Table of Contents

Final Design

ChamberCrawler3000

Yujia Cao, Kunlin Ran, Yuxuan He

UML of The Final Version

Please see the separate submission file ddl2uml.pdf.

Project Summary

Our Implementation is divided into three parts: Floor, Character and Item. For Floor class, an additional class Chamber is implemented to separate the Floor into five separate parts. Therefore, the characters and items can be randomly generated with equal possibility. It is easier for us to generate Character and Item within range as well. Floor is our main controller of the entire program. All the information of the characters and items will be stored in Floor. Also, Floor is responsible for displaying the contents to the screen.

We have the abstract base class Character containing all the values for characters, it has two derived classes, Enemy and Player, and all enemies and players are implemented as subclasses of them. Same design applies to the Item class.

Designing Details and Classes Breakdown

NOTE: fields and classes are only displayed partly to highlight features.

Floor:

Important Private fields:

- curFloor(int): indicates the number of current floor
- Won(bool): determines the gaming status of the player
- thePlayer (shared_ptr<Player>): stores the information of the player
- theDisplay (vector<vector<char>>): the entire display of the game
- theEnemy(vector <shared_ptr<Enemy>>): stores the information of 20 enemies
- theChamber((vector <shared_ptr<Chamber>>): contains 5 chambers
- thePotion((vector <shared_ptr<Potion>>): stores the information of 10 potions
- theGold((vector <shared_ptr<Gold>>): stores the information of 10 golds
- preTile(char): is the tile that the player previously walked on

Important Public Methods:

- Floor(std::string filename = "map.txt"): constructor, initialize the display of the floor with the input map file.
- setFloor()(void): using methods generatePlayer(), generateChamber(), generatePotion(), generateGold(), generateEnemy() to construct the entire game.
- readFloor(std::string filename) (void): construct the entire game when there is a command line argument for the exact layout of the chambers
- clearFloor()(void): clear the floor before entering the next floor or when the player input reset command
- consumePotion(std::string direction) (void): let the player use the potion
- movePlayer(std::string direction) (void): moves the player, also will call enemyMoveAndAttack() to let enemies reacts to the motion of the player.
- enemyMoveAndAttack()(void): will be called by movePlayer()
- playerAttack(std::string direction) (void): allows Player to attack the enemy.

Chamber:

Important Private Fields:

- Id(int): indicates which chamber the Character/Item is at
- topRow, topCol(int): the starting position of the chamber(left-top corner)
- rowLen,colLen(int): indicates the height and width of the chamber
- theDisplay(vector<vector<char>>&): is the reference of theDisplay in the Floor. Methods in Chamber uses this to mutate the display of the game.

Important Public Methods:

- isValidTile(const int row,const int col) (bool): determines whether a tile is valid or not(is valid if the tile is '.')
- generatePos(const char symbol) (vector<int>): randomly generate the position within chamber to put symbol
- withinRange(const int row, const int col) (bool): check if certain character at row and col are inside the chamber

Item(abstract):

Important Private Fields:

- Row(int)
- Col(int)
- type(string)
- symbol(char)

Gold:

Important Private Fields:

- quantity(int): is the quatity of Gold(1, 2 or 6)
- pickable(bool): determines if the Gold can be picked by the Player

Potion:

Important Private Fields:

- value(int):is the amount of change the Potion can bring to Player

Character(abstract):

Important Private Fields:

- HP(int)
- Atk(int)
- Def(int)
- sym(char)
- race(string)
- row(int)
- col(int)
- maxHP(int)

Important Public Methods:

As every character has different features, important public methods are added and implemented in subclasses. Methods in Character are implemented to access its private fields.

Enemy:

Important Private Fields:

- goldamt(int): indicates the amount of gold the enemy will give when he/she/it is killed, dragon will have 0 as the player will only gain the dragon hoard by killing it
- ishostile(bool): determine whether the enemy is hostile. This is used to check whether the enemy will auto attack the player when they get too close(1 block radius)

Important Public Methods:

- attack(std::shared_ptr<Player> p) (void): allows the enemy to attack the player
- moveEnemy() (void): this will be called by enemyMoveAndAttack() in Floor to react to the movement of the player

Player:

Important Private Fields:

- goldAmt(int): indicates the amount of gold the player has gained
- Action(string): this will show the action of the player and will be output to the screen
- deAtk,deDef(const int): default Atk and default Defence of the player. They are used when the player goes to the next floor and their Atk and Def need to be reset.

