FINAL PROJECT

Accomplished

Project Name: CC3K

2020-08-15

Table of Contents

Introduction

ChamberCrawler3000, cc3k, is a game that consists of five floors, where in each floor, there are five chambers. The player needs to slay enemies and gain treasures until it reaches the fifth floor. Any effects that potions given to the player will disappear when the play enters the next floor.
A brief description of the project structure is provided in the **Overview** section.
Throughout the implementation, more methods and fields are added to achieve the required functionality. Slight changes in program design are detailed described in the **Changes in Program Design** section.
We implement distinct classes and several important methods to accomplish the game process in high cohesion and low coupling way. This is discussed in the **Design** section.
Several Design Patterns are implemented to easily accommodate the changes to the features of the project. This is explained in the **Resilience to Changes** section.
All answers to questions are in the **Answers to Questions** and **Final Questions** sections.

Overview

- This project is designed using Object-oriented design and implemented using C++ languages.

Project Breakdown

- Interface for classes of Floor, Chamber, Character, Player, Enemy, Item, Treasure, and Gold
- Implementation for classes
  - Create all Player classes and Enemy classes
  - Create Item classes and Floor classes
  - Implement movement of Player and Enemy
  - Implement the interaction between the Player and Enemy
  - Implement the interaction between the Player and Item
  - Implement main to read inputs from the users

Changes in Program Design

The class relationship for final design is mostly the same as the original plan; However, we changed some fields and methods in each class for convenience.

Note that a 2D vector, ***grid***, is added to the field of the Floor Class, where its reference is called in Chamber Class. Any modification of Character will be implemented in Floor Class to increase the cohesion; therefore, we move the methods that generate elements like Player, Enemy, Potion, Treasure, and Stair to the Floor Class. As the Floor Class is the main controller of this game, ***erase[Treasure/Potion/Enemy]*** methods have been moved from the Chamber Class to the Floor Class for clearer and easier management. We add ***playerAttack*** and ***enemyAttackPlayer*** in the Floor class to achieve combat; ***enemyMove*** is added to achieve random move of enemies that are not in a combat; ***PlayerMove*** is implemented for the movement of the Player after user input, it also can control the Player to pick gold. ***Readfromfile*** is implemented so that player can choose to read in the floors from the file player provided.

The shared_ptr of Potion in Player Class is changed to a vector of shared_ptr of Potion to indicate all effects that a Player has instead of referring only one single effect.

The health, attack, and defense field are changed to ***double*** from ***int***, because we noticed that the Potion effects for Drow are magnified by 1.5, so the attack and defense might not be integer after using some potions. Besides, Orcs does 50% more damage to Goblins, so the damage might not be an integer. We change the field ***gold*** from shared_ptr of Gold to integer, because we only need to know the value of golds that the player owns.

Design

We have three main classes, Floor, Character, and Item, where each has several derived classes.

Class of Floor

A class of Chamber will be implemented as the derived class of Floor, in order to separate the Floor into five parts. Chamber, Enemy, Dragon Hoard, and Potion are in composition with Floor. We will erase them when the Floor is exited and generate new Enemies, Chamber, Dragon Hoard, and Potion when a new floor is re- constructed.

Important Methods in Floor Class

generateAll(shared_ptr<Player>): There are six distinct methods ***generate[something]*** (i.e. something refers to Chambers, Player, Potion, Treasure, Enemy, Stair), each will generate one particular type as many times as that we need. They are called in ***generateAll(Player)*** to generate all certain types that a floor needs. Any modification of amount of types will be done in this method so as to increase cohesion and convenience. We can modify the codes only in this method to add or eliminate any features.

enemyAttackPlayer(): If certain Player is within one block of Enemy's attacking range, it will attack the Player. We use this method to implement the combat by calling the overwritten ***accept(Enemy&)*** in the Enemy class in order to achieve low coupling. This will be discussed more in Enemy class.

enemyMove(): This method is implemented to ensure the Enemy who has the player nearby will not move while all other Enemies move randomly.

playerAttack(str): This method enables Player to determine whether the Enemy is within its attacking range. If there is an Enemy that can be attacked, the Player will attack it then the corresponding Enemy will fight back. If the Enemy is killed, then corresponding reactions such as dropping Merchant Hoard or Gold that depends on each type of Enemy is shown. Player will pick up them when walking over.

playerUsePotion(str): This method is implemented to determine if the direction that players want to use potion exist potion. If potion exists, and it's a permanent potion, player will use the potion and the effects will be displayed immediately; if potion exists and it's a temporary potion, it will be added to the player's temporary potion vector, so that the effects will be undecorated when player moves to next floor, and the effects will be displayed immediately as well.

