Chamber Crawler 3000

Haobei Song (Aaron)
Shangshang Zheng (Daniel)
Junheng Wang (Eason)

Introduction

Chamber Crawler 3000 is a classical rogue-like game played on the terminal emulator on Linux supported platforms. Players are free to choose from a variety of different characters and fight their way out by combatting against fierce monsters along their adventure to the last floor. Meanwhile, players should come up with appropriate strategies such as enhance their different abilities by picking up potions, in order to survive the brutal, merciless cc3k environment created by CS246 staff and professors.

Overview

In our design, we adopted the Model-View-Controller pattern to organize the whole game structure in order to potentially accommodate to various platforms. Following general purpose programming principles in object-oriented software design, any objects on the floor such as players, enemies, potions, even tiles, doors, walls etc. were implemented as inherited classes from the superclass "Object" (which is also a class inherited from Subject). Besides, the visitor pattern was not only implemented between players and enemies but among any objects you can see whether movable or not when you play the game, so that in principle any two objects can interact with each other distinctively, which furthers the generalization purpose residing in object-oriented design. Observer pattern was also used quite often in our program to achieve interesting gameplay mechanism and to implement the interactive display with players. Another functionality we implemented successfully is the strategy pattern on the game controller such as to have user decide if they want the downloadable content extension during run-time.

Updated UML

1. The Object class was both Subject and Observers, since we thought every object need to know their neighbors so that the enemy can be notified if the player is around. Now only the Object class is Subject for the display purposes. We deal with the neighborhood problem by applying a much more efficient algorithm, which comparing the position between enemies and player to decide whether they can attack or not.
2. We used strategy pattern for two kinds of Merchant, since we thought we need to implement two different merchant styles and switch them during combat. Now we think it is unnecessary to implement two merchants, so we add a Boolean field to do the check, and make the design simpler.
3. For game controller, in order to switch from different version, we apply the strategy pattern. So basically, our main main controller owns three different kinds of game controller, each represents a different kind version. One is for test, one is for the normal version and finally the dlc version. And all of these three versions inherit from a superclass called GameController.

Design

Interactions (Attack, Move, Pickup) Among Objects (Visitor Pattern)

In our design, all the objects are inherited from the same class "Object", which is a subclass of "Subject", so that any changes in objects can be observed by corresponding observers (class "Observer") such as text display and ncurses display (DLC) and produce different output. We define the interactions between two objects quite generally as "visiting" with different types such that a player (say Shade) attacking an enemy (say Orc) is an interaction with "Type Attack", a player (Shade) or enemy (Orc) moving to a "Passage" are interactions ("Type Move") between player and enemy with cells but the former result is a "True" while the latter is a "False" and also a player (such as Shade) picking up a potion (any type) is a visiting between them with "Type Pickup". This generalized visitor design pattern enables us to change any features and in principle could accommodate to any further modifications since we could customize any interactions between two objects players could see while playing it.

Potion (Decorator Pattern)

The potion system is implemented as a decorator pattern decorating itself in our model as we take into consideration player picking up (or trade) infinite many potions in which strategy pattern couldn't be applied. Each player "has a" potion and we use a virtual method to obtain the player information which may have been modified by the potion. In this way, it makes it much easier and neat to write the functions dealing with combat between players and enemies as we could simply call the same methods without any concerns and upon a player reaching next floor, the "Potion" goes out of scope so the player information is reset automatically. The most challenging problem is to make hp change permanently. To deal with it, we update the hp of player every time we get access to its information.

Relation between Dragon and Dragon Hoard (Observer Pattern)

Since the dragon hoard can be picked up after the dragon is slain, we need to notify the dragon hoard then.

Initialize objects on the floor

Since different objects belong to different classes we build "similar" constructors with only two parameters for every concrete class inherited from class Object. While initializing the objects randomly we could simply call the same constructor with different type, which facilitates collaborative programming and suggests good habits when working on large scale software developments as a team.

Game control
(Model-View-Controller pattern and Strategy pattern to implement DLC)

Considering the MVC style, we managed to divide the model(floor), view(display), control(game controller) apart. We believe this is quite important for the game.

With this, our game can easily add any kinds of controller (wasd control for example), any kinds of display (text display or graphic display) without change anything inside the model. Strategy pattern allows us to easily switch from normal version to dlc version, we provide the user with three different main options, considering the convenience of the TA, we separate the test version with the other two versions. This we believe provides user the best experience ever.

Interactive Display observes every object on the floor(Observer Pattern)

In this part, instead of having only one display, we actually implements two display window, one is simply called window, which shows the floor and all the characters in the board. The other is called panel, which observes the player throughout the game. We use observer pattern to achieve the interaction between the user and our game.
Window acts as a global observer, which is attached to every single object in the board, and the panel is like a personal channel, which only track the status of the player. Thanks to this strategy, our program is able to display every single change appear inside the model, which is such a good way to for the program to communicate with our user.
Colorful display is also worth mentioned here, with this, we believe we've managed to push our display into next level. (See DLC for details).

Resilience to Change

Model change

Our program is based on the much general visitor patterns among all the objects on the floor and could in principle realize any interactions between any two objects regardless of their mobility or distance with others. We are able to add any new players which could have different abilities against different enemies by simply creating a new class and add methods dealing with all the different enemies inside of that newly built class. Moreover, we could also change the player's behaviour upon picking up different treasures (such as get extra money), using different potions (special enhancement) or even distinct behavior when moving on the cells (going through the wall or even fly over to the other chamber) by simply adding the methods in the corresponding classes with which you want the newly added player to have different behavior. This is the power of Visitor Pattern where we got rid of almost all the tedious if conditions (except for the displaying, which for the general design purpose should be able to accommodate different platform or different styles on the same platform) and still were able to modify both the visitor class and visited class exclusively without violating encapsulation rules compared to modifying everything outside of the to be modified class itself such as in the game controller. In addition, we implemented every single object in the game as a standalone class, which are all in the different files. In other words, all modules interact via function calls, modules only pass arrays and structures back and forth, modules affect each other's control flow, modules also share global data (buff file). And all modules cooperate to perform exactly one task(accomplish the game). With all this achieved, we can say our program reaches the so called low coupling and high cohesion.

