CS246 A5: CC3K

s53cao, y348gao, y58tao

Introduction:

In the game ChamberCrawler3000 (CC3K), 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 that consist of chambers connected with passages.

At the beginning of the game, the player will choose which version to play. In the normal version, player will play by entering commands. In the DLC version, player will play with WASD controls and have more race options for the player character.

1 Overview

1.1 Controller

Instead of having a long and meaningless main.cc, we encapsulate most of the control parts into normal_controller and dlc_controller. These two are the only class that interact with players and are designed to translate the commands from players into computer language to give instructions for our Grid and TextDisplay class.

1.2 Subject

All the items and characters in our games are subjects, which are been observed by Observers, namely the Grid and the Textdisplay. They are generated using Factory Pattern and interact with each other using Visitor Pattern. We have two big subclasses under the Subject class, Item and Character. Intuitively, Treasures and Potions inherit from Item while PCs and Enemies inherit from Character. By exploiting the inheritance and polymorphism feature, we avoid a great deal of code duplications since there is many similarities between Treasures and Potions, as well as PCs and Enemies.

1.3 Observer

What we used is not the typical kind of Observer Pattern because in our opinion Observer Pattern are error-prone and tedious to fulfill. As a result, we use Visitor Pattern to take over the job of Observer Pattern when a Character moves or attacks. However, we still use Observer Pattern for our Grid and TextDisplay because we need to auto-update positions and states of every Subject after each move, where Observer Pattern seems to be the optimal solution. To be more precise, Grid stores the information of everything while the the TextDisplay are responsible to convert the information from machine language into human language. Whenever an event happens, the program will automatically notify the Grid and TextDisplay to print something on the screen or change corresponding state.

2 Updated UML

(see the attached UML document in separate submission)

3 Design

3.1 Design Patterns

1. **Factory Pattern**

   In our design, all objects(PCs, NPCs, Treasure, Potion) are initialized using corresponding ‘Factories’ (pcFactory, enemyFactory, treasureFactory, poitonFactory). ‘Factories’ abstract the process of creating an object, so each time an randomly chosen object type is wanted, simply call Factory::create() which will return a shared_ptr of the object just been created.

2. **Observer Pattern**

   We use Observer Pattern mainly for our control and display. We have two important classes, Grid and TextDisplay, both of which are observers for all characters and items. Any changes or movements will notify Grid and TextDisplay, indicating that we should print certain notifications onto the screen. The advantage of using Observer Pattern is that we don’t have to update our current state by hand, all the updating will be done automatically by notifyobservers and notify functions.

3. **Combination of Strategy Pattern and Visitor Pattern**

   Strategy Pattern and Visitor Pattern are the most commonly used features in our program. We use the combination of them to achieve the attack and be attacked operations, which are the core of this game. We also used them to pick up treasuries and use potions.

   Strategy Pattern and Visitor Pattern have the competitive advantage of abstraction and information hiding. Take the attack of different kind enemies as an example. In this case, we only need to implement a general attack(shared_ptr<PC>) function, which takes a PC pointer, in the superclass Enemy. By doing so, there is no need for us to know the exact type of PC during the runtime. Neither do we need to know any information about PC because we will call attackBy(shared_ptr<Enemy>) and pass the remaining task back to PC. The Potion and Treasure class use the same strategy.

3.2 OOP Characteristics

1. **Encapsulation**

   First of all, we comply with the basic OPP encapsulation requirement. To be more specific, we obey the following rules:

   - Implementation details are sealed away and all fields are private, only methods are public.
   - Can only interact with the provided methods, which guarantees the invariance won't be broken.
   - We make no friend in our project, which reduces the probability of breaking the encapsulation or invariance.

   Moreover, another important features of our project is that we obey the pImpl idiom (aka the bridge pattern) throughout our programming for the sake of information hiding and abstraction, which enables us to be more flexible when we try to make a change. To be more specific, there is no need to recompile any other files except the file been changed.

