Final design document – CC3K

Team Members:
Cui, Lin
Yan, Xinyi
Zhang, Xinyin

Overview

1. Main Function

In this game of CC3k (ChamberCrawler3000) we designed, the main function is used to store the information of the selected role of player character and the current floor which is destroyed and generated each time the player goes to the next level. The major part of our main function is a while loop, in which each iteration goes one step further in the game, corresponding to the input command, and display the game again to refresh the illustration.

2. Floor

A class *Floor*, which will be stored in main.cc, include the fields *display* (a vector of strings to make it easier for main function to display the status of game), vectors of the pointers to *Enemy*, *Potion*, *Treasure* and *Chamber* in this floor, a pointer to the *Player* in the current game and the name of file containing the floor layout passed by the command line. The class *Floor*, while not passed a filename, can also generate *Player*, *Enemy*, *Potion*, *Treasure*, Stair randomly by itself with a corresponding method each. Through the *Player *p*, the player's status is used to judge that whether the game is lost or won, or a new floor should be generated.

3. Object

The class *Character* and *Item*, included in a vector in the class *Floor*, are both inherited from the class *Object*, with positions and type (*Player*, *Enemy*, *Treasure* or *Potion*) illustrated. *Object* is a superclass for *Character* and *Item* to provide common fields and methods to reduce duplicate code.

4. Character

Character is a superclass for *Player* and *Enemy* and records character's Attack, Defense and HP and has a pointer to the *Floor* the character is on. The character's status is monitored by methods.

5. Player

This class represents a type of character, the player character. *Player* is a subclass of *Character* and records the amount of treasure the player has picked up, the maximum HP, the action it takes, whether the merchants will attack the player and whether each type of potion has been used for at least once. Methods are built for each of its actions: attack, use potion (for each type of potions, Visitor Design Pattern used). It has 5 subclass and each is a role of player character.

6. Enemy

This class is a superclass for all 7 types of enemies and checks if the enemy it represents is hostile and movable. Similar to *Player*, the actions of *Enemy* are also realized by methods.

Moreover, the subclass *Dragon* has a pointer to a *Treasure* whose position is within a block of itself.

7. Item

Item is a super class for *Treasure* and *Potion*. It monitors the status of the item (whether the treasure can be picked up and whether the type of potion is used before).

8. Treasure

Since *Treasure* is a pile of gold then the value of the treasure is illustrated. If it is picked up the method of *Player* will be called to update the record in *Player*.

