CC3K Final Project Report

(Team Member: Haoyang Sun, Zijian Zheng, Minghao Li)

Introduction

The cc3k game is a type of RPG strategy games where player can control a character to start an adventure in a multi-layer world. When exploring, player characters will encounter enemies, with whom they can engage a battle, treasures which they can pick up and potions which they can consume to obtain bonus effect. There are different types of player characters as well as enemies, whose skills and special abilities are completely different, enabling various strategies to win the game. The whole project is written in C++ with OOP approach. The project structure can be found in the UML diagram below. Several design patterns are applied to increase the readability and maintainability of the source code.

Overview

The character class is used to implement character objects which include players and enemies. Hence, player class and enemy class both inherited from character class. Since the reaction of the program can be seen as a response from a player's action, we implemented the observer pattern with the subject to be player and enemies along with potions to be observers. In another word, player class inherited from subject class and enemy and potion all inherited from observer class. Therefore, whenever a command is issued to player, player conduct actions and notify all the other objects on the floor to do things they are supposed to do. This is the core part of how the game moving forward.

As for the damage calculation in the battle, we implemented a damage visitor which follows the visitor pattern. Since the damage in battle is varied based on the type of the battle, a damage visitor allows simple function calls to calculate the battle damage.

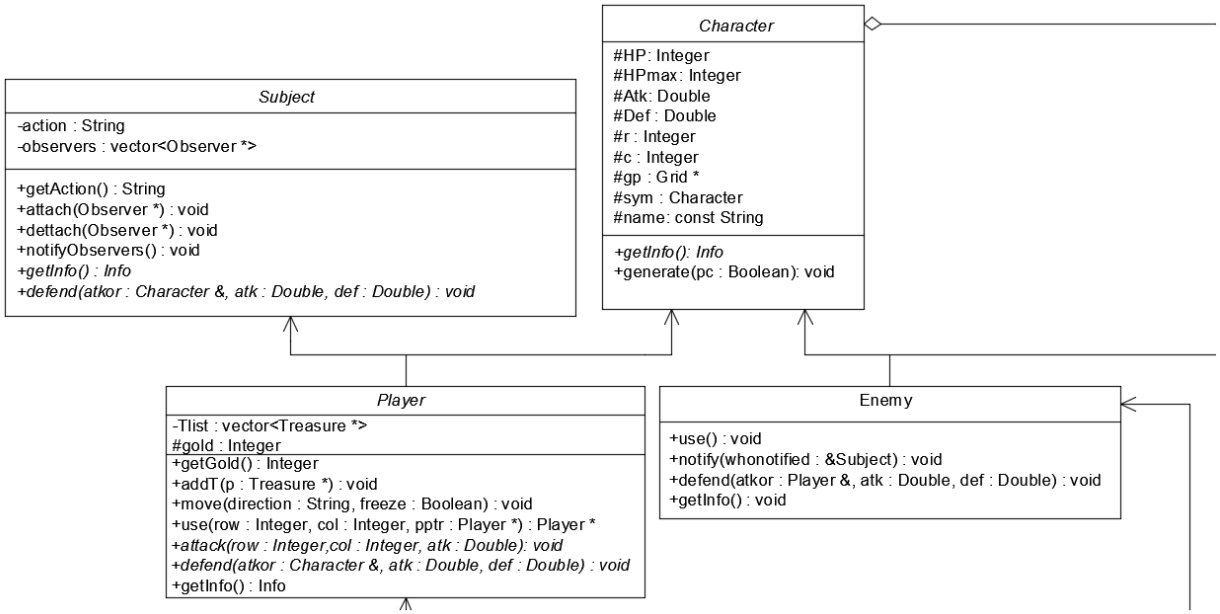
In terms of the potion effects, we implemented a decorator pattern to achieve easy management of the potion effects. Following this idea, a potion decorator class is created as a subclass of Player because only players can use potions on the floor. When the player encounters a potion, a potion decorator is applied to "wrap" the player and gives out decorated data in the future games.

