Final Project – Design Document: CC3K

Kang Shen

Yawen Chong

Zewen Lu

Faculty of Mathematics, University of Waterloo

CS 246: Object-Oriented Software Development

December 15, 2021

Contents

Overview

In this project, we designed and implemented a simple rogue-like game.

Our implementation can be divided into four main parts: Gamesys, Floor, Character and Item.

The purpose and structure of each part:

Gamesys: We use Gamesys to start the game and display the game to the screen.

Floor: We determine the five different chambers in Floor, and the five floors in the dungeon.

Character: Character has two subclasses: Player and Enemy. Player has five subclasses representing all player races. Enemy has seven subclasses representing all enemy races.

Item: Item has two subclasses: Potion and Gold. Potion has six subclasses representing all potion types. Gold has four subclasses representing all gold types.

Relationships between each part:

In Gamesys, we used Floor to generate all five floors' maps. Each floor owns 1 player, 20 enemies and 20 items (10 golds and 10 potions). Enemies and items are re-generated in each floor when initializing. When the player is no longer on the previous floor, Gamesys would determine whether the player reaches a stair, dies, or restarts the game.

The DLC part:

We will add a derived class DlcFloor and a new initializer DlcGamesys when the player imports the DLC. In the DLC, a new enemy, DrX, will be added. (See **Extra Credit Features** section for details about how to import DLC and how it works.) Other code structures are similar to the base game.

Design

Starting floors:

We used a class called Gamesys to control the progress of the game, and also store the different floors as vectors. Gamesys is responsible for loading the map of cc3k, and storing the info into different floors from 1 to 5.

Differentiate the chambers:

It's not easy to differentiate the chamber just by reading the base map, so we actually create an additional map which has numbers inside the chamber to record the positions of different chambers. Therefore, we can store the coordinates of each chamber using vectors of a structure of coordinates.

Starting the player and stairs:

Since the player is independent from different floors, we also store the player in our gamesys class in order to function properly throughout all the floors. The generation of the player is firstly choosing a random chamber and secondly recording the chosen position. Stairs generation is pretty much similar to the method we used for the player.

Starting the enemies:

We first use a vector of enemies in our floor class because each set of enemies are controlled by different floors. So basically each floor owns that set of enemies, and the floor class is responsible for initiating, moving the enemies. Also, we include <random> to generate random numbers from 0 to 1799 and create an array to store different types for enemies according to the given probability, and then choose randomly from the array. This method can simulate the random selection with given probabilities. For example, storing 400 strings called Human inside that array with size 1800 can represent the probability of 2/9.

Movements for enemies and the player:

The overall methods for moving enemies and moving the player are pretty much the same by reading the target position coordinate, and then setting the tile of the floor accordingly. In order to avoid conflicting with other objects, both enemies and the player need to record their previous position's information. In addition, enemies use random numbers from 1-8 representing 8 different directions, and then decide if the chosen direction is occupied, if it does, it will choose another random number until enumerated all possible directions.

Auto attack for enemies:

We first used a vector to store the hostile objects including the dragon hoard around the player, and looping the attacking method of enemies to attack the player within one block radius before the player's action. The attacking method uses the player's pointer as a parameter to control the data.

Attack for the player:

The method for attacking the enemies is very much similar to developing the attacking algorithm of attacking the player, but we use a similar moving algorithm to implement the attacking directions. Now the player can choose directions to attack the enemy if the enemy exists in that direction.

Restart and finish the game:

We use main.cc to restart or end the game. So, our gamesys.cc is to control each game, but main.cc is to control different games. We use two fields called end and re (restart) to mark the progress of the current game. After each turn, gamesys will check if the end or re is true to decide if it should give the control back to main.cc

Frozen movement of enemies:

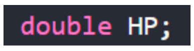
This is simply done by setting an integer field (initially set to 1). Each time for pressing the f keystroke, this field will multiply by -1. When the gamesys detects the value of this field is positive -1, it won't call the movement method for enemies.