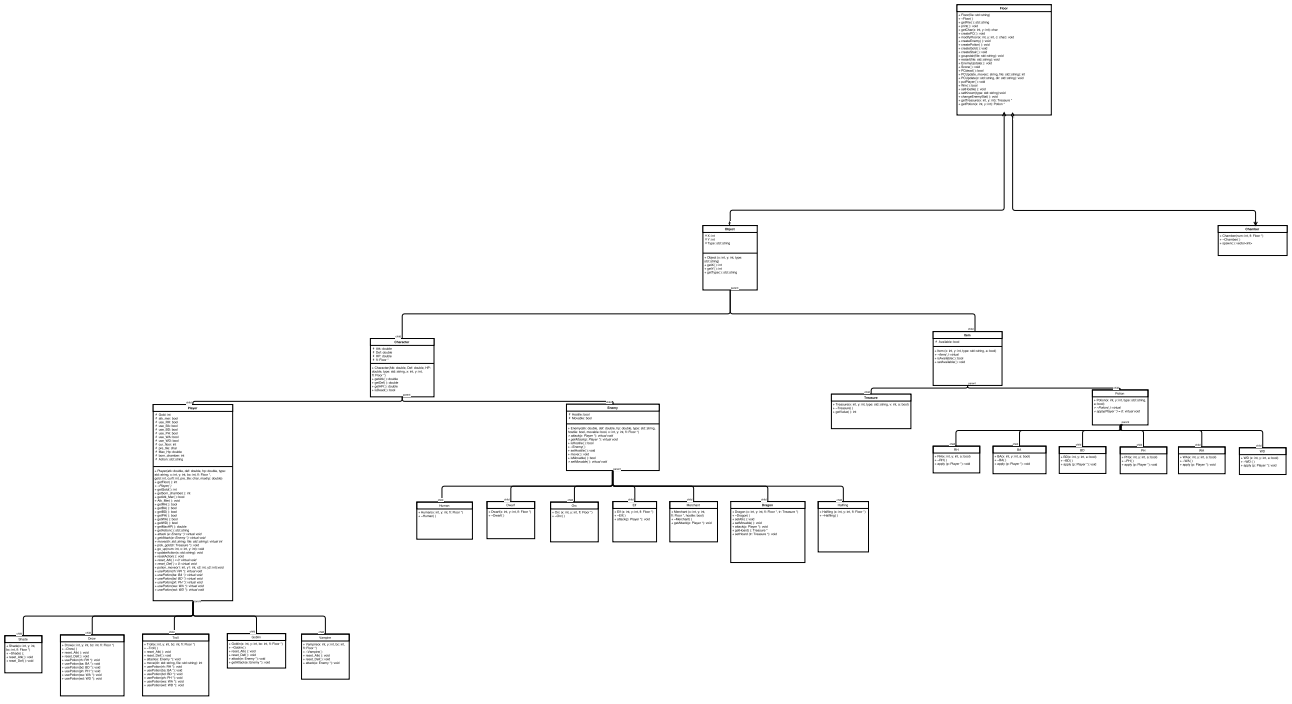
9. Potion

There are 6 types of *Potion* and each is designed to a subclass. Since each potion gives a different effect on *Player* then *Potion* is made to be an abstract class. Each type of *Potion* has its own method to be applied to *Player* through a pointer to it.

10. Chamber

There are 5 chambers in each *Floor* so every *Chamber* is given a number.

Updated UML



Design Techniques:

System Modelling(UML)

CC3K is a large program which includes many different classes. At first, we had no idea of how to start this projects. By designing a uml in advance, we gained a basic knowledge of how classes would interact with each other and what methods we were supposed to implement. Designing a UML helped us get started with our project.

Standard Template Library (vector)

In Floor class, we want to store all enemies, potions, treasures, chambers and the display of the current floor(stored as strings).

At the beginning, we tried to use arrays of pointer to store these objects. However, we found it was not convenient if we changed the number of objects on the floor.

We finally handled this memory management via vectors. In this way, we can add as many objects as we want to Floor class because the size of vector will be handled automatically.

Visitor Design Pattern (Potion Effect)

We first considered using Decorator Pattern for potion effect, but we found that decorator pattern would result in lots of small objects. Besides, if we use decorator pattern, player would be wrapped in potion class, which would make it difficult for Enemy to attack player.

Because the effects of potions depend on both potions and player, we used visitor design pattern to model the effects of potions.

In Potion class, we implemented a virtual method called apply(Player *). In all subclass of Potion(RH, BA, etc), we override this method as :

void apply(Player *p) { p.usePotion(this); }

The Player class contains all "usePotion" methods (e.g. usePotion(RH *)). Since Drow has special ability of using potions, we override those methods in Drow class.

When player goes upstairs, we will use reset_Atk() and reset_Def () to reset their Attack and Defence.

Inheritance

This project contains lots of classes. Many classes have common fields and methods. In order to remove duplicate code, we have a Object class which is a base class for all characters and items. The Object class contains fields which store x-coordinates, y-coordinates and types of objects. It also contains methods： getX(), getY() and getType().

Same idea applied to Player class and Enemy class. For example, Shade class(Drow class, etc) is inherited from Player class. Player class contains the majority of fields and methods needed for each player race. In the subclass of Player, we only need to override particular methods. Inheritance helps us remove duplicate code.

Resilience to Change

1. The basic strategy is to use modularization to divide the large program into several modules and store each module in a separate file. In this case, we store each class in a separate file. If we find changes need to be made, only the file contains that change needs to modified. We do not need to change other unrelated files. Modularization saves us lots of time and effort when changes need to be made in a large program.