Regarding treasures, a serrated class name Treasure is implemented to store information about the location, value and type of the treasure. Normal treasures are sitting on the floor waiting to be notified of player actions. In the case of dragon treasure, a special notify is implemented to check the condition of the corresponding dragon enemy. When a treasure is picked up, it will be removed from the special treasure list in the player

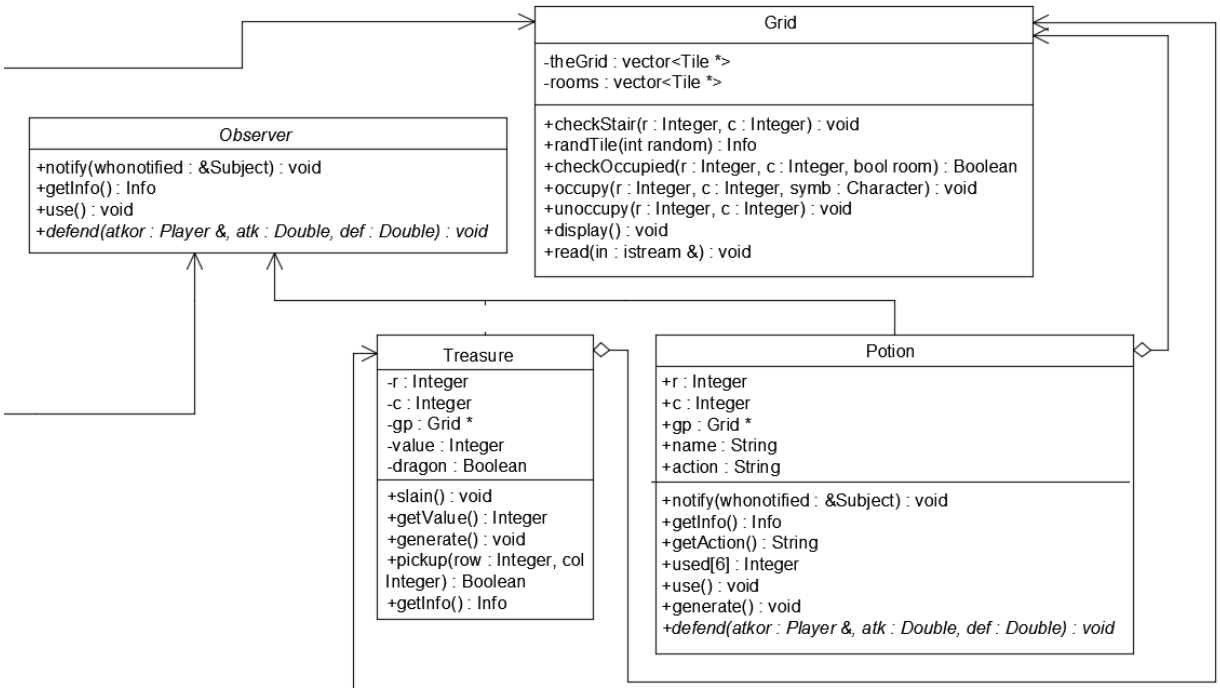
and player’s gold amount will be increased.

Finally, Grid and Tile class is created for display purpose.

