Final Design Document

Chamber Crawler 3000

Trinity Guo

Levana Li

Susan Shi

INTRODUCTION

This documentation is for our final group project ChamberCrawler3000(CC3k). We are a group of three: Trinity Guo (j283guo), Levana Li (x969li), Susan Shi (sqshi). We have implemented a simplified rogue-like game CC3k with the required core functionalities.

OVERVIEW

At the start of each CC3K game, we read in a floor plan text file, and store it in a *Floor*. We then ask the player to choose their favorite race, and generate the corresponding *PC* on the floor. After this, enemies and items with random races and types are randomly spawned on the floor, except for *Dragon* which only spawns next to a *DragonHoard* and *Stair* which must not be in the same chamber as the *PC*.

To implement this turn-based game, we used the MVC design pattern to increase cohesion and decrease coupling. User command will be interpreted by the main function and then sent to the Floor class for action. The result will be displayed using FloorDisplay.

The turn-based movements are ordered as following: the player always moves first, then the enemies move in “line-by-line” fashion. Note that for each turn, all the enemies on the current floor are stored in a vector and sorted by their x,y coordinates. In this way, we can simply implement the “line-by-line” fashion by iterating through the vector.

We have implemented PC and Enemy using the inheritance design. For instance both of them are under the same parent class Characters, such that they can use the basic accessors and mutators such as getHP() and setHP(). Under PC and Enemy, we use specialization, such that we can add new races with abilities easily.

We also used the visitor pattern to implement the combat system, for instance, different combinations of PC race and Enemy race have different special effects. A visitor pattern helps to avoid long and repetitive if conditions, and makes the code clearer.

In addition, the program uses ncurses to allow users to make movement commands directly through the keyboard.

DESIGN

Main:

- Our main function acts as our controller in the program. It will take in a .txt file provided by either command line arguments or the default text file. It will create a new game when the user starts (or restart) the program, and receives user commands and translates it to the actions that will be performed by the player character on the floor.

Floor:

- Our Floor class is an important part of the program. It acts as the model and manages user commands. Floor consists of exactly 5 chambers and stores them in a vector. Floor generates PC and game objects randomly: PC, stair, potion, treasure, and then enemies.
- It has a function called PCAction which handles the user's command translated by the main function.
    - If a player chooses to move, the floor checks if the movement is valid. (i.e. player not moving towards an occupied tile that is not gold, player is not moving to a wall) Then the player's position changes and the cell it was on will no longer be occupied. The cell player moved to will be occupied.
        - If player moves to a stair, a new floor is generated.
    - If the player chooses to use potion, the floor will check if there is a potion in the designated direction and let pc use that specific type of potion.
        - The potion will then be deleted from its chamber and the tile it is on will no longer be occupied.
    - If the player chooses to attack an enemy, the floor will check if there is an enemy in the designated direction and let pc attack.
    - After the player makes their move, the floor will calculate enemy death.
    - Dead enemies will be removed from the chamber and its tile will no longer be occupied. Player may gain gold or gold may be spawned based on the Enemy race.
    - If the Enemies are in freeze mode, then they will not make actions, else the floor allows enemies in the player's chamber to perform their action. Player death is calculated after all enemies have made actions.

Chamber:

- A chamber consists of many cells, enemies, potions, treasure, and occasionally a stair.
- When a new floor is randomly generated, each chamber also resets to its original state.
- When new enemies/ items/ stairs/PC are generated, they occupy a tile in a specific chamber.

Object:

- Object is the superclass for all game objects. It records the row and column number and the type of the object.

Item:

- Item is a subclass of Object with no additional fields or functions.

Treasure:

- Treasure is a subclass of Item.
- It has an additional private field named amount and a virtual function pickup.
- Treasure is implemented using the Strategy design pattern.

Potion:

- Potion is a subclass of Item.
- It has an additional virtual function use.
- Potion is implemented using the Strategy design pattern.
- When the PC moves on to the next floor, all temporary potion effects are removed by resetting PC status. (done in Floor)

Characters:

- Characters is a subclass of Object. It has additional fields: HP, atk, def, MaxHP, initAtk, initDef and dead.
- It has additional virtual functions allowing a Character to move (change position) ;attack, or attackedBy another Character.

Enemy:

- Enemy is a subclass of Characters.
- It overrides virtual functions in Character and has a function called makeAction.
- This function works as the following:
  - Unless the enemy is an untriggered Dragon or a hostile Merchant, it will attack if the PC is within 1 block radius.
  - else unless the enemy is a Dragon, it will move.
- All different races of Enemy characters are subclasses inherited from Enemy. They may override the attack/ attackedBy functions depending on their race.

Player:

- Enemy is a subclass of Characters.
- It overrides virtual functions in Character
- All different races of Player characters are subclasses inherited from Player. They may override the attack/ attackedBy functions depending on their race.

Cell:

- Cell can be classified into many types.
- only Wall and Stair cannot be occupied because a player cannot stand on a wall, and if the player reaches the stair, it goes to the next level.
- Enemies and items can only occupy tiles.
- At the initial state of the game, all cells are not occupied.