Important Public Methods:

- movePos(string direction) (void): allows the player to move to 8 directions around him/her
- usePotion(shared_ptr<Potion> p) (void): allows player to use the potion and to mutate its field. This consumes a Potion pointer so that we will know which potion the player is using
- useGold(shared_ptr<Gold> g) (void): allows player to pick up gold and add to goldAmt. This consumes a Gold pointer so that we will know the value of gold the player is picking
-

Bonus Features

- We used vectors instead of arrays and shared pointers instead of raw pointers in our entire implementation.
- We added different colors to enemies, items and PC when printing to the screen.
- We added the “welcome” screen, “lose” screen, “win” screen and “byebye” screen when

the player just enters the game, loses the game, wins the game and quits the game respectively.

- We added another type of enemy – Midterm (E), and another type of player character – Student, so that when student gets attacked by midterm, the student dies right away. Implementing such feature only needs to type in "dlc" in the command line when running the program.
- We provide three types of command line options, including adding default map, choosing seed(must be greater than 0) to randomly generate character and items, and adding DLC.
  - To add default map: type in valid file name
  - To use seed for srand(), type in any integer greater than 0
  - To use DLC, type in dlc (must be lower case, otherwise unrecognized)

example format for using command line options:

./cc3k <filename> <integer> dlc

Differences Between The First Design And The Final Version

Basically, how we structure our program and the relationships among classes did not change, and we followed what we thought in our first design. The changes lie in specific functions and fields of each class. For example, we added the reference to theDisplay field in Chamber class so that it is easier for us to change theDisplay.

At the same time, we changed all raw pointers into shared pointers so there's less worries about memory leak.

In our original design, we did not consider the command line option, so it can only print the floor in the one file with all five floors identical. In the final version, we support loading specific floor from file provided in command with given layout.

How Do We Accommodate Change?

We used inheritance relationship for items and characters (enemies and players), so that when we want to add new items and characters, we can easily inherit from base class and add its own features. In this way, it's easy to add new items and characters, as well as adding special features for each item or character (actually, the way we added Student and Midterm used such method).

Cohesion & Coupling

Our design tries to fit the requirement of high cohesion and low coupling as much as possible. Functions in each class cooperates to perform one task. (Floor only has methods for operating floors, Enemy only has methods for dealing with enemies, etc.) Every elements in each class is designed to pursue the same goal.

However, certain degree of coupling is unavoidable. Some functions in the Player class will use the pointer of other classes(Enemy, Gold, Potion) in order to mutate the fields in them. As for Dragon, we put the pointer of Gold so that we can find the Gold more efficiently when the Player has interaction with dragon. Also, Player and Enemy class are dependent to satisfy the need of implementing the attack feature.

Q&A in Plan of Attack

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?

A: We will use an abstract base class Character and all different races can be implemented as subclasses. We will override the constructor, destructor and some methods of the base class in subclasses so that we can maintain the variety of each race. In this way, each race can be easily generated, as they will all use the base class constructor. Adding additional classes will be easy as well.

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

A: Generating enemies will be the mostly the same as generating player characters as they all share common field(HP, atk, def, .etc). However, player will be only generated once in one game.Different enemies will be initialized by using rand() function. We will use a base class containing common fields for all characters. All enemies contain the same things except for the Dragon which contains a pointer to the Gold.

Question. How could you implement special abilities for different enemies. For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?

A: Special abilities will be achieved by overriding the attack(player *) method in each enemy subclasses. We can add different features in this method so that player and enemies will be updated differently.

Question. What design pattern could you use to model the effects of temporary potions (Wound/Boost Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?

A: We can use design pattern to model the effects of potions each floor. We can use the decorator class to update the fields of the player every time we want to change it. When the player goes to the next floor, we can regenerate an undecorated Player class to get rid of the effects of temporary potions. But actually we don't use any patterns, instead we just explicitly track the used potions by calling the function findPotion in the floor class.

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?

A: We will use an abstract base class Item containing the common field of treasure and potion (potion, name, symbol,.etc). Treasure and Potion can be implemented as subclasses and different features will be achieved by overriding methods.

Final Q&A

Question. 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?

A: The most important aspects are coherence and communication. We discussed how to implement the program initially, and divided the work, so we must follow what we've discussed. In this way, it can avoid many problems, such as compilation errors. If one member wants to add some functions and fields in one class but not telling other members, it can affect the implementation of other classes, and cause fatal errors. Therefore, coherence is really important. About communication, understanding what our members mean is significant. If you don't understand what they're saying and just do your own part, there's high probability our program can't compile.

Besides, following the plan we developed together is also very important. If one member does not finish her part on time, it affects the progress of the team, and hence not good in teamwork.

Question. What would you have done differently if you had the chance to start over?
A: As mentioned in the above sections, we did not consider command line option initially, and it brought lots of troubles afterwards. Therefore, we would consider command line option at first and add corresponding statements in main.cc.

Also, we would consider using design pattern carefully. We did not use any design pattern when implementing this time because we found some problems occurred and we did not understand the design patterns fully. If we used design pattern, it would probably be easier to implement.