2. **Inheritance & Polymorphism**

   Our design extensively implements and relies on the characteristics of inheritance and polymorphism. To minimum duplicated private fields and public methods of superclasses (Character, PC, Enemy, Potion, Treasure) and specific class such as Shade and Elf. Using inherited classes, the workload of implementing concrete objects is significantly smaller than what it would be without inheritance. Polymorphism can be also seen throughout our design. We won't specify the class until the attack method is called. More specifically, we determine the type during the runtime, which again avoid code duplication.

3.3 OOP Principles

1. **Cohension and Coupling (Separate compilation)**

   We claim that our project satisfies the basic high cohesion and low coupling requirement.

   - High Cohesion: Most of our modules have strongly and genuinely related elements, which work together for a common goal. One good example is our Enemy class. The only purpose of this class is to introduce the Enemy class and its related functions such as move and attack, without dealing with any other unrelated features.

- Low Coupling: Generally speaking, there is high degree of 'independence' among our modules. Our Factory Pattern and Visitor Pattern contribute to the minimum recompilation when the client try to make a change or add new features to our current program. We will give more explicit introductions and examples in the Design Pattern section.

2. **Resource acquisition is initialization (RAII)**

   One of the outstanding shining points of our project is that we make the greatest use of c++11 new features by using smart pointer to share information and use vector to store information instead of using raw pointers or arrays. This guarantees us there will be no memory leak and reduce the possibility of segmentation faults dramatically. On top of that, by using RAII, our codes are much more clear and easier to understand, which saves us a lot of time during the debugging.

4 Resilience to Change

1. **Use of "Factory Method Pattern"**

   We take advantage of inheritance and the "Factory Method Pattern" in order to be resilient to potential change such as new PCs, NPCs, Items and adding specific features for each of them.

2. **Applying "pImpl" diom**

   We create separate struct the private fields of 'Grid' and 'Character' classes so that less time will be spent on re-compilation when no information contained in these field will be exposed or vulnerable.

3. **Avoided Hardcoding**

   Instead of using raw numbers, we used meaningful constant and variable names as much as possible to be flexible when the fundamental design of the game is changing. For example, we could simply change the values of our constant to adapt new map dimensions.

4. **Use of "Observer Pattern"**

   By using "Observer Pattern", we can easily adapt multiplayer pattern or update new maps without changing a lot of codes. For example, if we want to have more than one players, all we need to do is to add different players as observers of each other, with a small adjustment to our game controller.

5 Answers to Questions

Player Character:

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?

    - There is a PC abstract class that has concrete subclasses each representing a race of the player character. After the user chooses the race, the player character will be created with its location and race using PCFactory.

    - To add an additional race, we only need to create a new subclass to the abstract PC class with features of this race defined in the corresponding concrete class, and then add this race to the PC race selection list.

Enemies:

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

    - There is an Enemy abstract class that has concrete subclasses each representing a race of enemy. Different enemies except dragon will be created with its location and race using EnemyFactory. Dragon will be created separately and affected by the generation of dragon hoard.

    - It is similar to the generation of the player character except that the player's race is chosen and enemies with different races are randomly generated according to specific probabilities. For the remaining, they uses similar factories to spawn characters randomly and use the same constructor from Character class to construct the objects. (Note: Dragon is a special case; it is different. We will place and initialize the Dragon separately due to its special properties (stationary location, relationship with dragon hoard).)

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

    - For attack and move, functions like attack, attacked by and move are pure or partial virtual in the abstract superclasses like Character and Enemy. The concrete classes for different enemy characters inherited them from the superclass and override the functions if necessary.

- For potion and treasure, different races have different factors, by changing the factors, we can implement the various abilities related to potion and treasure.

- We use similar techniques as for the player character races. We have an abstract superclass PC. With only the basic attributes in the superclass, every subclasses for player character races inherited from it will have their own implemented abilities.

Item:

1. 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 weighing the advantages/disadvantages of the two patterns.

    - The advantage of Strategy Pattern is that it requires less and much simpler encapsulation because in our case, we have 6 different types of potions, all of which have different behaviours. Thus, our base potion should have pure virtual behaviour and each subclasses have their own implementations, which make the max use of Strategy Pattern.