Note: usePotion(Potion) and ***addPotion(Potion)*** in Player's class to add corresponding effects to the player. Low coupling is accomplished as the implementation is actually done in the Player Class where the Floor Class is the controller of this action.

clearFloor(): Clears all elements on the current floor, including Enemy, Chamber, Potion, and Treasure.

<u>Important Methods in Chamber Class</u>

Validpos(int, int): This method is to ensure whether this position is valid for elements, that is, not in the stairs or walls.

generateRand(char): This method is to randomly generate parameter c, determine whether c is in the valid position by calling ***Validpos(int, int)*** and return c's position.

Class of Character

Two separate derived classes, Player and Enemy, each shared the common fields from the base class Character. Shade, Drow, Vampire, Troll and Goblin are implemented as the subclasses of Player, while Human, Dwarf, Elf, Orc, Merchant. Halfling, Dragon are implemented as the subclasses of Enemy. Note that Dragon is a special class that is in aggregation with the Dragon Hoard from the Treasure class. Visitor Pattern is used in combat between the Player and the Enemy. Factory Pattern is implemented to generate Enemies. These design patterns will be discussed in detail in **Resilience to Change** section.

Important Methods in Player Class

attackEnemy(): This method is overwritten in each particular Player Class (i.e. Vampire, Drow, Troll, Goblin, and Shade) to show the detailed information when combating with Enemy. If the attack is missed from the Player to the Enemy, a message will be displayed. It will also display the messages of the Type of Enemy, the specific damage that is received from the Enemy, and Enemy's current HP.

usePotion(shared_ptr<Potion>): This method is implemented to show the special effects that the Player gained when using the Potion.

addpotion(): This method is created for temporary Potion effects which will be erased from the Player when entering the next floor.

removeTempPotion(): This method is used to remove temporary Potion effects when the Player entering the next floor.

Important Methods in Enemy Class

attackPlayer(Enemy&): This method in called in Floor Class to implement the combat between the Enemy and the Player. It is overwritten to distinguish the different effects that appear when attacking certain Players.

Class of Item

Two separate derived classes, Potion and Treasure, each shared the common fields form the base class Item. Potion has different types of effects; each will be recognized by the string Type. A Potion pointer is added in the Player class, in order to manage the effects that has been added to the player. Once the Player enters the next floor, the Potion pointer will become null to ensure those effects are limited to the floor they are used on. Dragon Hoard and Gold are derived classes of its base class Treasure. The Dragon class contains a pointer of Dragon Hoard.

Resilience to Changes

As mentioned in the **Changes in Program Design** section, some special fields like ***grid*** and methods like ***attackPlayer(Enemy&)*** are added to the original plan for high cohesion and low coupling purpose. These new implementations improve the efficiency of the program and create flexibility when adding new features.

As we separate the elements into different classes, adding features will be easy if we can define which class it should be in. For example, if we want to add any Player or Enemy, few codes need to be added to the derived class Enemy. We do not need to add any fields to enable this functioning as Enemies and Players share common fields in Character Class. For any other features that works on all the classes, we can implement it only in ***player.cc*** instead of writing the repeated codes in other classes. In this way, our design with new methods and fields (mentioned in **Changes in Program Design** section) fulfills the goal of maximizing cohesion and minimizing coupling, and also create flexibility to any changes.

The following Design Patterns are implemented in the design of the program.

Visitor Pattern is used in combat between the Player and the Enemy. Since most of the abilities for Enemy needs to be shown in a combat, Visitor Pattern is suitable to determine when to use and effects of using these abilities. Some enemy characters have special abilities when they interact with some specific player characters, 5 virtual attack methods are implemented in the abstract Enemy class, each corresponding to a Player Character. Every derived Enemy class will override all these methods. In Player class, every derived class needs an ***accept(Enemy&)*** method to implement the effects brought by the abilities from the enemy. Whenever we want to add new effects or features that are received from the Enemy, implementation will be in the ***accept(Enemy&)*** method. In this way, high cohesion is shown.