2. Inheritance also helps us to easily make changes to our program. For example, in our program, the player race class (e.g. shade, goblin) is inherited from Player class. Player class contains all common fields and methods that every player race share. Player race class only contains their unique fields and methods. By doing this, we can save lots of effort if we want to introduce a new player race(or enemy race). The only thing we need to do is to create a new race class inherited from Player class, then the new race class will inherit all fields and methods in Player class. If this new race has special ability, we can add methods in it. In this way, we do not need to implement all methods again for a new player race class.

3. Change in rules.
   Let's take Player class as an example. Suppose the attack rule need to be changed.
   Case 1: The attack rule for the majority of races needs to be changed. (e.g. player can attack enemy twice). In this case, we only need to change the implementation of attack method in Player class. Then, all subclass of Player will inherit the new attack method. We do not need to change their attack method separately.
   Case 2: The attack rule for a certain player race needs to be changed. In this case, we only need to override attack method in that particular player race class. Nothing else needs to be changed.

4. Change in input syntax.
   Suppose when selecting player race, we want the player to enter "shade", "vampire" or etc to specify their race instead of entering "s" , "v" or etc. We only need to change related methods in Floor class. For example, in "CreatePC" methods, we need to change the condition if (input == "s") to if(input == "shade"). If we wan to change the input syntax for direction (e.g. "n" stands for north direction rather than "no"), we need to change the move method in Player class. No method in Floor class needs to be changed since if player moves, the "update_player" method in floor class will call "move" method in Player class and pass the parameter to "move" method.

5. Introduce new features.
   Suppose we want enemies can also pick up treasures. We only need to add a method called getgold(Treasure *) in the Enemy class. We do not need to modify other class like Player,

Floor or something. If we want the player can store some potions, we only need to add a field called potion_store and a method called storePotion(Potion *) in Player class. Again, we do not need to change other unrelated class like Enemy.
To sum up, if we need to introduce new features to a particular class, we only need to modify related classes, rather than all classes.

6. Cohesion.
   The common goal of all class modules is to achieve the interaction between the Player and all other objects on current floor. Enemy class enables enemies to move and attack player. Potion and Treasure class generate items that player can get on the current floor. Chamber class is responsible for spawning all objects on floors. Player class enables player to attack enemies, move, use potions and get gold. Floor class is responsible for updating the display of the current floor. Our main function will call methods provided in Floor class to start the game.

7. Coupling.
   There is still some interaction between modules. For example, the attack method in Player class needs Enemy as a parameter and the attack method in Enemy class needs a Player as parameter. In order to minimize circular dependencies, we used "forward declaration". We eliminated the interaction between Enemy class and Potion class. Enemies can not pick up potions from the floor. We also eliminated the interaction between Chamber class and other objects class. Chamber class is only included in Floor class. I think we can try to achieve lower coupling if we have enough time.

Answers to Questions

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

   Solution: We implemented a super class called 'Player', which included common fields and methods shared between player races. And then we implemented each race as subclass of Player. When we want to add additional races, we just need to add other subclasses (eg: like add otherRace.h and otherRace.cc). By doing this, we don't need to modify the super class and other subclasses.

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

   Solution: First, we implemented a super class called 'Enemy', and then implemented six

subclasses for each enemy race.
The CreateEnemy method in Floor class is responsible for generating enemies. Since enemy have specific possibility distribution of being spawned, we can have 0 ~ 3 represent human, 4~6 dwarf, 7~11 halfling, 12~13 elf, 14~15 orc, 16~17 merchant, and use the function *rand* in <cstdlib> to generate a random integer in the range 0~17. This random integer will determine the type of enemies I would generate.
We will then randomly choose a chamber and get the position of a random tile, after checking the availability of that tile, we assign the enemy to that tile.
It is different from generating our player character. The generation of player race depends on the user input while the generation of enemies follow certain possibility distribution. Besides, Player is only generated when we start or restart the game, but enemies will be generated when player go upstairs.