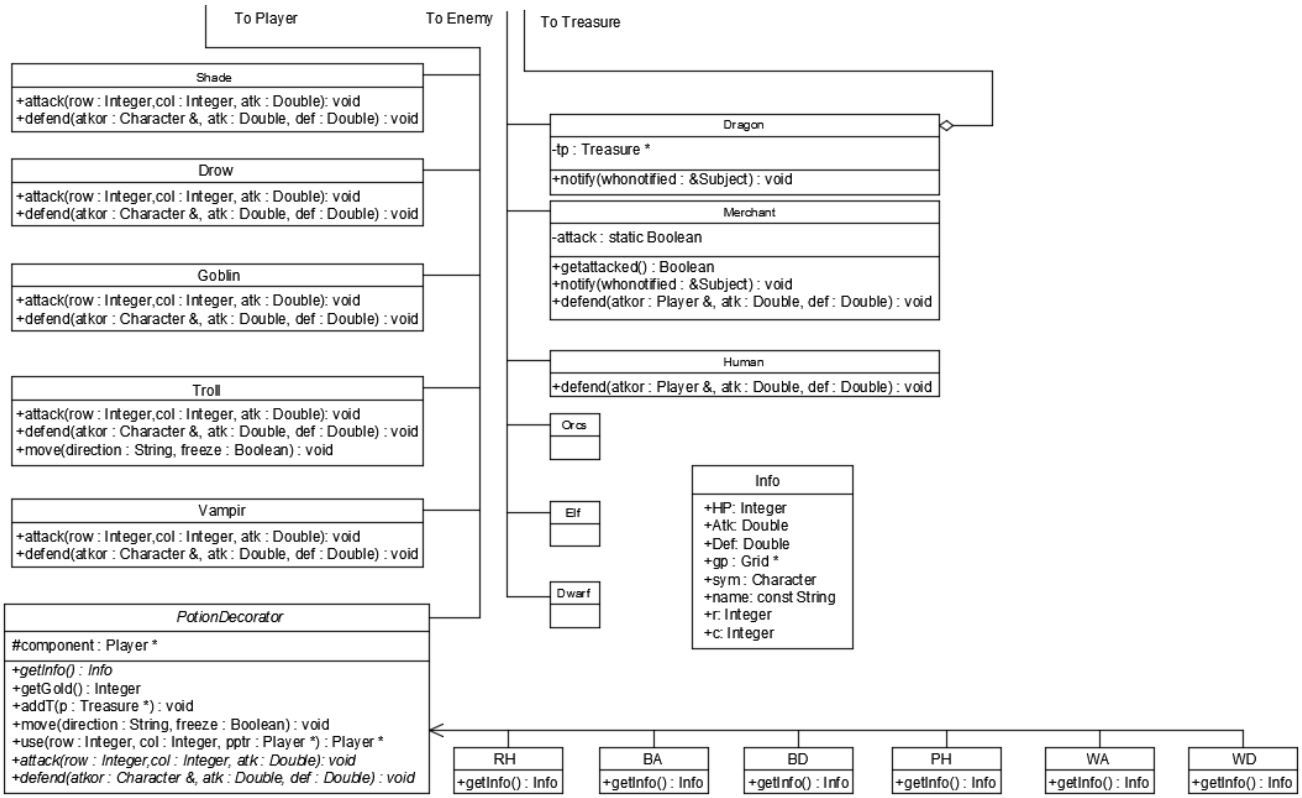
Updated UML (Since the whole UML is too big, for display purpose, it is divided into four sections)



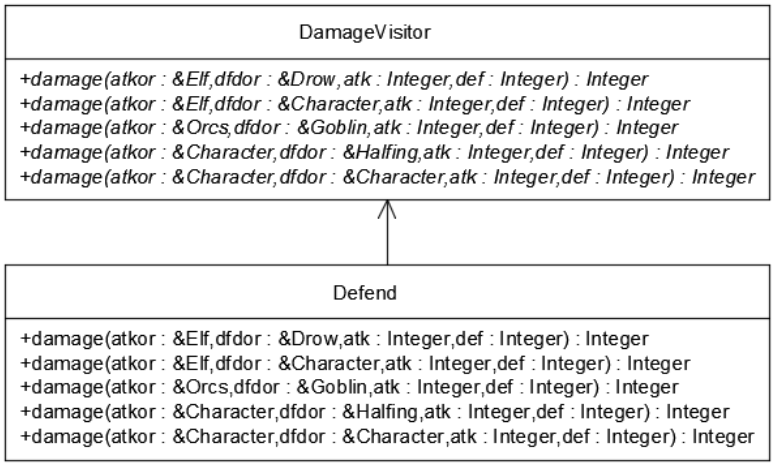
Sec1. Top Left



Sec2. Top Right



Sec3. Bottom1



Sec4. Bottom2

Design

- *Observer Pattern*

We set Player class to be the subject (i.e. inherited from Subject class), and set Enemy class and Potion class to be observers (i.e. inherited from Observer class). When the game is initializing, in the main program, by reading in the command which specifies the player character type, a player character is spawned. Then, all the other observer objects are spawned randomly in the floor, attaching themselves to the spawned player object through attach() function. When the game starts, when a player action is conducted, such as move(), attacked() or use() functions are called, the player object will call notifyobservers() function which is defined in the subject class to call the notify() function in all the observers objects. Since every concrete observer objects have different notified response and we don't want users to create a concrete observer object, the nofity() function is pure virtual. Meanwhile, when a battle is encountered, the player will search for the object that it's battling with in its observer list. This requires observers to be "aware of" the functions called in the battle, but it doesn't need to know its implementation. Hence, declaring these methods as pure virtual methods is necessary. Whenever an enemy is killed or potion is used, it will call the players detach() method its notify() method, removing itself from the list and delete itself.