Display Change

Since we implemented Model-View-Controller pattern in which the input, output and model are separated from each other, we can change the display of any objects into different style or format freely to adapt to various platform (terminal line or ncurses window (DLC)). We could just change the corresponding display characters (or colors (DLC)) without any modification in the model, as we assign each object in our main (Model) program a distinct enumerator, which could be identified by the display (View) and decide what to display as needed.

Game Control Change

To facilitate multiple usage of our program by different users, we resort to strategy pattern and create three different game controllers under the same base class. Namely, we built one game controller for normal user, one for test user and one for DLC user, but the commands are of the same as they would override corresponding virtual methods inherited from the super game controller class. Therefore, users can choose different version of the game based on their own need during run-time and don't have to remember too many commands of the same functionality within different versions.

Answers to Questions

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

All races inherit from Player superclass. When we generate a race, we can easily create an object with specific race type. It also makes it very easy to add additional classes because new race is a subclass of Player. The only thing we need to deal with is to implement the extra functionality for the new race and some other special relationship with other Objects. For instance, when we add a Vampire subclass, we need to implement the Visit method to let it gain HP when the action is set to be Attack. since the player class has implemented virtual methods(be_visit), which gives all its subclass the same ability being visited. Now we should only implement the special behavior for the concrete player.

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

The system will use a template method pattern to generate all the objects in the floor. The enemies, except for the dragon, are generated based on their generating possibilities. The method for generating different enemies is implemented separately. The advantage is to make it easier to add more enemies (i.e. large number and more types), since we add a new method to generate new types and enemies. The dragon is generated when the dragon hoard is created. The player is created in a same template method, but implemented separately, since there is only one player and player is chosen based on the input. Therefore, it is different to generate enemies and player.

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

We used visitor pattern to reveal the abilities of both enemy and players during the combat. The Player and Enemy superclass has a general be_visit method for all players and enemies, which gives the most general behavior for them. Since each enemy and player have different abilities when encountering with different character, we can use visitor pattern to dispatch the actual scenario that we have to deal with. To do this, we implement the specific behavior for individual concrete player and enemy in order to override the virtual method(be_visit). As for abilities like potion effect, gaining health etc., we can override the methods differently according to different characters. we will use the same techniques as for the player character. Since they are both characters, we use similar techniques in some ways(i.e. override the virtual methods)

Question. 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.

Decorator pattern works better. The advantage of Decorator pattern is that we can easily stack the effect of Postions, and we can easily reset the atk and def when the player reach another floor. The disadvantage is that we need to be careful when modify the hp of the player, since the hp is changed permanently. The advantage of Strategy pattern is that we can easily apply each potion to the player. The disadvantage of Strategy pattern is that we cannot easily reset the atk and def when going into the next floor.

Question. 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?

We have a template method to generate both treasure and potion. When generating them, we use some helper function to randomly generate the different potions and treasures. Since they are inherited from item, we can reuse the helper function to generate them.

Extra Credit Features

we managed to complete WASD control, Enemy chasing, Add some more characters and enemies, add colorful display, alternating day and night every 20 moves, and Random Floor Generation.

1. Wasd control: we use initscr(), getch(), mvprintw(), mvaddstr(), clear(), move(), refresh(), endwin() functions inside the curses.h to achieve this. The user can use keyboard to interact with our game.

2. Enemy chasing: we implemented one method to find the distance between two objects, and an extra fields called chamber_num. Every time when the player walks into a new chamber, the corresponding chamber number is assigned to the player. Then all the enemies in the same chamber would start chasing the player. (Except the merchant, which only chase the player when it's under revenge mode.

3. More characters and enemies: this part, we simply add more classes into our program, and implement each characters with different features, which takes a lot of time but easy to achieve.

4. Colorful display: after we explore the curses.h library, we discovered that, it also provides the ability to display color. So we used init_color, start_color(), attron(), attroff(), COLOR_PAIR etc.

5. Alternating day and night: after we have the color, we add day and night to add more fun to the game. This is achieved by implemented two different display color, and switch between them every 20 turns, some extra character have special ability when the night comes.

6. Random floor generation: we think of a method to modified the given map, which provided us new random map. Also, we have a creative way to read chambers. After reading the floor file, we generate a random coordinate, if the point falls inside the tile inside the unread chamber, then we use recursion to start expending the read in chamber vector until it hits the wall or the read tile.

Final Questions

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?

First of all, it is necessary to have a plan when developing software in teams. In the plan, we can break down the whole project into small pieces, and allocate each part of the project to other teammates, which will remind us to make some progress constantly. Before developing software, every teammates should know well about what the project is going to be and what requirements we should achieve. The first step is to design a UML for the program, which will give us a rough overview of the program. In UML, the team should discuss about various design patterns that are appropriate to use and the proper relationships between classes. The preparation of developing a software, we think, is so important because we are able to successfully develop a program only if we are in a right track. In addition, during development, it is quite necessary to make the program display first. The reason is that every team members cannot test the program until the program can display something.

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

We will think of a more efficient method that eliminates as many fields as possible, By using more pattern. Also, if we have another chance, we'll pass the board to the player, enemies, so that we can make more changes to under the subclass.