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

   Solution: At the beginning, we intended to use the visitor design pattern to implement the attack method for enemy character. However, we found that we would need to implement seven methods in Player class if we use visitor design pattern.
   To be more specific, we need to implement getAttack(Human *), getAttack(Dwarf *), getAttack(Elf *), etc in Player class and also override those methods in player subclass. Besides, only Elf and Dragon have different attack rules.
   Finally, we decided to implement an attack method in Enemy class, which is a super class of all enemy races. We override attack method in Elf and Dragon class. By doing this, we only need to implement three attack methods, which is much better than seven methods required in visitor design pattern.
   For other enemy abilities, if the ability is shared in all enemy races, we would implement it in Enemy class, otherwise, we would implement it in subclass.
   We used the same technique for player character races. However, there is still something different. We also used visitor design pattern to implement potion effects on player.

4. The Decorator and Strategy [8] 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 weighing the advantages/disadvantages of the two patterns.

   Solution: Strategy Pattern will work better.
   The advantage of strategy pattern is that our algorithms (like usePotion) can be reused and we can change which methods to be used at runtime. With strategy pattern, we can modify our objects at run time instead of wrapping the objects lots of times. It is also easy to add another

kind of potion.
The disadvantage of strategy pattern is that clients must be aware of different strategies.
The advantage of Decorator pattern is that we can wrap a component with any number of decorator and we can modify the object at run time instead of going back to change.
The disadvantage of Decorator pattern is that it can result in a lot of small objects. Using decorator pattern can complicate the process of instantiating the component because we need to wrap the component many times.

5. 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?

   Solution: We implemented the super class called 'Item' and two subclasses called 'Treasure' and 'Potion'.
   We need to determine if treasure and potions information are available or not. We also need to set treasure and potion information to be available. Therefore, we implemented isAvaliable() and setAvaliable() methods in Item class. By doing this, we do not need to implement these two methods twice.

Final Questions

1. 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?

We have learned a lot from this project, such as the converting the programming knowledge in practice and implementation skills. However, there are more important lessons taught by this project. To specify, the importance of communication, overall plan and responsibility.

From the experience of completing this project, the most important thing we have learned, about developing software in teams, is the significance of communication. From the startup to the deadline of this project, the project was discussed in our group throughout its construction.

There is no doubt that every individual is distinct and independent so that conflicts about the program designing are unavoidable. However, we discussed and decided to first draw a simplified UML for an overall plan of the project and then have a meeting to discuss the best solution combining our points of view. After that we communicated to assign different parts of project to each person. Due to the communication, each member of our group takes the responsibility she is good at. Then in the detailed implementations, if there is something we are not sure or we need to make change to our plan then we will discuss about that through the mobile social apps to get a quick solution.

Another lesson we learned from the project is to make a detailed and practical plan before actual implementation. There is no way for a team to develop a complicated program without a good plan. On the plan development, we all kept patient and carful so that the plan is made useful for the further implementations. We spared no effort on establishing a plan of the game since we all agree that the plan can be a guide of implementation, rather than only a schedule of completion dates, if it is designed well. This is finally proved by our implementation progress.

Also, responsibility is crucial to the teamwork in software development. We are fortunate that in our groups all members take their responsibility seriously and no one tried to avoid contributing to the software development. Since work has been assigned to each person, then it is the duty of that person to do it well. The goal of developing the project successfully will not be achieved without the effort of any member.

Thanks to this project, we have learned the importance of communication, plan design and sense of responsibility in developing software in team.

2. What would you have done differently if you had the chance to start over?

First of all, we would start the project earlier if we had the chance to start over. This project was released some days before we started to plan for it. However, we delayed the startup date for the project and found the time is limited later. So, if we are given the opportunity we would move up the software development.

Secondly, we would spend some time on conducting research about the information of the type of game we designed, CC3k, and even play some games before the planning of the project to get more familiar with the game rules. That will help us with the software construction to a great extent.

Finally, we will record our progress and the problems we met during this progress more frequently, for the convenience of changing the structure of project, and also for the later maintenances and updates of the game.