FloorDisplay:

- Our FloorDisplay class manages the interface to present data.
- It has a Floor pointer and uses the ncurses library to allow WASD controls.
- It will print the current floor after each player action.
- It will also display corresponding messages based on player action and results.

RESILIENCE TO CHANGE

We designed our program such that all races inherit the class Characters. If we need to include a new race for either the player or enemies, the change can be easily made by adding a new subclass with overridden functions. In addition, if we want to give players other special powers, we can directly add the corresponding functions to the Player class.

Furthermore, we used the Strategy design pattern for Potions and Treasures, if we need to include more types of potion or more types of treasure, we can do so by adding more subclasses to Item.

We can also change the probability of spawning different objects by changing the range for random numbers in class Floor. This can also provide us the ability to modify the difficulty of the game by increasing the number of enemies or decreasing the number of enemies per floor.

If we wish to provide more command options such as freeze all enemies, we can easily do so by modifying our main function or our PCAction function.

Since we are using MCV in our project, we were able to maintain a high cohesion and low coupling in our program. The functions in each class are designed to perform one task that is related to itself. For example, in the class FloorDisplay, the functions are only there for the class to display the floor layout. And in the class Floor, the functions are only there to operate actions performed on the floor. Although there is a certain degree of coupling between certain classes due to dependency and aggregation, the use of MCV still decreases the degree of coupling. Hence making our project resilient to change.

ANSWERS TO QUESTIONS

Q1. 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?

- Structure the code using inheritance, more specifically using specialization. We will have a parent class called Characters that has PC and Enemy as its subclasses. PC would be a parent class which contains all the choices for player races as child classes. Similarly, the Enemy class will have all enemy types as its child classes.
- In this way, new races can be easily generated by simply adding a new child class to the PC or Enemy class and implement its initial states (such as HP, Atk, Def and Symbol) as needed in the constructor.

- Create corresponding race based on the user input/ generation requirements by calling constructors

Q2. How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

- Different, because there is a mutual attack relationship between enemies and the PC. We would use a visitor pattern to implement the attack effects, so we must be able to distinguish between an enemy and the PC.
- Although the structure for the Enemy class is similar to our PC class, the two parent classes should be strictly separated in order for us to manage our code readability when implementing the visitor pattern.
- Also, when we initialize Enemy and PC on the floor, we have different specifications, for instance, Enemy is bound to its spawned chamber and it doesn’t interact with other items. Also because multiple Enemy are spawned on each floor, we could implement using the Factory method.
- In addition, the race of enemies are based on only probabilities but the player character could be generated based on the race the user chooses.

Q3. How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

- Although we did not build separate concrete visitor classes for the enemy and player respectively. We applied the visitor pattern such that the attributes of different characters would change accordingly as they attack or are attacked. The damage on that character would be different if attacked by different race because of method overriding, that is, the specific method would be invoked as the parameter (subclass of Enemy) is passed in.

Q4. 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.

- The advantage of Decorator is the ability to add functionality or features to an object at runtime, and we want to temporarily increase/decrease the PC's Atk and Def without changing the PC object. The disadvantage of using decorator is that when the PC moves to another floor, we have to reverse the effects of some potions while keeping the effect of RH and PH.
- But in the end we used the Strategy pattern, and we uses a reset

Q5. 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 construct a superclass called Item from which Treasure classes and Potions classes inherit.
- Since both Treasure and Potions are generated at random, when the floor is generating items, it needs to make sure the chosen cell is an unoccupied tile. Such a method is implemented as a private function in the Floor class. When Items are being generated, they will call this helper function to make sure the item can be generated on a cell and spawns it. Treasure and Potion both use this method to avoid repeating code.

EXTRA CREDIT FEATURES

We used a curses library to make this like an actual game with WASD controls. So our display is not in standard output but rather in a new window. Instead of printing a 2D vector to standard output, we decided to make use of the curses library. This was challenging because we need to know where each game object is located on the 2D vector and display its corresponding symbol on the screen.

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?

   During this assignment, we learned many valuable lessons about developing a large project in a team. It is very important to start the assignment early and clearly distribute the work to each individual member. Communication is vital when it comes to working as a group since all team members have their own habits of naming variables. So we need to make sure our teammates understand what each name represents to avoid confusion. In addition, since we all have unique ways of thinking, we need to communicate our plans thoroughly as we progress. This can avoid many confusions such as the misuse of private fields. Debugging also requires a lot of communication since each one of us wrote our codes separately, so when we are debugging the entire program, we need to fully understand each other's code. We also learned that working as a group can be more efficient since we can discuss our ideas and concerns together. The main lesson we learned is the need for an extremely detailed plan. Sticking to the plan helped us to be more efficient and kept us on track.

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

   If we had a chance to start over, we would develop and debug step by step. We started debugging once we finished every component of the program, but then we have many bugs to fix and it was hard to figure out which part went wrong. So it would be better if we made sure each stage of the program is working and then continue on to the next stage. In addition, we would probably try to make use of other design patterns such as Model-View-Controller and Visitor to handle this assignment. If we did these things then we would probably have enough time to add other bonus features and minimize code reusability.

   The most important thing is, we should start from the most fundamental functions. We planned too much at the beginning and later found out we are not able to put them all together. It is also crucial to communicate with teammates frequently and have comments on all codes.