Xinhao Xu
Zian Wang
Joyce Dai

CS 246 Final Project-Design
CC3K

Introduction

Our game is called ChamberCrawler3000(CC3k), which is a simplified rogue-like game. It can be run with or without an input file that specifies its map. CC3k's interface is made up of a checkerboard 79 columns wide and 30 rows high (5 rows reserved for displaying information). The game is played as follows: players can choose from different characters (Shade, Drow, Vampire, Troll, and Goblin), each with their own attributes (HP, ATK, DEF). The player has to walk through a dungeon, kill enemies and collect treasure until reaching the end of the dungeon (the end of the dungeon is the 5th floor). The dungeons are made up of different floors with rooms connected to passageways. When players choose their characters, they will appear on the first floor of the dungeon (each floor has 5 chambers), they need to walk through different chambers and reach the stairs leading to the next floor, during which they will encounter different enemies (Human, Dwarf, Elf, Orcs, Merchant, Halfling and Dragon). Each enemy also has its own stats (HP, ATK, DEF) like the player. The players will also encounter different potions that enhance or weaken their own attributes (RH, BA, BD, PH, WA, WD). When they reach the end of the fifth floor of the dungeon, the game is won.

Overview

We have implemented the following classes:

- Cell
- Character
- Player (Shade, Drow, Vampire, Troll, Goblin)
- Enemy (Human, Dwarf, Elf, Orcs, Merchant, Halfling)
- Dragon
- CharacterCreator
- Item (Potion, Gold, Stair)
- ItemFactory
- Floor

The detailed relationships between the above classes can also be found in the uml-final.pdf file.

To initiate the game, the floor class will create 25x79 cells by reading in an empty map (empty_map.txt) if a map is not provided and storing the corresponding characters in each cell. Then, it will spawn all the enemies, gold, potion, player, and staircase objects, creating their shared pointers and storing them within the class. Each of the player, enemy characters, potion, gold, and staircase items will all contain an x and y coordinate to indicate their location. When a character or item is spawned, it will modify the cell's character based on its x and y coordinates to reflect its position for the graphics rendering.

Xinhao Xu
Zian Wang
Joyce Dai

A game loop is implemented in main.cc and for each loop, after the player enters a command and the corresponding function is called, it will tell the floor to move the enemies and let the enemies attack. Enemies will not attack if a player has just been spawned and will not move if given the command f.

The floor class is responsible for calling the player to move, attack, and use potions when the proper command is entered. The specific implementation for the above functions is implemented in the player and its children classes. The floor will also call enemies to move randomly and initiate the enemies to attack the player when conditions apply. Overall, it is the main class that controls the flow of the game.

The player and enemy classes inherit from the character class and are the parent classes for the player and enemy races. Their special abilities for moving, attacking, and using potions are implemented by overriding functions in player and enemy classes.
When a command is entered, if an item is used, it is removed from the floor and its corresponding cell location will become empty, devoted by '.'.

As the floor class contains its current floor number, when the floor number exceeds 5, then the game is won and a winning message is displayed. The floor also contains a shared pointer of the player to check whether the player's hp is above 0. If not, the game is lost and a losing message is displayed.

Design

The below classes are designed as follows:

- Cell
  A cell represents a 1x1 square on the map of the game. Each cell contains a character that represents what the cell contains. The notation follows: '@'-player, '\'-stair, '|'-wall, '–'-wall, '+'-doorways, '#'-passages, '.'-floor tiles, W-Dwarf, E-Elf, M-Merchant, L-Halfling, H-Human, O-Orc, D-Dragon.

- Character
  The Character class stores all the stats of a character - their hp, attack, defense, max_hp, race, current location, potion effects, and action message. It has four virtual functions that manage the attack and death behaviors of the characters.

- Player (Shade, Drow, Vampire, Troll, Goblin)
  The Player class inherits from the Character class and keeps track of the player's previous location symbol, the gold that is collected, and the chamber that it spawned in. It is responsible for the player's move and use potion behaviors. It also has two virtual functions that define the potions effects and the move effects for players that have special

abilities. All Shade, Drow, Vampire, Troll, and Goblin classes inherit from the Player class. Each of their special abilities is implemented by overriding the virtual functions from their parent classes.

- CharacterCreator
  The class CharacterCreator is responsible for generating a character object when it's given a name, x and y location, chamber number, and optional gold location if it needs to create a dragon. It returns a corresponding shared pointer.

- Enemy (Human, Dwarf, Elf, Orcs, Merchant, Halfling)
  Similar to the Player class, the Enemy class also inherits from the Character class. All Human, Dwarf, Elf, Orcs, Merchant, and Halfling classes inherit from the Enemy class. Each of their special abilities for an attack is implemented by overriding the virtual functions of their parent classes.

- Dragon
  The Dragon Class inherits from the Enemy class. A dragon is only spawned when a dragon hoard is generated. The dragon will always guard the dragon hoard on the right.