Factory Pattern will be implemented to generate Enemies. An abstract ***Type*** class is generated, with the pure virtual method ***getEnemy()*** and ***setType(char)***. In the derived ***concreteType*** class, we override these two methods. Depending on what type of enemy we want to create, the ***Type*** will be set to the notation of that enemy. For instance, if we want to create a Dwarf, W will be set as a ***Type***. Calling ***getEnemy()*** will produce the corresponding enemy according to the ***Type*** we set. The ***Type*** can be changed by ***setType()*** whenever we want to create a different enemy.

Answers to Questions

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

   Answer: Each race will be inherited from the abstract Player class, so that all races can share the common field and common methods. To obtain a different variety of each race, methods of each derived class that inherited from the base class will be overridden. Each race will be easily generated in this way. Additional races will be added easily in the same way.

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

   Answer: We will use factory design pattern to generate different enemies. An abstract ***Type*** class is generated with virtual methods. These methods will be overridden in the ***concreteType*** class. Depending on what type of enemy we want to create, the ***Type*** will be set to the notation of that Enemy. As all Enemies share the common fields of the Abstract Enemy class, and all Players share the common field of the Abstract Player class, generating enemies can be almost the same way as generating players. However, the difference is that unlike players, which generate only one race, 20 enemies will be generated per floor. Since we need to generate multiple enemies in the game, it will be better to randomly generate enemies and tell the factory what enemies we want to generate. In this way, loose coupling and segregation of responsibilities will be achieved.

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

   Answer: <u>Implement abilities for the enemy character:</u> We will utilize visitor design pattern. Since most of the abilities for enemy characters are to be used when the player and enemies are in a combat, we can use visitor design pattern to determine when to use these abilities and effects of using these abilities. Some enemy characters have special abilities when they interact with some specific player characters, so we decide to implement 5 virtual attack methods in the abstract Enemy class, each corresponding to a player character. Every derived enemy class will override all these methods. In player class, every class needs an **accept()** method to implement the effects brought by the abilities from the enemy.

   <u>Implement abilities for the player character:</u> Each player character has the same abilities no matter

which enemy they are interacting with, so the **attack()** override method will be implemented in every derived player class to make change according to the effect brought by abilities.

4. **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.**

   Answer: Strategy Design Pattern would work better. Decorator pattern would allow us to add features to an object at run-time instead of to the class as a whole. However, it is mainly used to incrementally add multiple features by adding subclasses. Using the Strategy pattern would eliminate conditional statements when we select behavior. On the other hand, it would require all the potions to have subclasses.

5. **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?**

   Answer: We create Treasure and Potions as the derived classes of the abstract class Item. Common fields will be inherited. When generating Potion, the field of string Type is added to recognize the particular effect, and the field ***type(int)*** is added to indicate certain type of effect. For generating Treasure, the field ***amount(int)*** and the ***pick(bool)*** are added.

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?**

   We learned to use Git and Github as it is a distributed version-control system for tracking changes in source code during software development. This system greatly enhances the efficiency when coding together as a team. It is also important to distribute suitable workload to team members in order to exploit each member's strength and avoid weakness. Furthermore, this game is designed using Object-Oriented Programming concepts, which required us to completely master the knowledge and implementation of it. Throughout the planning and implementing, we not only thoroughly understand more of Classes, UML, Inheritance, Design Patterns, but also utilize them in the purpose of maximizing cohesion and minimizing coupling. Moreover, Advanced C++ knowledge is also being implemented into our project. We tried as much as we can to use Smart Pointers, Visitor Pattern, and Factory Method to enhance the code efficiency and avoid memory leak.

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

We would start earlier so that we can implement extra features. More design patterns that can enhance code efficiency could be implemented if we had enough time. Throughout the implementation, we would implement Observer instead of Visitor Pattern since it is more reasonable in this situation. Observer would update Player's appearance to the Enemy and notify the Enemy of Player's position instead of depending on the coordinate system to indicate each element's position. However, since we have already decided to used ***grid***, a 2D vector that stores all Characters, it would cost much more time to implement Observer instead of Visitor Pattern. Therefore, we decided to use Visitor Pattern for this situation. Moreover, most Characters were previously stored in Chamber Class as we planned, however, we found that it can be more flexible to be managed if we stored them in Floor Class. Thus, we moved Characters to Floor Class for achieving high cohesion as most of the important elements were stored in the Floor Class. We would plan more thoroughly next time since it costed sometime to move these elements to other classes.

Conclusion

Throughout the entire project, we learned to work as a team. Each takes responsibility in one's own strength. We fulfilled our goals with a clear design of the game and implemented the knowledge of Object-Oriented design and Design Patterns in this project.