Dragons and their hoards:

We construct the dragon hoards by creating the dragon to link them together. A dragon pointer inside the dragon's hoard field. So, the dragons can protect their bounded hoards and destroy the bound once the dragon is eliminated. Then the player can pick up the dragon hoard.

Effects of the potions:

Since the effects of RH and PH potions are permanent, when the player uses them, their HP field will be changed permanently.

(The 

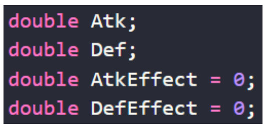
```
double HP;
```

field can be found in *character.cc*)

For potions BA, BD, WA, WD, we have fields “AtkEffect” and “DefEffect” to store their temporary effects. Their effects will be stored as int, and the actual Atk and Def will be changed in terms of these fields: “AtkEffect”, “DefEffect”, “Atk”, “Def”, where “Atk” and “Def” is the player’s Atk and Def without effects.

For example, we have a player with race Shade. It’s “Atk” field initially is 25. If the player uses BA, “AtkEffect” will be 0 + 5 = 5 since BA increases Atk by 5. Now, the actual Atk will be “Atk” + “AtkEffect” = 30. Then, if the player uses WA, “AtkEffect” will be 5 + (-5) = 0. The actual Atk now is “Atk” + “AtkEffect” = 25.

When the player moves to a new floor, “AtkEffect” and “DefEffect” will be reset to 0, so that all temporary effects will be gone.

(You can find these fields: 

```
double Atk;
double Def;
double AtkEffect = 0;
double DefEffect = 0;
```

in *character.cc*)

Resilience to Change

Since we used inheritance for many different classes, when changes occur in the specification, we can change our design easily.

For example:

1. If we need to add any additional characteristics to a certain character or item, we just need to add it in the corresponding subclass.
2. If we want to add new races of player and enemy or new types of gold and potion, we just need to add a new subclass to superclass Player, Enemy, Gold or Potion.
3. If we wish to add a new feature to the game, we can create a new subclass whose superclass is Gamesys or Floor.

Answers to Questions

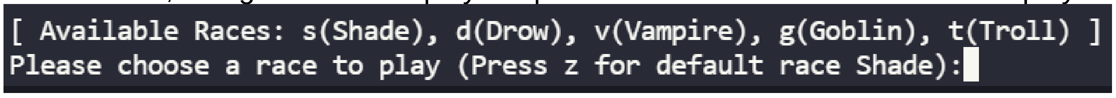
Question 1 (2.1 Player Character):

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?

<u>Answer</u>:

Similar as we talked about in our plan, we implemented all player races to be the subclasses of the class Player which inherited from the abstract class Character. Then, by polymorphism, each race's special characteristics and abilities can be implemented in their own classes.

Furthermore, we give the real player options to choose a desired race to play:



```
[ Available Races: s(Shade), d(Drow), v(Vampire), g(Goblin), t(Troll) ]
Please choose a race to play (Press z for default race Shade):
```

Therefore, each race could be easily generated.

It is easy to add additional races. We can add a new subclass to the superclass Player and implement the new race's characteristics there.

Question 2 (2.2 Enemies):

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

<u>Answer</u>:

Similar to implementing player races, each enemy race is implemented as a subclass of the class Enemy which inherits from the abstract class Character. Therefore, the player and enemies will share the same fields in Character class, such as "Def" and "Atk".

However, there are some differences, too. The player and enemies have their own unique fields. For example, the player has field "score", and enemies have field "hostile". Also, in the game, there is no option to choose the enemy races. All enemies are generated with certain probability in random chambers on each floor. Furthermore, the player will be generated only once per game, but enemies will be generated for several times (for each floor, we need to generate 20 enemies with different races).

Question 3 (2.2 Enemies):

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:

We override the attack method for those that have special skills for attack. (Elf, Orcs, Merchant, vampire). For drows, we check the race of the player when using the potions. For halfling, we check the race of enemy in Player::attack(Enemy* enemy). For dragon and goblin, we change the corresponding status when we run the game.