-*Potion Decorator*

Following the decorator pattern, we implemented a Potion Decorator class inheriting from Player class because a potion is essentially a player with a component pointer pointing to the previous layer. In fact, we don't care about the HP, Atk or Def in the potion decorated player. We only care about how to recursively manipulate the HP, Atk and Def retrieved from the interface method getInfo(). Hence, we implement the pure virtual method getInfo() inherited from Potion Decorator class which is from Player class in every concrete Potion class. Since the player who is "wrapped" in a potion decorator is still a player, it needs to override all the battling methods since its HP, Atk and Def retrieved from getInfo() has been changed. However, every player has different abilities which mean different battling methods, so we can implement them the same, To solve it, when a battling method is called, we recursively call the same battling method in the previous layer until it reaches the base class which has an implementation of its own battling method. By doing this, base HP is changed but the HP that retrieved from getInfo() is still decorated based on its base HP. So as Atk and Def. When entering the next level, we will reconstruct a new player character with its decorated HP as the new base HP, Atk and Def can be determined from its type, and all the other information can retrieve from getInfo().

-*Treasure and Dragon Treasure*

For normal treasure, we store all the treasure on the map in a vector inside the player

class. They are generated after the generation of enemies and randomly spread on the map. While there is a special case for the dragon treasure. To implement the dragon treasure, we first randomly spawn the dragon. We put in inside the player's observation list. Then, we generate the dragon treasure randomly within one block around the dragon. And we do it once for each dragon. Finally, we store the pointer to the dragon treasure inside the dragon class. By doing this, when the player moves to within one block around the dragon or the treasure, the dragon will be notified through the observation list or by the dragon treasure through the treasure vector. If any one of them notify the dragon, the dragon will deal damage to the player, which calls the defend() method in the Player class. What's more, we store a Boolean value in the treasure class so that only when the dragon is killed, the player can pick up the dragon treasure after judging from the Boolean value. This design makes sure that each dragon treasure will be matching to only one dragon. And both dragon and dragon treasure will be correctly notified and dealing a correct amount of damage.

-Damage Visitor Pattern

Since different types of enemy and character skills will generate different damage depending on the type of attacker and defender, if we implement them using if and else cases, there would be a lot of typing and it would be really hard to add in new characters and abilities. Therefore, we used the visitor pattern to let the compiler to do the heavy work. In the damage method of the concrete DamageVisitor class Defend, we implement different damage method given attacker type, defender type, attacker's attack and defender's defend. Thus, when the damage method is called, and all the parameter are passed in, the compiler will automatically choose to execute the "most fitted" damage method implementation to calculate the damage. By using visitor pattern, we can reduce the coding needs to be done when implementing the battling methods in the players and enemies. Besides, it's a really good practice to keep the code maintainable and convenient for adding new characters with different abilities. All the changes need to do is to implement a new damage method in the Defend class.

Resilience to Change

Our source code allows easy maintenance to every section of the implementation. For new player character type development, you only need to implement a new Concrete Player class inheriting from Player class, and implement your own battle function if need. If the ability involves changing the damage calculation, simply add a new damage() method in the Defend class which is a concrete subclass of DamageVisitor and specify the attacker and defender as you need. If a new enemy need to be created, similar to the player character, we only need to implement its own concrete subclass inheriting from Enemy class and change the DamageVisitor if needed. If a new potion type is needed, simply create a new concrete subclass of PotionDecorator class and

implement the effect of the potion in getInfo() method. If a new treasure type is needed, you only need to create a new treasure objects with the field value rewritten to the new value.

Answers to Questions in Project Specification

For generating each race, we use the inherited structure on the player including all the races. The base class is the Player and all the different races will be the subclasses. If you want to generate a race, you just simply construct a subclass object which inherits all the important field (like HP, Atk, Def) from its base class. By this way, if you want to add a new race, you only need to add a new subclass inherit from the base class and implement any necessary methods considering different ability. Comparing to making all the races as a different class or using a lot of if-then statement to build up a player, this way can make it easier to add new races and easier to implement races without changing a lot of codes outside this class.

Generating enemies is similar but different from generating a player. Because every round the player will move first and then the enemies will move randomly or attack player depending on the position of them. So, the player is always on the offensive. Then we decided to use an observation pattern to implement enemies. We put all the enemies on the map as observers to the subject player. Each time the player moves or attacks or even uses potions, we firstly finish what the player will do, then we notify all the enemies under player's observation list. And the notify function will decide whether the enemies are moving or attacking judging by their positions. Although we still use the inherited structure to generate enemies (since they both have different types as subclasses), this makes it different from generating a player.

