CS246 Final Project (CC3K+) DD2 Documentation

Yang, Xiu
Huang, Yongjia
Yue, Zhongqi

Introduction

The game of Chamber Clawer 3000+ is a simplified rogue-like. In the game, the board consists of 79 columns wide and 30 rows high (5 rows are reserved for displaying information). Game play is as follows: The player character moves through a dungeon and slays enemies and collects treasure until reaching the end of the dungeon (where the end of the dungeon is the 5th floor). A dungeon consists of different floors, which consist of chambers connected with passages. In this project implementation, each floor will always consist of the same 5 chambers connected in the exact same way.

In the game, player can slay enemies to obtain golds, use potions that may enhance character's abilities or harm them. The compass, which is with one of the enemies, can be obtained after the player kills that particular enemy. Then a stairway will appear on the map, with the path to the next floor. However, if player accidentally step on the location where the stairway locates before compass is found, player will go to the next floor directly.

The game ends in one of the following three conditions: 1, The player's HP reaches 0. 2, The player reaches the top floor and wins. 3, The player chooses to quit or restart the game. If the game ends when the third situation happens, a new map will be printed to player after restarting.

In the implementation of this game, massive OOP design concepts were used and the concept of class inheritance, polymorphism, design pattern play a crucial role.

Overview

There are four main components in this implementation. They are Character component, Enemy component, Potion component and Map component. In each component, there are several inheritance and aggregation relationships among classes.

In Character component, there is an abstract base class PC, containing all the common fields of a PC, such as HP, Atk, Def and gold. Additionally, two Boolean values, compass and BS are used to track if PC has equipped with compass and barrier suit, as this will affect the display and damage calculating. Then, concrete classes of pc such as human, orc inherit from the base class. Extra skills and abilities of PC are stored in their own classes. Accessors and mutators are implemented in these classes to facilitate implementation as well as keep encapsulation. Moreover, TakePotion() and Combat() are implemented for PC to use a potion, or attack an enemy in a combat.

In Enemy component, there is an abstract enemy base class, inherited by the other enemy classes, such as class Troll, class Vampire and other enemy classes. Accessors and mutators are also implemented to keep encapsulation.

In Potion component, abstract base class AbstractPotion is used. Then, all six potion classes are inherited from base class AbstractPotion.

In Map component, observer design pattern is used. There are four major classes, class Subject, class Observer, class Cell, and class Map. Subject and Observer are two abstract base classes to achieve Observer design pattern. In Subject, there is a vector of Observer pointers for later notifying uses. Class Cell represents one single location (a floor tile) on the floor and inherits from both Subject and Observer. Pointers to enemy and potion are stored in each cell and are used to connect each cell to what is in the cell. For example, if a cell has an enemy in the cell currently, a pointer to an enemy object is used to connect this cell with the actual enemy object. In the meantime, pointer to potion field will be null pointer. Each cell also contains a state attribute (an enumeration class), including all possible states of a cell, such as gold, each type of enemy, PC, each type of potion and so on. This state will benefit in combat and random generating. Finally, class Map connects everything together, with a vector of vector of Cell attribute. Methods such as Map::usePotion() will call PC::TakePotion() in class PC. Similar design is implemented to attack enemy, which Map::attackEnemy()

will call PC::Combat(Enemy*) and Enemy::Combat(PC*). Other methods are implemented to move pc, collect gold, and display status to user.

Design

Challenge 1: repetitive code for all PC and enemies

Solution: Solution to this challenge is to use concept of abstract class, inheritance, virtual/pure virtual method, polymorphism and polymorphic vectors. Since all PCs and enemies are similar, we then created abstract base class for PC and enemy, so that all common fields are stored in the base class. Then, subclasses can inherit these fields and also override methods that exist in base class. Such way of using virtual/pure virtual method will achieve polymorphism and compiler will know which the most accurate method is to call.

Challenge 2: how to make Merchant non-hostile at first, then become hostile after being attacked

Challenge 3: how to move enemy randomly only once, movement of enemy can happen twice sometimes

Solution: The above two challenges share similar solutions. For challenge 2, since Merchant is not hostile at first, we added a Boolean value attackMerchant to indicate if merchant has been attacked before. This value is set to false initially, but once the player attacks Merchant, the value will be set to true. Then all the merchants that player encounters after will be hostile to the player. For challenge 3, solution is similar, by adding a Boolean value move, indicating if a particular enemy has been moved. At first, we were trying to move all the enemies, then display the floor to user. However, there is chances that two enemies would move to the same location, overlapping. Therefore, we came up with the plan of adding Boolean field.

Challenge 4: How to make sure enemies have a 50% chance to miss the attack

Solution: Since all numbers modulo 2 return a 1 or a 0, we then randomly generate a number from [0, 1] and deliberately set 1 as : “not miss”, and 0 as “miss”. Therefore, if a 1 is generated, we would call PC::Combat(Enemy*), otherwise, we will print the information to user that 0 damage is dealt to pc.

Things that have done differently from DD1 design

1, We planned to use Decorator design pattern to track potion use of player but chose not to in the end. The reason is that C++ has containers such as vector, that can easily help us resolve this problem. We have put a vector as a field to PC, so that whenever a potion is used, we would put the potion into the vector. In this way, it is easier to implement and easier to debug. When player enters the next floor, we would eliminate the temporary potions and add back/minus the value to the corresponding field in PC. This will then achieve flexibility and similar effects of decorator design pattern.

