Final Design Document of CC3K

Junda Kong (j23kong) & Yang Li (y655li)

Outline

Part One: Relationships between classes

I. Main Classes & Subclasses
- 1.1 Welcome
- 1.1 Grid
- 1.2 Board
- 1.3 Cell
  - 1.3.1 Character
  - 1.3.2 Item
- 1.4 Room

II. Important SubClass: Player and Enemy
- 2.1 Player
- 2.2 Enemy
- 2.3 Example

Part Two: Introduction of some important requirements (including Plan of Attack)

III. Moving Option (two different ways to play CC3K)
- 4.1 Using command as default
- 4.2 Using WASD (ncurses.h)

IV. Difference between final version and original version

V. Questions in Plan of Attack

VI. Final Two Questions

Main Classes & Subclasses

1.1 Welcome

This class is used for printing the starting screen, it will print a 25*79 graph on the terminal and containing the information of our game, CC3K.

This class has three methods, (all three are *static void* method)

welcome(string)
welcome_win(string)
end(string, int)

the first one and the last one is used for our normal playing way (ie. Using command like “no”, “so” to make the player move), notice that we can choose to play this game by using inputing command or using WASD, if we use command to play, *welcome(string)* and *end(string, int)* will be used. Another method named *welcome_win* will only be used when we use <ncurses.h>, since the we all know that in <ncurses.h> we cannot use the standard input and output of C++, so the code for printing will be different.

These three methods' parameters are strings, which means we need files and fstream method to get the input content. We designed several files such as “welcom.txt”, “choose.txt” to print different content of welcome screen in order to efficiently reuse these methods in other functions.

1.1 Grid

This class is our main class of this game. It is a Singleton Pattern, marked as *game*, which will be used while the program running. It contains a series of Cells(Cells can be floor, wall, doors as default, players, items, and enemies), a pointer to the Board(which is the visible textDisplay to let users observe the Grid), an array of Room(each level has five rooms), etc.

We list some important fields for example:

*theGrid: Cell***
*td: Board**
*thePC: Player**
theCha: Room[5]

For other fields, their meanings can be seen by their variable names, *level* means which level player staying, *desc* means each action's description, *file* means whether we use the default file “dungeon.txt” or other outlet of the dungeon. Array *myEn*, *myPo*, and *myTr* are three collection of the special stuffs in each floor.

Now we talk about the four important fields and important methods.

Field *theGrid* is the main “place” for player, enemy move on, for potions, treasures exist in. theGrid is formed by 25*79 Cells, in order to make everything we designed can be placed on this “place”(or a map).

Field *td*, is the pointer to the Board. Each time we want to output(print) the Grid, the relative Board, marked as td, will be printed and output the visible graph.

Field *thePC* means the player we used. Since the Grid is singleton pattern, there is only one player each game, we can use several methods in Player class to access thePC's information.

Field *theCha*, which means the chambers(rooms) in the map. Every floor has 5 rooms, since we have to use the room to randomly generate everything with proper probability as required, we must create a field that represent each rooms.

As for methods in Grid class, some methods' usage are very obvious, such as *setFile()*, *getScore()*, (which means, get Players' gold value and determine the score), *isItOver()*, *isItWin()*, (to determine whether player dead or win), *getLever()*, (let others know which floor we already stay), *addPlayer(char)*, generating methods like *geneEnemy()*.

Remaining methods are,

easyPlay(std::string fname = "dungeon.txt");
commandPlay(std::string fname = "dungeon.txt");
void clearStuff();
void newFloor(std::string fname = "dungeon.txt");
void newArrange(std::string fname);
bool nearPotion();
void start();
void startArg();
bool ifRange(int &r, int &c);
void emAction();
void pcMove(std::string m);

easyPlay and *commandPlay* is two **static** methods that we will use in main functions, both these two methods are used for catching inputs and recognize the command and output the reactions. The differences between these two methos are, the easyPlay is used for WASD controlling playing way, which means, it needs <ncurses.h>, since it seems more easily to manipulate(for users who play that), its name is easyPlay. Other method commandPlay use command to play the game(e.g. “so”, “no”), and will never create new window and refresh terminal page.

Method *clearStuff* is used to clear all fields except *thePC*, it will be called everytime the player get upstair. The information of thePC will be reserved.