    - The disadvantage of Decorator Pattern is that we shouldn't have more than one effects on a single potion. Hence, it's kind of a waste to have 6 decorators as we know we can only use one of them.

    - In our own implementation, we will combine the Strategy Pattern with the Visitor Pattern in order to avoid code duplications. To be more specific, we will use six potions, namely the vistors, to visit two elements, i.e. Drows and Others.

Treasure:

1. 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?

    - On one hand, both Potions and Treasures are inherit from the superclass Item, in which case they will have similar fields and methods (ex: position, user). They will override to achieve their specific features and share the other methods for the sake of code reuse.

- On the other hand, for the generation, we randomly generate their positions using the same helper function randomSpawn in Grid to generate a random position yet not occupied by others and then give the potion to their corresponding factories to put their symbols at the position and randomly decide the specific type. By using the same helper function, we avoid duplicating code.

6 Extra Credit Features

1. WASD controls

In the dlc version, player plays the game using WASD controls. To implement it, we used the function getch() in the <curses.h> library to get inputs from keyboard. The challenging part was the display problem. It had output place problem (ex. if it prints “a” at first line starting at index 0, for the second line, it will prints from index 1 instead of what expected 0), echo and junk at first. To solve the problem, we use mvaddstr(int y, int x, const string str) instead of cout to do the display work and use clear() and refresh() at suitable time to avoid echo and junk. (To print the whole interface using mvaddstr, we put the content in the ostringstream “ss” in the “string print()” function in TextDisplay. At last of this function, we use “ss.str()” to get the content as a string “s” and return it. In the dlc controller, it gets the return and converts it into c-format string using “s.c_str()” and give to mvaddstr to display successfully.)

2. Use of Smart Pointers

Throughout our project, we didn’t use a single raw pointer. All the memory address are expressed using smart pointer. In order to achieve the goal of no memory leak, we struggled a little bit due to the fact that we are all beginners in the field of smart pointer. The most impressive and changeling problem about smart pointer is that when we pass ‘this’, which is a raw pointer, as an argument of function, we need to construct a brand new smart pointer, which proved to be error-prone. We solved this problem by defining our own destructor for smart pointer using lambda function.

3. More player character races

We have added a new PC called “Witch”. This extra feature is not challenging due to our design. Our design adapts the “Factory Method Pattern” and the idea of inheritance. Our PCFactory simplifys the process of constructing new character. Calling create() on one instance of PCFactory will automatly create desired PC. Adding “Witch” only requires us to insert this class in the PCFactory and its feature will inherit from its superclass. Any extra features that “Witch” are also set through set method in superclass.

4. Well-designed Interfaces

We redesigned welcome interfaces, PC selection interface and End Game interfaces to make our game more attractive.

7 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?
   - Individual's capability is limited and group work is important. Generally speaking, group work makes one plus one greater than two possible. Guided by a good leader, our team conquered many problems that are challenging when a single person works on the project. Each person has own area of expertise, someone may be better at designing project, others may have more experience on debugging. Working in a team, large project can be finished more efficiently.

   - From the experience, we learned that timely communication and clear necessary comments are important while developing software in teams.

   - When writing a large program, developing the right strategy or before writing code is essential. It may take a lot of time to develop the strategy, but it will significantly reduce the amount of time spent on revising and redoing the work.

2. What would you have done differently if you had the chance to start over?
   - Spending time on the right tool is essentiall.
     At the first place, we did all the our separately in outer own laptops, which causes our great deal of troubles due to the incorrect merge. It's not until the very last day did we find that we can actually submit our work to Gitlab and let the server do the merges automatically for us, which should have save us countless effort and time.

   - Using template.
     We may also take advantage of the Template in c++11. As our project proceed, we notice that we used enum class quite often and we should have made a template class for the sake of clearness and readability like we did in Assignment 04 Question 04. However, when we realize this, it's already too late to make that change.