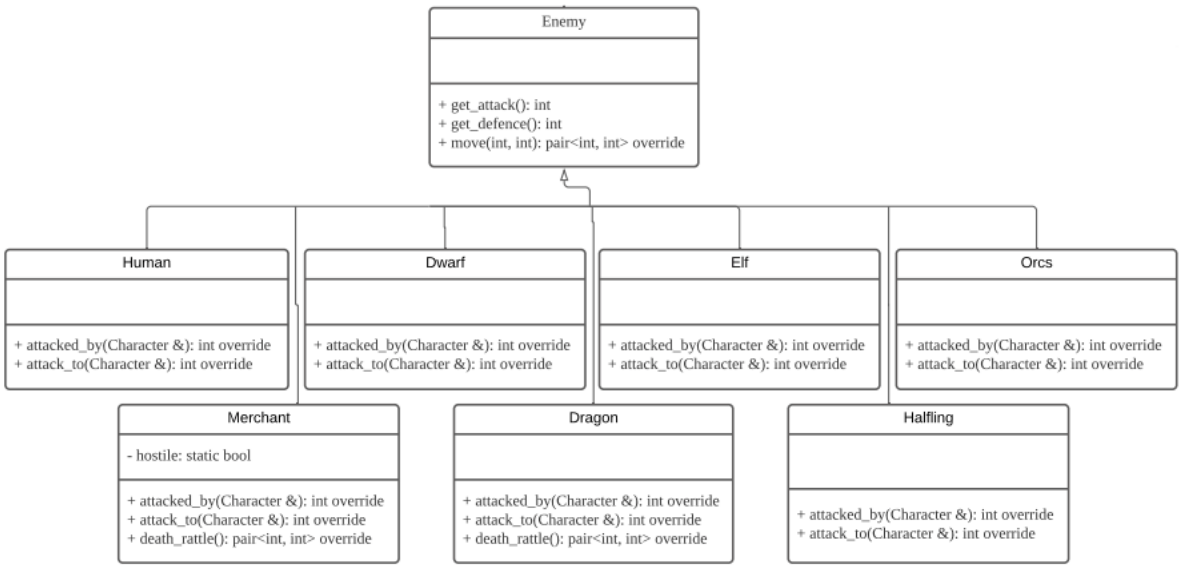


Question. How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

To generate different enemies, we can create an abstract enemy class that inherits from the character class, then for each type of enemies, they can be implemented as a concrete class that inherits from the enemy class. The way that we generate the enemies is the same as how we generate the player character since both strategies use inheritance.

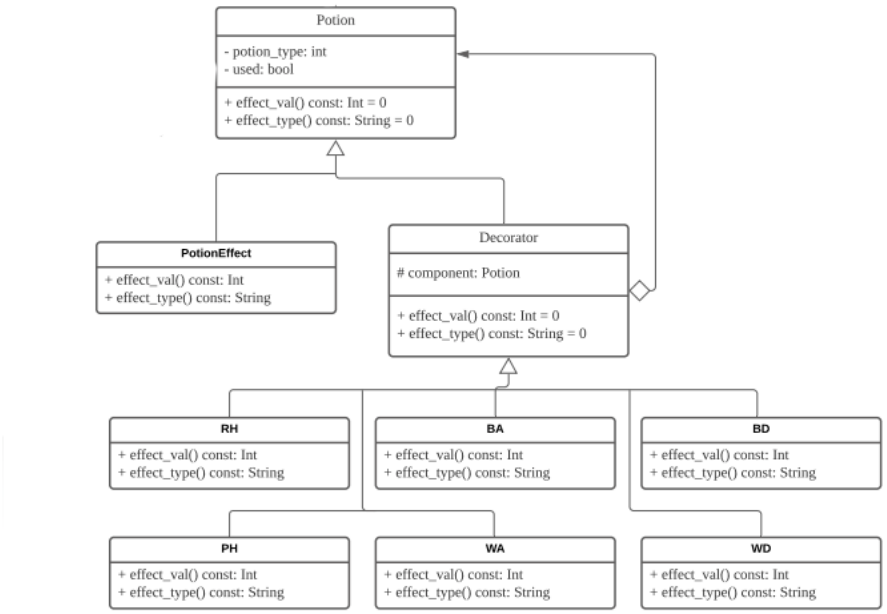
Question. How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

To implement the various abilities for the enemy character, we can store each of their particular features and attributes in their concrete classes. Then, we can implement a Visitor pattern. When a player is attacked by an enemy, the enemy's pointer will be dereferenced and passed to the character that is being attacked, to specify its effects. We are going to use the same technique to implement the various abilities for the player character races. This is because the player will also attack or be attacked by the enemies, just as how the enemies will attack or be attacked by the player. Therefore, both the enemy and the player characters' abilities can be implemented by a Visitor pattern.



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.

The advantages of the Decorator pattern are that it enhances the extensibility of an object by allowing the program to add additional features to an object at run-time rather than to the class, and it increases the flexibility of the program as any functionalities and features can be withdrawn. This is particularly useful to model the different effects of the potions, and to remove their effects when the player character reaches the next floor. Its behavior also allows us to modify any effects of the potions without having to change an entire class, so it makes coding and debugging easier. The disadvantage of the Decorator pattern is that it may result in many repetitive code in different classes, making the code harder to maintain. The advantage of the Strategy pattern is that it allows the program to interchange between different algorithms at run-time, allowing the algorithm to vary independently of the client. Each potion's different effect can be implemented as different algorithms, and the program can select these different algorithms when a player uses the potion. However, the disadvantage of the Strategy pattern is that how each algorithm is selected has to be specified, and the algorithm is only called under certain circumstances. Since the Decorator pattern allows us to freely add features and remove them, while the Strategy pattern may result in having to write many different algorithms, we believe that the Decorator pattern is a better 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.



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?

We can use the Template Method pattern to generate Treasure and Potions to avoid duplicate code. First, we can create an abstract item class with methods to generate the item's information and conditions for its spawning. Then, create concrete classes for each type of Treasure and Potions, and override the above methods with specific setups, such as when a merchant or a dragon is killed, a Treasure will be generated or can be picked up respectively.