Method *newFloor()* and *newArrange()* is two different methods that have same usage. They are used to create new floor(i.e. new level, new map, everytime we get upstair). They always appear with **start methods** at the same time, which are *start()* and *startArg()*. When we use default dungeon given by instructor(we copied it into file *“dungeon.txt”*, this will be drawn exactly in *theGrid* and *Board* through *newFloor* and *newArrange*). Notice that we have a option that run this program with argument, which represent a file containing exact layout of each floor, we create a file like that called *“arrange.txt”* as an example. WHETHER HAVING AN ARGUMENT or NOT, will need totally different codes to complish. Thats why we have *newFloor() with start(), newArrange() with startArg()*. The first two, generating floor first, then add stuffs by *start method*, the rest two, directly generating everything except the player first, then add the player by **start method**.

Method *nearPotion()* is used for determine a potion is around the player, because we notice that we need to print the information like “meet an unknown Potion”.

Method *ifRange()* is always called by *emAction()*. The method *emAction()* will be called everytime after player have some valid motion(including attacking, using potions). Once *thePC* in any enemy's range(by *ifRange* determined), the enemy will stop moving and start to attack player. Both attacking action and moving action(randomly moving) for enemies are included in *emAction()*.

Method *pcMove()* seems obvious, but it is tricky, since the upstair usage is included by this method at the name time. Once the player meets the stair, the restart method will be called, and the new map will be generated(how to generate is still depending on whether we using an argument).

That's all important descrptions of Grid class.

P.S. Notice that the Grid class has two operator<< ways, it will be mentioned in Board.

1.2 Board

The Board class seems be similar to A4Q3's textDisplay class, but we concise it. It only helps us to make the Grid visible and relates to output methods of Grid class. The relationship between Board and Grid is aggregation, the Grid “has a” Board.

There are two output way of Board. Note that there are two friend functions(same as Grid).

friend std::ostream &operator<<(std::ostream &out, const Board &td);
*friend std::ostream &operator<<(std::ostream &out, const Board *td);*

Obviously, the first one is an override << of Board, the second one is of the pointer of Board. The

reason why we override it two times is that we want the output of *Board&* when we are using command to play the game(i.e. no <ncurses.h>'s print function), and the output of *Board** when we are using <ncurses.h>'s print functions(such as *printw, addstr*). As a result, each time the output in ***commandPlay*** method in Grid uses the first override, each time the output in ***easyPlay*** method in Grid uses the second override.

1.3 Cell

Cell class, each cell can be an element in Grid.

As default, cell is the floor, wall or door(thats the reason why we don't have Door class or Wall class, since Cell is defaultly regarded as them).

However, Cell has many subclasses, it can be characters and items.

The relationship between Cell and Grid is also aggregation.

When cell regarded as floor tile, it will use method *addNeigh(Cell*)* to add neighbors once after the *newFloor()* calling. The reason why we create the neighbors is that we have to arrange every tile in separate rooms(5 rooms) in order to make every stuff generate in each room with same probability. As a result, we create the field *numNeigh, alSet*(it will determine that whether the cell is already in a Room, which means, be set already), and we create the method *unSet()* to let outsiders access the value of alSet, create *arrRoom()* to push the Cell in the Room's field(it will be mentioned in Room).

For method getR() and getC(), it will give others exact position of Cell.

We modified the *SetCoords* method, once the this method called, it will be *notifyDisplay()* (which means, everytime the position is set, the Cell will reflect to change the Display of Board immediately, it will save a lot of time.)

Other methods will have same usage as the Cell class in A4Q3.

1.3.1 Character

The Character is a subclass of Cell. It containing Player and Enemy. HP, ATTACK, DEFENSE will be defined in this class. This is an abstract class. More information will be introduced in II.

1.3.2 Item

This is Item class, a subclass of Cell. We do not design the Item class perfectly since all stuffs about items(treasures, potions) are defined in this only one class(i.e. there is no subclass of Item). The field *value* represents how much money the item will give or how much influence the item will affect. The file *name* is used to mark the Item, in addition, everytime we want to describe player's action(once it meets items), we can use *getName()* to reach the strings and add them to the description directly.

The method *setTaken()* and the field *isTaken*(bool) are only special for dragon hoard. They determine whether the dragon dead and the dragon hoard is able to be taken.

The method useOn(Player&) is used for influencing the player's field value.

1.4 Room

The Room class is an class that seems useless. However, it is important when we generate potions, treasures and enemies. Our requirement is let these stuffs generate in room with same probability. We use the new technique we learned in class, *<vector.h>*.