The abilities of player are decided to make an effect when they are needed. We implement all the abilities of different races by implementing each of them within their own class method (they each have their own attack, defend, and use potions methods). So, the system will automatically determine the race of the player and then call the specific methods which include their abilities. However, we use a different strategy on enemies. Since all the enemies' abilities make effects during combating, we simply design a visitor pattern to calculate the damage. So that all the enemies can share the same defend function. Basing on that player' s abilities make an effect at a different time from enemies', the techniques we use on player and enemies are totally different.

Using Decorator pattern and Strategy pattern will make the implementation of the potion class totally different, and they each have their own characteristics. Using a Decorator pattern will make it easier to trace each potion the player uses and the order the player uses them. However, it will be difficult to distinguish the effect between HP potions and other potions. Because the effects of HP potions are permanent. Entering

the next floor won't make an effect on the player's HP. While using Strategy pattern will directly change the fields of player. Then it will be much easier to access the fields of the player after taking a bunch of potions. This saves a lot of time during running and makes it more efficient. However, using Strategy pattern will cause a problem that when we get to the next floor, we won't able to know what potions did the player use on the previous floor. So, it will be super difficult to remove the effect of the Atk and Def potions the player used on the previous floor. To solve this problem, we chose the Decorator pattern. The actual fields of a player will not be changed after using a potion. Instead, we use Decorator pattern to build up the effects of HP, Atk and Def potions. The effects will happen when we use getInfo() to get the value of player's HP, Atk and Def. When we get to the next floor, we just get the HP of the player and erase all the decorator, which includes the final HP from the previous floor and the unchanged Atk and Def. Using the Decorator patterns really helps to develop the program as well as maintaining encapsulation. This is still the same as what we first thought about, and is proven after we implement it.

Generating potions and treasures are quite similar. They both need to be generated at an available position under similar amount and probability. So, we decided to build up a function just for a random generation in the class Grid, which includes the whole map and knows all the available positions. It will return and occupy a random available position, which makes it not available. Then we simply just call this function to occupy a specific position and give this position to the potions of treasures we generate. Making a unified random generating function is super useful because we don't have to implement different random generators when we are creating different items. This saves a lot of our time and makes the code less duplicated. This is quite different from what we first thought about, but after implementing it, we found this way more useful and helpful.

Answer to Final Questions

We worked at a super cooperating team. After building such a huge program that we had never done before, we realize the importance of working as a team. If we were going to do this alone, it will be impossible to finish the assignment on time. It is because we cooperate, we can finish this program. Each time when we met a problem on design, on debugging or on testing, there will always be one of us providing a new idea or a new solution to solve this problem. And every time we felt frustrated on the error the program produced, we just encourage each other to continue to work on it. This is how a team work. We all use what each of us is good at, and make the best contribution to the team as we can. Efficiency is maximized when we were working as a team. Also, we considerably improve our communication skills when we were working as a team. Now we can fluently express our idea to each other without omitting or making mistakes on understating and illustrating. And in the future, we believe we

can handle any challenges while working as a team.

At first, we underestimate the time we need to finish this assignment, so we started quite late. Luckily, it wasn't too late when we find out we need more time. So, we raised our speed and try to allocate more time working on it every day. Also, we met a problem during the middle of the assignment, which is unifying languages. Since we are all responsible for different parts of the program, when we tried to combine all the parts together, we found out that we are using the different function of argument names in some places. So, we wasted a lot of time to change it and unify the language use. If we have a chance to start over, the first thing we will do is to start early. We will allocate more time every day to meet and do the assignment. And we will discuss the whole plan in details and unify all the language we are going to use. If anything is going to change on each person's part, make sure others are notified. When we do this, I believe we will finish the program early without making the awkward situation of unifying languages. And we will be able to have more time on developing some extra credits contents (like some Downloadable Contents).

Conclusion

In conclusion, we learned a lot from doing this project. Not only we learned how to combine and perform all the knowledge we learned in class to actually implement a complete program, but also, we learned how to work closely as a team and how to communicate with each other in a language that everyone knows. Since time is limited, the final program we developed might not be perfect. We can still spend some more time to make it more efficient, make better. But we are still satisfied with all the effort we made and we won't be regretting it. If we are going to do something similar in the future, or in our career as software developers, we already have preparation about it. We are ready for accepting any challenge in the future.