Final Design Document(CC3K)

Owen Ling and Yifei Zhang

Overview:

-Floor and Cell
-Item
-Character
-Map Generation
-Combat
-Item Pickup

Our group has divided this project into the sections listed above. We completed Floor and Cell, Item, Character, and Map Generation before we completed Combat and Item Pickup.

Cell

We first created the Cell Class, which contained information of its
-***type***,which can be vertical wall, horizontal wall, empty, doorway, passage, or floor tile. We have implemented accessor and mutator functions for this.

-state, this is specifically for “floor tile” cells, their state can be empty, occupied by potion, occupied by player, or occupied by enemy. We have implemented accessor and mutator functions for this.

-Chamber, This is also for the “floor tile” cells. Having the chambers helps us massively during the generation of various items in the map, as there are certain items/characters that cannot be in the same chamber, such as the character and the stairway. We have implemented accessor and mutator functions for this.

-Enemy, each cell also contains an enemy pointer that stores an enemy if the enemy is in the tile, or a nullptr if there is no enemy in the tile. We have implemented accessor and mutator functions for this.

-Decorator, this includes a pointer to a potion or gold for the player to pick up. We used the Decorator pattern where the player acts as the concrete component, and the gold and potions being the decorators onto the player.We have implemented accessor and mutator functions for this.

-***Stair,*** this is a boolean variable for “floor tile” Cells, if the floor tile is the staircase, then it has value “true”, else, it has value “false”. We have set this variable to a default value of “false” and have implemented a mutator function for this.

Floor(The basic features, which excludes Combat)

Our Floor class is basically our gameboard. It contains the following:

Player: We decided to store the player in the Floor as there can only be one player character in an entire floor. We have a player pointer to modify the player character’s status, and we also have two variables that store the players coordinates, which makes Combat and Player Movement very convenient, as we know where the PC is at all times; we don’t have to iterate through the entire Floor everytime to find our PC. We have accessor and mutator functions for the PC.

Level: There is a variable that stores the current level that the PC is currently on. Whenever the PC gets to the stairway of the current level, the Floor is cleared, with the PC still intact. Then, a new floor generation happens, and the “level” variable gets incremented by 1. The “Floor” variable starts at 1. When it reaches 6, the game prints out a congratulatory message and exits the game. When the user types ‘r’ into the command, everything is the same, except that the PC is also destroyed and recreated, and that the “Floor” variable gets set to 1.

Character Movement: This includes Player Movement and Enemy Movement.

-Player: since we have the PC’s coordinates stored in the Floor, when the user enters a command for the PC to move in a certain direction, wecheck the tile in that direction to see whether it is a “floor tile”, “passageway”, or “doorway”, and whether it is empty. If it is, then the PC moves there; if it is not empty, then nothing happens.

-Enemies: all enemies on the map move when the player moves. After the user enters a command, the player first moves. Then, we iterate through the entire loop. If an enemy is a dragon, then it doesn’t move, if it has a player beside it, then it doesn’t move. We have a variable stored in the Enemy class that checks whether the enemy has been moved, if its value is true, then we skip it, if it is false, then we move the enemy. All Enemies have this variable reset to false after we iterate through the entire Floor.

Character

We used inheritance for creating our characters. We have an abstract base class that acts as the base for its subclasses: Player and Enemy. Character Class contains health, attack, defence, and gold, which are common to both classes. We have implemented accessors and mutators for all of them. There is a virtual function for Death in the base class, since the death of PC and of Enemy yields different results. There are also other features that are not present in both Enemy and Player that need to be implemented.

-Enemy: we have implemented variables indicating whether an enemy is hostile or not, in order to determine whether Merchants will attack the PC. The "moved" variable helps in the Enemy Movement, preventing the Enemies from moving twice or more. When enemies dies, a different value is produced by the death function in order to determine how much gold is dropped. There is also a variable that keeps track of the type of of Enemy it is.

-Player: two boolean variables determining whether the PC has a compass or a barrier suit has been implemented and has default value false. When the Player dies, the user has the option to restart the game or to quit the game. PC picks up potion and gold using a decorator pattern as mentioned above in the "Cell" portion.

Item

Different items are derived from one abstract class, which contains all methods needed by an item. Since each item only has one effect, there is a base class that is used to override all methods and set them to trivial methods. And each type of item are decorators of the base class, which overrides the needed method from base class. In this case, all items other than major items will change only one parameter of the Player object, so an integer is returned when calling a method from an item. For major items, due to shortage of time, barrier suits are not implemented. When a compass is picked up it will change a boolean inside the Player object (the boolean indicates whether the PC has the compass).