The profit of using vector is that we do not have to focus on how many tiles we put in a room. The room is only for us to record and arrange the Cells properly. We use *addFloor* to make Cells push into the vector field *rm*. Once every tile has been set into a room, the way we generate potions and others will be easily. We only have to random a number between 0-4(there are 5 room, marked as 0, 1, 2, 3, 4), once we get the number, we select a room, then use *giveCoords* method of selected Room, it will change the parameters into random proper position (x, y) (or r, c), then return the integer to tell us which Cell in the Room will be replaced by potions(or treasures and enemis), then let this Cell becomes determined potion(or treasure, enemy) by using *changePos* method.

Moreover, the Room needs to be cleaned when we get into next floor(upstair), so there is a method

called *cleanRoom()*.

Important SubClass: Player and Enemy

2.1 Player

Subclass of Character.

Player contains Goblin, Drow, Vampire, Shade, Troll.

AND our DIY Player: Porco! (who will add 50 HP after smashing an enemy)

We use Double Dispatch to accomplish the attacking methods between players and enemies. Moreover, for special case, we have example in 2.3.

Some important fields are:

defAt: int
defDf: int
maxHp: int
neutral: bool

the defAt, defDf are used for resetting the player's ability when we get upstair. The maxHp limits the add HP methods to reach the higher HP than maximum value.

The *neutral* is determine that whether the player infuriate the merchants. The reason why merchants are special will be mentioned in 2.2. As a result, there is *ifNeu()* and *setNeu()* methods.

The method *isItOver()* determines whether the player dead. It will be called in Grid.

2.2 Enemy

Subclass of Character.

Enemy contains Elf, Merchant, Halfling, Dwarf, Human, Dragon.

AND our DIY Enemy: Yang! (who is strong and will give player 50 gold after death)

We use Double Dispatch like Player.

Enemy has additional field than Character, *hostile*.

Only merchants' hostile is different than other enemies. Recalling that we define a field named *neutral* in Player, the *hostile* for merchant is relating to merchants' action. Since their hostile is False, the merchant will not attack player at first. However, once the player attack the merchant, as special *attack(Player &)* method in Merchant, the player's *neutral* has been changed, then in this condition, though merchant's hostile = false, it find that player's neutral = false, all merchants will start to attack the player.

As for Dragon, dragon only generated when a dragon hoard generated, which means, dragon will not generate with other enemies at the same time! Dragon will generate first once the dragon hoard generate, and it will be counted as an enemy, so we use field ***dra(int) in Grid***, to record how many dragons have been generated. Once the dragon is generated, the dragon hoard will let the dragon point at itself. Using *setH(Item*)*.

Moreover, we have a method for dragon to move, which names *autoMove*, this method will give the random coordinates around the dragon hoard(of course, if dragon can reach that position), to let dragon move specially.

2.3 Example

Two methods in vampire.cc

```
void Vampire::useOn(Enemy &em) {
    int dmg = this->At;
    int def = em.getDf();
    double total = ((100.0 / (100.0 + def)) * dmg);
    int tot = (int)total;
```

```
        em.setHp(tot);
        this->setHp(-5);
    }

    void Vampire::useOn(Dwarf &em) {
         int dmg = this->At;
         int def = em.getDf();
         double total = ((100.0 / (100.0 + def)) * dmg);
         int tot = (int)total;
         em.setHp(tot);
         this->setHp(5);
    }
```

Attack Method in player.cc

```
    void Player::attack(Enemy &em) {
         em.useOn(*this);
    }
```

Moving Option

4.1 Using command as default

In main function, we will choose to type 'c', to use command to play, or type 'e', to use WASD to play.

Using command has already been told in cc3k.pdf. What we want to inform is that, if the player move invalid, the enemy will not move until you get the right motion. However, if you use potion or attack in wrong direction, the enemy will regard you as missing attacking and still get action.

4.2 Using WASD (ncurses.h)

We used the <ncurses.h> to make this game like an actual game with WASD controls. To accomplish attacking and using, type j+direction to attack, type k+direction to use items. There still has eight directions, which are W A S D 1 3 Z C.

Be aware do not touch Q by mistake, otherwise you will quit the game :(

(Since press Q will end the game)

Difference between final version and original version

We only considered how to move, how to attack, etc when we were doing our original version, but we did not consider how to randomly generate stuffs in every room(ATTENTION: ROOM! We only generated them randomly without considering the rooms at first). In the final version, we added a new class named Room to accomplish that.

We also did not consider the win condition and the lose condition at first. In final version, we added them, and decorate the output information, make it more visible.

Moreover, in the original version, we only can read the “dungeon.txt” to input the dungeon, which means we never considered the Command line options. Now, we can add an argument, to give a specified floor with giving layout in file.