2, We planned to use Visitor design pattern but chose not to in the end. We were not sure if we would use visitor design pattern correctly, so we find another way of doing it to make sure the code will compile and behave correctly. Instead, we chose to pass a pointer to an enemy to Combat function, and PC will act accordingly based on what is passed in.

Resilience to Change

The entire project is built upon inheritance, design pattern and different relationships among classes. Inheritance can connect base class with all of its derived classes. Adding more player characters, enemy types or potion types can be easily done by adding a new derived class. Then the new derived class can override some of the methods in base class to achieve its own functionality. This all thanks to inheritance and virtual method so that we have polymorphism. If the new feature is like compass or barrier suit, then adding a new private field in the corresponding class would be sufficient and easy to add. We can always add in accessors or mutators if appropriate. If we want to add new skills for a character or an enemy, then we can add in new method that deals with it.

Another reason that the project can deal with changes well is that one file/one class only deals with one small part of the project. For example, class cell only deals with what could happen in one single cell. Class TextDisplay only deals with updating the right information to user. And class Map only handles the floor and manipulate cells. Since cell's functionalities have already dealt by class Cell, class Map do not need to

care/solve the problem in each cell, and only need to manipulate cells correctly. All the small pieces just mentioned work together, so that the game can be displayed to, interacted with users.

Answers to Questions

Question 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 classes?

Answer: We used concepts regarding inheritance, abstract base class to generate each race. We first defined an abstract base class, which will not construct any object. Then each race will inherit from the base class, since they share same fields. Unique skills of player characters can be implemented in their own concrete classes, without influencing other classes. Moreover, adding new races/classes would be easy. Simply enable the new race to inherit from the same base class as other derived classes already did. Any new race's additional features can be added in the new class.

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 used the same method when generating enemies, that is creating abstract base class and let concrete classes inherit from it. This part is the same because player characters share similar structures with enemies. To be specific, all player characters and all enemies share same common fields, but with little difference in some unique skills. This can be done similarly by inheritance. However, the difference is that player character only needs to be generated once, but enemies need to be generated multiple times with different probabilities. This is achieved by using function srand() in C++ library.

Question 3: How could you implement special abilities for different enemies? For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?

Answer: These functionalities are implemented by PC::Combat() function. Since most of these functionalities only occur when player is around one block radius from the enemy, we put these features in the functions that handle the encounter of player and enemy. The function will then react accordingly to what the PC is facing right now.

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: Two design pattern can achieve the same goal. Decorator design pattern and Strategy design pattern. They both have their own advantages and disadvantages. The advantage of Decorator is that it is flexible. Decorator works like a linked list, so that we can add in new decorator or remove existing decorators easily. However, the disadvantage is that when we actually implemented decorator design pattern, it does not work well with the process of constructing objects. In particular, we spent lots of time trying to initiate potion objects but failed. On the other hand, the advantage of Strategy is that it would allow the algorithms to vary independently of the client. However, since in this project, the effects of potions are simply adding/subtracting HP/Atk/Def. It is unnecessary to implement new algorithms and make them interchangeable.

Question 5: How could you generate items so that the generation of Treasure, Potions, and major items reuses as much code as possible? That is for example, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code? How could you reuse the code used to protect both dragon hordes and the Barrier Suit?

Answer: In our design, the board is made of cells, and every cell has an attribute called State which reflects what item is currently in that cell. For generating major items, the state of the cell is first changed to the item in that cell, such as State::Compass or State::BS. Moreover, in our design, the items can be divided into 2 categories: the items implemented into classes and items that are not implemented into classes. For the items that are implemented into classes, our Cell class has a pointer, pointing to that class's object. For items that are not implemented as a class, such as gold or major items, we only need to alter the state. In our design, inheritance is used for different abstract classes and different abstract classes are completely different. Thus, when items are generated in the cell, only the required code for generated that class is needed and no code is duplicated.

For the protected dragon hordes and barrier suit, the method is the same. We use a function that passes row and column of barrier suit/ dragon horde as parameters. Inside

the function, all available positions around the barrier suit/ dragon horde are recorded. Then a random number is generated in the range of 0 to the maximum number available (position – 1). Next, the dragon is placed in that position. By doing so, codes are reused.

Extra Credit Features

We have added a new character race called Saitama (One punch man). It has 10000 HP and 100 Def and 100 Atk. Adding this new race is fairly easy, since we used inheritance in PC component. This would result in just adding a new class, with no new attributes. All attributes are inherited from base class. We then override other methods so that we can achieve its unique functionality.

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?
Answer: Two things I learned from working with a team for the first time. First, communication is super important when doing teamwork. Communicating with team members about any questions you have, answering any questions they may have can greatly increase the efficiency of writing projects and debugging. Second, respecting each team member is very important. Let everyone have the chance to express their thoughts and feelings. This would improve the efficiency of team working.

2, What would you have done differently if you had the chance to start over?
Answer: Make a clearer plan and assign work in details. Before starting the project, we should go into as much details as possible, so that we can minimize the changes made later. Any changes made after assigning work will slow down the project greatly.