Student ID #: s56gao h63lu

Final Report of CC3k

Contents:

Part I. Answers to questions

Question 1: **How could you design your system so that each race could be easily generated? Additionally, how difficult does such as solution make adding additional classes?**

Answer: We plan to use inheritance relationship. We design a base called Player, which is a subclass of Character, and all classes – Human, Dwarf, Elves, and Orc – are subclass of Player. In our design, we decide to use template method pattern, where we give the base Player the implementation of increaseHP(), decreaseHP(), etc. and all of its subclasses and override these method based on the property of different race. Also, it is not hard to add new classes from Player. You just need to override the method which implement the unique properties of new player and inherit all other methods from superclass Player.

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

Answer: We use the same method to generate enemy character as we do with player character. As Player class, we still use template method pattern. We write methods attack(Player &p) and isHostile() and all of subclasses inherit these methods. At the same time, we write a virtual method getAttack(Player &P) and the subclasses can override it based on the different properties of enemies. Both of enemy class and player class are inherited from Character Class. The reason is that both player and enemy have some same property, for example, both of them have HP, Atk, Def. Thus, designing them as the subclasses of Character Class reduce the amount of repeated code we need to write.

Question 3: **How could you implement special abilities for different enemies? For example, gold stealing for globing, health regeneration for trolls, health stealing for vampires, etc.?**
Answer: Before we implement the code, we planned to write a method called specialAbility(). However, when we actually work on this part, we find it is much easier to add some codes in the getAttack(Player &p) and attack(Player &p) than to write a new method. Thus, we decide to use Template Method Pattern to override the attack(Player &p) and getAttack(Player &p) methods to implement the special ability of different enemies. For example, for Phoenix's getAttack(Player &p) methods, if the HP of Phoenix is lower than 0, it has 1/3 probability to revive. For Troll's getAttack(Player &p) methods, if it get attacked, it can regenerate its HP by 5 points.

Question 4: **What design pattern could you use to model the effects of temporary potions (Wound/Boost Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?**

Answer: We decide to use decorator pattern to implement this part of code. Every time the player uses a potion, like BA, BD, WA, and WD, we create a new player called special with the same curHP, curAtk and curDef as the original one and put the effect of these potions on the newly created player. Also, every time the player got attacked, both the HP of original and newly created players will be reduced. Every time the player use the potions, RH and PH, the effect of these two potion will influence the hp of special player and original player. And when the player goes to the next floor, these newly created player will be destroyed, leaving the original player intact.

Question 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 duplicated code?**
Answer: Before we implemented this part of code, we decided we write a class called potion and make the 6 different potions as subclass of this class. And we make potion class and treasure class both as a subclass of the Object class. However, when we implemented this code, we found that even though the 6 kinds of potions have different property, the

difference is trivial and it is complicated if we write a 6 subclass. Thus, we decided to write one potion class, with a field called function, which is a string to record the function of potions. This reduces the amount of code we write. Also, both the treasure and potion class are subclass of object class, which inherit the fields, such as x, y, symbol, name. Therefore, when we generate potion and treasure, we do not need to write duplicated code.

Part II. Plan of Attack

First, on July 10 (Sunday), we will discuss how to design this project and have a big idea about different parts of this project. We decide that first two days of this week we will focus on the design this project, find the relationship between different parts and have a draft of the UML diagram.

Second, starting from July 12 (Tuesday), we will start work on the .h files and make adjustment to the draft of the UML. And at the end of July 13 (Wednesday), we will have a final UML (still needed to be adjusted) of this game.

Third, starting from July 14 (Thursday), we will start work on the implementation of this project and make adjustment of .h files and UML. The following is the specific spilt of the process of implementation:

<table>
<tr><th colspan="2">Split of Work</th><th>Time</th></tr>
<tr><td>Hongyi Lu</td><td>Shang Gao</td><td>Expected finish Date</td></tr>
<tr><td>Item
(Potions, Treasures)</td><td>Characters
(Player, Enemy)</td><td>Thursday July.14 – Saturday July 16</td></tr>
<tr><td colspan="2">Meeting
have a discussion to improve the code
Debug</td><td>Sunday July. 17</td></tr>
<tr><td>Floor
(Construction of the floor)</td><td>Floor
(Random Generation of the items of floor)</td><td>Monday July 18 –
Tuesday July 19</td></tr>
<tr><td colspan="2">Meeting
Have a discussion to improve the code
Debug</td><td>Wednesday July 20</td></tr>
<tr><td colspan="2">Work on the attack part together
Work on the generation of different level together</td><td>Thursday July 21</td></tr>
</table>

Finish the display and finish most of this project (Hope to see the actual result after run the code)		Friday July 22
Continue work on the previous part of code and improve them	Bonus: Add ability to merchant to sell	Saturday July 23
Meeting Have a discussion to improve the code Debug have a final version of UML and improve all of the answers to questions		Sunday July 24- Monday July 25

Part III. Overview of the design

When we discussed the design of this large project, we decided that the whole project should be divided into two parts. The first part is different objects on the floor (wall, door, passage, tile, Space, player, enemy, potion, treasure). The second part is the structure (chambers and floor) and the control center of this game.