Map Generation

-Player Generation:

We stored all "floor tile" cells into a vector, we then shuffle the vector and take the first element in the vector. We also store the chamber number of that Cell in a variable.

-Stairway Generation:

We then store all “floor tile” cells, except for the ones that are in the same chamber as the PC(using the variable store before) into a vector. We shuffle the vector, and then take the first Cell. That Cell will be the stairway.

-Potion Generation:

We store1,2,3,4,5 in a vector, the number stands for the 5 chambers. We shuffle the vector 10 times, each time taking the first number in the vector. Then, we collect all the empty Cells in the chamber of the chosen number into another vector, and shuffle that vector, and take the first Cell in the vector. That Cell would contain a potion. Note that after each iteration(out of the ten), the vector storing all empty cells in the specified chamber is cleared.(Please check the bottom of this section)

-Gold Generation:

Gold is generated using the same technique as generating Potion.(please check the bottom of this section)

-Enemy Generation:

-Dragon:

We check through the map to see if there are any Dragon Hoards of Gold. If it exists, then we store the 8 surrounding Cells into a vector(only if they are empty). Then we shuffle the vector and take the first Cell. That Cell will have a Dragon.

-Every other Enemy:

Generated using the same technique as Potion and Gold.(Please check below)

How do we determine which type of gold/potion/enemy to spawn?

We use a different number for every different kind of item. We place these numbers into a vector. The amount of a specific number being placed into the vector depends on its possibility to be generated. then we shuffle the vector and take its first element. The type of item that corresponds to that number will be the item spawned.

Combat:

-***playerAttack:***

-The player chooses a direction to attack, we use the formula given to calculate the damage and the Enemy's remaining HP. If the Enemy dies, then we check his type and change the Cell containing the Enemy to the Corresponding Item spawned, using a Decorator.

-enemyAttack:

-The enemy scans around the 8 Cells around it to determine whether the PC is in those Cells. If it is, then the enemy deals damage to the PC and the damage and PC's HP are calculated. If the PC dies, then the Player has the option to restart(r) or quit(q).

-Printing:

-Since we had to print the health of the PC after printing the Floor, and the actions after the health of the PC. We created two vectors. One storing the damage and one storing the type of the Enemy. We first move the PC, then move the enemies, the enemies then attack the PC if they are in range, each time they attack the PC, the Enemy's damage and type are recorded into the 2 vectors. After that we can then print the Floor, followed by the PC's information, including its HP, and then, based on the 2 vectors storing the damage and the type of Enemy dealing the damage, we can print the actions done.

Item Pickup

Items are picked up using the Decorator Pattern. Where the Items(Gold, Potion, Compass, Barrier Suit) are decorators and the PC is the concrete component being decorated.

Questions

Q: What lessons did this project teach you about developing software in teams?

It taught us that communication between team members is very important. When experiencing problems, we need to inform our teammates instead of keeping it to ourselves. We had to constantly update each other about what we are doing and how we are going to do it. This helps us to better structure our plan to build the program and helps each of us understand what our tasks are. It really saved us a lot of time and effort as we are completing this entire project with 2 people and limited time, on top of experiencing the sudden, abrupt departure of one of our teammates.

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

Most importantly for us, we would need to know for sure if our teammates are fully committed to the project and have no plan to leave the project anytime. The departure of our teammate has placed a huge toll on us two. Secondly, we could have understood the project better before we went into our discussion.

Q: 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?

We stored all types of races into one class, with a char variable that distinguishes them. Thus, to add another race, we simply add another race into the class, with its own distinguishing character, health, attack, etc.

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

We have numbers corresponding to different types of enemies. We place the numbers into a vector. The amount of numbers put in depends on the probabilities given. Then we shuffle the vector and take the first element. The element would be the enemy spawned.

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

We can use the Factory Pattern and each ability can be treated as a method and can be dynamically added to each type of enemy.

Q. What design pattern could you use to model the effects of temporary potions so that you do not need to explicitly tract which potions the player character has consumed on any particular floor?

We can use the decorator pattern. The PC would be the concrete component and each potion would act as the decorators so that the effects are automatically imposed onto the PC when the potion is used.

Q. How could you generate items so that the generation of Treasure, Potions, and major items reuses as much code as possible? How would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code? How could you reuse the code used to protect both dragon hordes and the Barrier Suit?

We can write the general process of generation into a function that takes a vector as a parameter, it shuffles the vector and produces the first element in that vector. In this case it would be a number that corresponds to a certain type of item.