Our original version also has several bugs, but we finally resolve them.

In addition, at first we only can play this game by using command, which is very complex and always make a lot of trouble when we test it. After learning some <ncurses.h>'s functions online, we add the WSAD controlling way, which can be chosen when the game start. This really make our life easy. Testing becomes convenient, and playing becomes much more interesting.

As for starting game, we add the welcome screen, which make our game like a real game, and we add some interesting information to introduce the game.

Finally, we add *two new Characters(Porco and Yang)*, just for fun!

Questions in Plan of Attack

Notation: Q2.1, Q2.2a, Q2.2b have not changed in final version

Question 2.1

We will design a Grid class, which contains the cells and the display board. Every time we want to generate the board, we only have to read the method in Grid(such as start(), generate() or something else). If we adding the Grid class and Cell class and Board class at the same time, we need to clearly know the relationship between each two classes, the Grid is the main one to read method, the Cell is like a subject that needs observer, the Board's job is let all stuff be visible. How to figure out the relationships and how to establish the relationship should be the difficulty that we have to overcome.

Question 2.2.a

We can use the visitor pattern, which is double dispatch. More information and more examples are in II, 2.3. *attack* method is let others strike me, and *useOn* method is let me strike others.

Firstly, Enemy is a abstract class, and the types of enemy such as dwarf, elf will be the subclasses of Enemy, and when we are going to generate it, we have to use random number to calculate the probability of the occuring of each enemy. Then, we use random number to determine whether the enemies can be put(i.e. when coords r, c becomes a valid place, which is '.', the enemy can be set on the Board).

Different enemies generated by calculating random numbers,

Different players generated by using command. The generating way is different but the setting way is almost same. The way where we have to drop down the player is same as how we drop down an enemy on the Board.

Question 2.2.b

We design two types of methods for enemies and players, which are useOn(&) and attack(&),

useOn: let the attacking use on some one, and minus the Hp

attack: let some one else attack me, i.e. be attacked

We will use clever combination of overriding and overloading (VDP), like we've been taught in lectures(striking turtle/bullet by rock/stick).

The player will use the same techniques since player is same as the enemies, they both have attacking action and being attacked action(defending action).

Question 2.3.1

We can use the visitor pattern. (Original design)

However, at last we just designed Item class without any subclass. What we designed now is, when we use the method *useOn* of Item, we consider the all the cases(condition) of the Item type. Since the Item have value, how to use this Item(more specificly, is “be used”) determine the item's name, this determination(or identification) will be finished in just one method. So once the player using the potion, only have to call *Item::useOn* to accomplish it and print out the relative method(print out the field: string *name*).

Question 2.3.2

We can let Treasure and Potion be subclass of same class, assume class A, and when read the generating method, we can directly let the parameter become A to generate. (Original design)

It seems that, since we assemble these two thins together in one class, we failed to use exactly one function, but we still can use the same method to generate treasures and potions, which has already been designed in the constructor of the Item class. However, in Grid, generating these two stuffs still needs several same codes to accomplish.

Final Two Questions

1. What lessons did this project teach you about developing software in teams?

Thinking together always better and faster than thinking alone. Though this game was difficult to design, we still finished it with cooperation. The most helpful factor is debugging. Each one of us is not a excellent tester, for this long coding program, we cannot debug alone since there may have plenty of bugs waiting for us. We look through and check the codes together every day, and come up with different views, debate with each other, solve the trouble at last. Therefore, two men's observation is always better than one!

Moreover, we two both have some weakness in C++, one is not good enough on designing pattern, the other one is not good enough on checking long code. That's the main reason why we work together, we share our advances to each other and eliminate our disadvantages as many as possible by working together, leaning to each other. In other word, developing software in teams needs SHARE!

During this assignment, we found that we still have to word hard on some area of C++, especially what we learned after midterm, because the content after midterm become more and more difficult. We try to use as much new knowledge in this assignment as we can, such as vector, cast, and we supervise each other to review the important knowledge.

Sometimes, one will have a question that the other one never think about, during coming out the new question(problem), we improve our skills. Thus, we have to LEARN TOGETHER.

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

We would not let the Item class be so implicit, we will make several subclasses of Item, such as Potion, Treasure and add some fields, then we can reuse more code efficiently.

Moreover, we would consider to generalize Grid class and Board class to make the relationship between Grid and Cell more clear, more easily to manipulate.

Also, we might use more “vector” rather than multiple array. That sounds more interesting and more efficient.