- Item (Potion, Gold, Stair)
  The item class stores the x and y coordinates of the item, its effect type, which is a string of their names, effect value, and whether it can be picked up or not. All Potion, Gold, and Stair classes inherit from the Item class. Gold's effect value reflects what kind of gold it is. The value is 2 for normal gold, 1 for small gold, 4 for merchant hoard, and 6 for dragon hoard. The different types of potions have their own classes that inherit from the Potion class. Their effect type is what the potion type is (RH, BA, BD, PH, WA, WD) and their effect value reflects their positive and negative effects(-5, -10, 5, 10).

- ItemFactory
  The class ItemFactory is responsible for generating an item object when it's given a name, x and y location, and optional effect value and pickup boolean for gold. It returns a corresponding shared pointer for the generated item.

- Floor
  When the game is initialized, it will initialize a floor by creating 25x79 cells that reflect what it contains. The floor contains vectors of all enemies, dragons, items, and a shared pointer of the player that are spawned on the current floor. It also contains a vector of string to keep track of the potion types that the player has already used. The Floor class is responsible for initiating the player attack and calling enemies to attack the player when it's within one block radius of any enemy.

Xinhao Xu
Zian Wang
Joyce Dai

How the player and enemy attack features are implemented:

- The attack feature of this game is implemented by using a visitor pattern. The attacker will visit the victim (attack_to(Character& c)), then the victim will calculate the damage caused and modify its hp (attacked_by(Character& c)). These functions are virtual functions in the Character class. For characters that have special effects when being attacked, such as how orcs do 50% more damage to goblins, we will override the attack_to and attacked_by functions of the corresponding classes for their races. The attacked_by function also contains an on_death() function that returns the amount of gold generated when the character dies. It is also a virtual function in the character class and is overridden for enemies, such as Human and Merchant, since they drop gold when they are slain. When an enemy dies, it will notify the floor class to remove the stored pointer and change the character store in the cell. If a Human or a Merchant dies, then a pointer to a gold object is generated, stored, and the cell is also changed.

How the use potion method is implemented:

- The potion feature of this game is implemented by using a factory method pattern. Each potion is generated with a type and effect value. The floor stores a vector of strings that keeps track of the types of potions used on the current floor. When the player uses a potion, it will modify the vector of items and remove the potion from it. It will also modify the cell that corresponds to the potion's location. The effects of the potion can be obtained by the type and effect value and it is implemented in a virtual function named potion_effects(string type, int effect). The corresponding effect is added to temp_atk or temp_def attributes of the player. To remove the potion effects when the player enters a new floor, these two values will be set back to 0 and the vector of used potions will also be cleared. For dwarfs, who can maximize the potion effects by 1.5, it overrides the potion_effects function and returns the maximized effect in its class.

How to implement potion effect on characters:

- We utilized the decorator pattern to implement the potion effect on characters. for each concrete class (eg, WA, WD …) inherited from PotionDecorator. We can override the methods of the getters of Character class to modify the attributes shown to the client when the getter method is called. The decorator by design is easy for the client to implement since when adding/deleting a potion, we only need to add/delete a linked list node of different type of potion class and all the effects to the character will be automatically adjusted.

Xinhao Xu
Zian Wang
Joyce Dai

Resilience to Change

Our program is designed that can be adaptable to many changes as described below:
The character races along with their special abilities can be easily added, modified, or removed.

- By utilizing the power of polymorphism and inheritance, it is fairly easy to add/delete a new type of character by simply creating a new subclass of Player/Enemy. The newly added characters can have their own attribute initialization and the attacking behaviors can be overwritten by the virtual method attack_to() or attack_by(). The client only needs to do some small changes when using the CharacterCreator class and the program is still able to function with minimum tweaks and new characters added/deleted.

  As we utilized the factory pattern to help us create Character with ease. On the client-side (class Floor), we don't have to change the way we create new characters if we decide to add or delete specific types of characters since all the creation of characters is taken care of by the CharacterFactory class. This also reduces coupling since class Floor only needs to include the CharacterCreator class instead of including all the child classes of the Character class in order to use their constructor functions.

The types of items with different effects can also be easily added, modified, or removed.

- For example, if we wish to implement a weapons system, and we wish to generate weapons randomly on the map, we do not need to change any code that was previously written. This is because similar to creating characters, we also used factory patterns to help to create Items. All the creation of the different types of items is taken care of by the ItemFactory class. This further reduces coupling, for the same reasons as the character creation process.

A graphical display can be implemented without affecting the program that is already written.

- As we used the observer pattern to notify the rendering method of any changes in the game, then if we want to change the command line display to a graphical display in the future, it can be done with ease. We can easily change from notifying each cell of the cell array to each pixel of the canvas in the new graphical rendering implementation without affecting the game logic in the floor class , making our program more adaptive to different display methods.