The technique we used is inheritance and polymorphism which are the same as we did for player races. However, the detailed techniques may be different since we have special enemy classes (dragon and merchant) who have pointers which point to gold.

Question 4 (2.3.1 Potions):

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:

We still think the strategy pattern will be a better option because of these reasons as we previously provided in our plan:

The advantage of the strategy pattern is that it allows us to change the effects of potions used by the player at runtime, and new strategies can be introduced without changing other parts of the code. We can also modify the effects of the potions based on the race of the player character and whether the effects of the potion to be used would be permanent. One of the disadvantages is that we have to reset the player's statistics after he arrives on the next floor.

The decorator pattern can combine the effects of different types of potions easily. However, it can only add additional functionality to the existing functionalities of potions, and it's difficult to remove a specific decorator from many decorators.

However, we did not use any design pattern to implement the effects of Potion. We used a simple strategy which is described in the **Design** section above. (You can find it in “Design: Effects of the potions”.)

The reasons why we used this strategy are:

1. The effects are only about decreasing and increasing Atk and Def. By using our method, each effect can be tracked and reset easily.
2. Compared with our strategy, the design pattern will make our code more complicated.

Question 5 (2.3.2 Treasure):

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:

We did exactly what we described in our plan.

We implemented an abstract class which is called Items. Then, we created two subclasses, Potion and Gold (treasure), for the superclass Item. Therefore, we can avoid duplicate code by inheritance.

Extra Credit Features

1. Instead of using arrays, we used vectors in our project.
2. Players can determine the level of challenge they face at the beginning of the game (Easy or Normal). They are teleported to the fifth floor if they choose the easy mode.
3. We provided a free DLC for players. We added a new type of enemy called Dr.X which is represented as the symbol X on the map. Dr.X moves randomly within the five chambers. If the player is in the same chamber as Dr.X, the player has ⅓ chance to die right away. This feature is added when the user runs the game as "./cc3k dlc" (quotes removed).

Final Questions

a) *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?*

Answer:

After doing the project with my teammates, I realized that it is important to keep communicating with each other. Before starting the project, we need to discuss our plan clearly, so that everyone has their parts to do, and we can finish the project cooperatively. Then, during the implementation. When you come up with a new idea about the design, it's beneficial to tell your teammates since you can discuss it together and may find a better solution. Also, it's important to update your progress with your teammates and explain your work clearly to them. If one person changed or added something into the code without telling the group, then unexpected errors may occur to make the debugging time-consuming.

Another important thing is to set up and follow deadlines of each part of the project. For example, those "Estimated Completion Dates" we wrote in our plan. It is helpful for improving our motivation and efficiency since we have clear goals to achieve.

b) *What would you have done differently if you had the chance to start over?*

Answer:

1. We will consider using design patterns for effects of potions. Though our provided strategy is simpler, the design pattern will be a safer choice if any new potion effects are needed. In other words, our method may only work for what is required in the current specification. When the potion effect becomes more complicated (for example, decreasing Atk by 10, then multiplying it by 10, etc.), we need many more arithmetic calculations to track and reset each effect, which may lead to more complicated codes and many errors. Thus, using design patterns will be a safer and easier choice in this case.

2. We will consider using factory design patterns for the generation of enemies, so that we can add a difficult mode with more enemies generated on each floor.

Cohesion & Coupling

The structure is similar to the base part. We tried our best to maximize cohesion and minimize coupling. Every function performs one single task and we use inheritance to produce enemies, players, golds, and potions so that we can add new types of characters and items easily without modifying the other classes. We use different classes to start the game with/ without DLC so that the DLC part wouldn't affect the basic part.
However, we cannot completely reduce coupling. Since we have to implement different abilities of characters, the Enemy class and the Player class must be dependent on each other. Also, the Item class is dependent on the Character class since Dragonhoard is protected by a Dragon, and potion effects are related to the race of the player, so that we cannot reduce dependency.