For the first part, we first write the base class, called **Object**, with fields x, y, name and symbol. All of the objects inherit from this class, because all of the objects have the same fields. And then we write a class, called **Character,** with fields beginHP, beginAtk, beginDef, curHP, curAtk, curDef. The Character class is a superclass of Player and Enemy. The reason is that both player and enemy have some similar properties and we want to put them together to make the whole design more intact and clear. Also, inheriting from the Object class, the classes Vwall, Hwall, Door, Passage, Space, Tile, Stair reuse as much code as possible.

For the second part, the **Chamber** class is a container of the tile and have some methods to modify the tiles in the chambers. The reason why we design this class is that we want to group the tiles and access them easily. The biggest class of our project is the **Floor** class. This class is mainly responsible for spawning different items onto the floor and controlling the attack and movement of player and enemies. The other class, called **TextDisplay**, is responsible for displaying the whole floor on the screen. It is an observer of all of objects on the floor.

Therefore, the whole design incorporates the ideas of **Model-View-Controller**. The object class and all of its subclasses are the model, which maintain the data of the game. The floor

class is the controller, which is a connector between the model and viewer. The TextDisplay is the viewer.

For the design patterns, we used observer pattern, decorator pattern, visitor pattern, template method pattern to design different part of the project. We will talk about them in details:

1) We use **observer pattern** in three places. The first place is that we let the enemy Merchant to be the observer of all other Merchants. Thus, when one Merchant gets attacked, it can notify all of its observers and make all other Merchants hostile to the player. The second place is in the design of TextDisplay. The class TextDisplay is an observer of all of the object on the floor. Every time the state of the objects changes, it notifies the TextDisplay and TextDisplay reacts to this changes by redrawing the whole floor on the screen. The third place is that every tile is an observer of all its neighbor tiles. The reason why we use Observer Pattern in this part is that when enemies and player move from one tile to another one, the tile which the character step on can observer which neighbor tile is not occupied and inform the character which tile to move on.
2) We use **decorator patter**n in the design of the Player. When the player uses the potions (BA, BD, WA, WD), we create a new Player called special with the same HP, Atk, Def and goldNum as the original one, and add the effect of the potion onto this special player. From that point, every time the Player uses a Potion, the effect of potion (BA, BD, WA, WD) will only influence the special player, and the effect of the potions (RH, PH) will influence both the original player and special player. When the player goes to the next floor, we delete this player, leaving the original player intact.
3) We use **visitor pattern** in the implementation of combats and use of potion and treasure. The reason we use this design pattern is that different player and enemies have different effect on the HP and gold. So every time we attack, or get attacked, or use potions and treasures, we need to make a decision of the effect based on the two objects. Thus, visitor pattern can solve this problem easily.
4) For the **template method pattern**, we use it in the implementation of player class and enemy class. The reason is that all of the subclass of player and enemy have some similar attributes, but still have unique features. So using this pattern, we provide a skeleton of the object and let the subclass implement some methods

based on the type of players and enemies. And for the methods getAttack(Player &p) and attack(Player &p) in the Enemy class, we also use template method. The reason why we decided to use this pattern is because different enemies have different special ability when they attack player or get attacked.

Finally, for the cohesion and coupling of the whole project, we tried our best to reduce coupling and increase the cohesion of different modules. And we thought we made a great job. The reason is that the only completest dependencies that our project has is that the Floor class need to depend on other objects classes and use their method. For cohesion, different module has different goals. For example, for the Chamber class, its only responsibility is to monitor tiles in the chambers and all of the methods in this class is related to modifying the tiles. For the Floor class, its responsibility is to generate items on the floor and control the action of player and enemy. All of the private and public methods have a relation with these two functions.

Part IV. How to accommodate changes

First, if you want to add new object onto the floor, such as new player races or new enemies, you just need to create a new subclass of the corresponding superclass. In our design, we consider the situation where we need to change the objects or add new player and enemy, so we put all of the common implementation in the superclass. Therefore, adding new race just requires designer to implement the unique part of this new race without writing duplicated code or changing the existing code and overriding some virtual methods of superclass. For example, if we want to add new races to player, such as knight, we only need to write a new class and implement the attack and getAttack methods to add special abilities to this race without changing previously finished code. Also, in our design, we use visitor pattern to implement the combating part, using of potions and getting treasures. The designer does not need to change this part of code when they add new races because all of the code do no need to know the type of the parameter in advance and they can make a decision of which function to use during the run time.

Second, if designer wants to change the rule of the game, for example, the designer wants to change the mechanism of the random generation of the items on the floor, he/she just

need to change the corresponding probability in the random generating functions, which requires little work to finish.

Third, if the designer wants to change the layout of the game floor, he/she just need to provide a different default floor layout file and changing the code related to the location of different chamber in the Floor class. This still does not require lots of changes.

Also, in our project, we follow strictly the rules of compilation dependencies, and we reduce the unnecessary includes as much as possible and use class declaration whenever possible. So when you change the .h files because you add new features, it usually requires one or two program to recompile, which is very time efficient.

Part V. 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?

   In this project, we learned that the communication in teams is important because implementing a program with others people is different from implementing by oneself. You need to communicate with your team members to divide the job and to combine different codes implemented by different people every day. For larger programs, we find it is efficient to write the program if we can find where we can implement separately and where we need to implement together.

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

   We would use pimpl idiom if we could start over to reduce the recompilation time if we want to change the fields of the class. Also, we want to figure out a way to divide the Floor class, because our floor class has two responsibilities (spawning items and control for the action of player and enemies). We want to follow the principle to let our class to have only one responsibility.