In our floor class, we separate the interactive behaviors from the graphics rendering. We use a vector of cells to store the current image of the game, and the class RenderGraphics is responsible for rendering the image and text display. For all other functions that are declared on the class floor, They all serve the purpose of performing game logic corresponding to a command given by the user. We also put the responsibility of altering the attributes of characters to the character class itself. So instead of the floor directly changing the attributes of the characters, characters are responsible to manage their own attributes. All those design decisions are examples of high cohesion.

By all those conscious decisions we have made. It is very easy for us to add new characters or item types, change what type of symbols can represent different objects, and the interaction between different characters. This makes our program more modular and maintainable.

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 races?

For each race to be easily generated, we use inheritance to create an abstract base character class, then create an abstract player class that inherits from the character class, then each race will be a concrete class that inherits from the player class. Each time when a floor is generated, a player pointer will be passed into its constructor, so that the status of the player character can be remembered for all 5 floors. With this strategy, it will be fairly easy to add additional races. It can be simply done by creating another class that inherits from the player class with its own race features and attributes.

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

To generate different enemies, we create an abstract enemy class that inherits from the character class, then for each type of enemy, they can be implemented as a concrete class that inherits from the enemy class. The way that we generate the enemies is the same as how we generate the player character since both strategies use inheritance.

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.

To implement the various abilities for the enemy character, we store each of their particular features and attributes in their concrete classes. Then, we can implement a Visitor pattern. When a player is attacked by an enemy, the enemy's pointer will be dereferenced and passed to the character that is being attacked, to specify its effects. We are going to use the same technique to implement the various abilities for the player character races. This is because the player will also attack or be attacked by the enemies, just as how the enemies will attack or be attacked by the player. Therefore, both the enemy and the player character's abilities can be implemented by a Visitor pattern.

Xinhao Xu
Zian Wang
Joyce Dai

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 discussing the advantages/disadvantages of the two patterns.
The advantages of the Decorator pattern are that it enhances the extensibility of an object by allowing the program to add additional features to an object at run-time rather than to the class, and it increases the flexibility of the program as any functionalities and features can be withdrawn. This is particularly useful to model the different effects of the potions and to remove their effects when the player character reaches the next floor. Its behavior also allows us to modify any effects of the potions without having to change an entire class, so it makes coding and debugging easier. The disadvantage of the Decorator pattern is that it may result in much repetitive code in different classes, making the code harder to maintain. The advantage of the Strategy pattern is that it allows the program to interchange between different algorithms at run-time, allowing the algorithm to vary independently of the client. Each potion's different effect can be implemented as different algorithms, and the program can select these different algorithms when a player uses the potion. However, the disadvantage of the Strategy pattern is that how each algorithm is selected has to be specified, and the algorithm is only called under certain circumstances. Since the Decorator pattern allows us to freely add features and remove them, while the Strategy pattern may result in having to write many different algorithms, we believe that the Decorator pattern is better 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.

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 use the Factory Method pattern to generate Treasure and Potions to avoid duplicate code. First, we create an item class and ItemFactory class since we are using the Factory Method. The two classes could help us to determine what type of items are generated and what effect it has(if it is a potion)/what value it is(if it is gold). Then, we create concrete classes for Gold and Potion and override the above methods with specific setups. For example, when the ItemFactory class determines the item is the gold, and when a merchant or a dragon is killed, a Treasure will be generated or can be picked up respectively. When the ItemFactory class determines the item is the potion, it would be recorded whether the potion has been used or not and pass the effect value to the player.

Extra Credit Features

Our program did not use any "new" or "delete", all of our pointers are smart pointers. By doing so, we can ensure that no memory error or memory leak will occur. When we needed to use or store a pointer, we made sure that we used make_shared instead of "new".

Xinhao Xu
Zian Wang
Joyce Dai

Final Questions

Question1: 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?

When developing software in teams, the key point is not only on developing good software but also on teamwork. A good team requires a high degree of cooperation and coordination among each team member. First, it is very important to discuss how to build and manage the program, which will give us a general outline of the whole program. We can build the framework of our program through UML diagrams, which can express the dynamic and static information in software design. This can help the team to understand the functional requirements of the system in a visual way. Secondly, we need to draw up a timetable so that we can better divide our work and know the current completion of the program, just like the plan required in ddl1 In addition, developing in a team can cultivate the ability to communicate and understand among team members to a great extent because each team member needs to communicate with each other and adapt to the ideas of others.

When working alone, we need to follow OOP design principles, which can help us complete and debug our programs more effectively to a great extent, which helps us save a lot of time.

Question2: What would you have done differently if you had the chance to start over?

In the design phase of our project, we should have the idea of MVC in mind. Instead of implementing a large part of the game logic in the character classes or the item classes, we should leave the control of the game flow to the floor class, separating "C" (Control) from "M"(Model). We would also consider using a hashmap to store all our character and item objects on the floor. This way, we wouldn't need to loop through the vector to find the corresponding object at a location and therefore we can enhance the efficiency of our program.