CS 246 Assignment 5 Final Project Report

Introduction

The Game of ChamberCrawler3000+ is a simplified rogue-like and this terminal-based program is the final project of CS 246 for me and my teammates. The following is a report for this project that contains our design of implementation, some demonstrations of the UML diagram, and a planned responsibilities assignment schedule within our group.

Design Summary

Our implementation is based on the polymorphism of virtual methods. First, construct an abstract base class Cell and derive three main types of cells that can be displayed in one unit of tile, those are, Items (include Potion, Treasure, BarrierSuit and Major Items), Enemies (include Vampire, Werewolf, Goblin, Merchant, Dragon, Phoenix and Troll) and Players (include Human, Dwarf, Elves and Orc). We construct an abstract class for each type and derive several concrete classes for their specific races or different items. In each concrete class, we would override virtual methods to implement their specific abilities or behaviors. In addition to the concrete units, we would implement the class Cell as a wrapper for Object and it would denote the position of each cell in the overview floor. To implement the floor, we decide to construct a class Grid which would store 79 * 25 cells and our game.cc will implement this game by controlling this class. Finally, each Grid will have a printMap function which calls functions from Cell that records the floor in characters and implements the methods to display the floor on the screen.

Overview

Classes/methods:

- Main
  - Receives input from the user and outputs text information based on the user's commands.
  - Output text information based on user commands.
- Gird
  - enemyInrange(int, int)
    - Return true if the position of the enemy is in range, false otherwise.
  - playerInrange(int, int)
    - Return true if the position of the player is in range, false otherwise.
  - init(char)
    - Initial the floor, and set the position of different characters.
  - update()
    - The method will go through the grid, and update the actions of enemies and players to grid.
  - setFloor()
    - Create the Floor without players, items, and enemies.
  - setEnemies()
    - The method will produce a random position and check if that position can create an enemy. If true, create the enemy in that position.
  - setStair()
    - Randomly create Stair
  - setPotions()
    - Randomly create Potions.
  - setTreasures()
    - Randomly create Treasure.

    - setCompass()
        - Randomly create Compass.
    - setDragon(shared_ptr<Cell>)
        - The method will produce a random position around the treasure and check if that position can create a dragon. If true, create the dragon in that position.
    - setPlayer(char)
        - Overload method used to randomly create player
    - setPlayer(shared_ptr<Player>)
        - Overload method used to randomly create player
    - setBarrierSuite()
        - Randomly create BarrierSuit
    - setChamber()
        - Create different chamber
    - enemyAction()
        - Enemy attack player
        - Make enemy move randomly
    - playerAction(string)
        - Check the command that the player input, and do the corresponding action. (move, attack, use potion)
    - printMap()
        - Print the map
    - clearFloor()
        - Clear all characters in the grid.
    - getPlayer()
        - Return a cell pointer which contains a player pointer
    - getStair()
        - Return a cell pointer which denoted as stair
    - getCell()
        - Return a cell pointer which on a position
- Cell
    - setItem(shared_ptr<Item> i)
        - Initialize item, denote the type as 2
    - setEnemy(shared_ptr<Enemy> e)

        - Initialize enemy, denote the type as 3
    - setPlayer(shared_ptr<Player> p)
        - Initialize player, denote the type as 1
    - setStair()
        - Initialize stairs.
    - getType()
        - Return the value of type.
    - getDisplay()
        - Return a char. (Player @)(Enemy W, V, N, M, D, X, T M) (Item P, G, B,) (Stair /) (Compass C)(. # + - |)
    - getR()
        - Return an integer representing the current cell in the row
    - getC()
        - Return an integer representing the current cell in the column
    - getNeighbours()
        - Return a vector of shared_ptr<cell> which contains the cells which are around the current cell.
    - update()
        - Update the map
    - move(int ar, int ac)
        - This is an overload method. This method allows the pc to move, attack, and use potions.
    - attack(int ar, int ac)
        - This method first determines if the input target is an enemy, and if so, attack it, and return the string of the player action.
    - use(int ar, int ac)
        - This method first determines if the input target is a potion, and if so, use it, and return the string of the player action.
- Item
    - getName()

        - Return a char that represent the current symbol of the item
    - getReal_name()
        - Return a string that represent the current name of the item
    - useItem(Player &player)
        - Pure virtual method that is used to return a string of PC's action.
- Treasure
    - setIs_dead()
        - If the dragon is dead, set true, then the PC can get a dragon horde.
    - useItem()
        - Add gold to PC
- BarrierSuite
    - setIs_dead()
        - If the dragon dead, set true, then PC can get barrier suit
    - useItem(Player &player)
        - Add barrier suit to PC
- Potion
    - useItem(Player &player)
        - Pure virtual method that is used to return a string of PC's action.
- WA
    - useItem（Player &player）
        - Reduces attack value
- WD
    - useItem（Player &player）
        - Reduces defense value
- PH
    - useItem（Player &player）
        - Reduces HP
- BA
    - useItem（Player &player）
        - Increase attack value
- BD

    - useItem（Player &player）
        - Increase defense value
- RH
    - useItem（Player &player）
        - Restore HP
- Enemy
    - attack(Player &player)
        - Pure virtual method use to attack PC
    - getHP()
        - Return a integer that represents enemy's HP
    - getAtk()
        - Return a integer that represents enemy's Atk
    - getDef()
        - Return a integer that represents enemy's Def
    - hasCompass()
        - Return a boolean that represents if the enemy has a compass
    - getHorde()
        - Virtual method used to get the horde.
- Dragon
    - attack(shard_ptr<Player>&) override
        - Enemy attack PC and return a string that represents the damage that enemy did.
    - getHorde() override
        - Get the horde
    - isDead() override
        - Return a boolean, true if the dragon dead, false otherwise.
- Goblin
    - attack(shard_ptr<Player>&) override
        - Enemy attack PC and return a string that represents the damage that enemy did.
- Werewolf
    - attack(shard_ptr<Player>&) override
        - Enemy attack PC and return a string that represents the damage that enemy did.
- Merchant

        - attack(shard_ptr<Player>&) override
            - Enemy attack PC and return a string that represents the damage that enemy did.
- Phoenix
    - attack(shard_ptr<Player>&) override
        - Enemy attack PC and return a string that represents the damage that enemy did.
- Troll
    - attack(shard_ptr<Player>&) override
        - Enemy attack PC and return a string that represents the damage that enemy did.
- Vampire
    - attack(shard_ptr<Player>&) override
        - Enemy attack PC and return a string that represents the damage that enemy did.
- Player
    - addGold(float n)
        - Virtual method use to add gold to player
    - addTempBuff(string buff)
        - Add temp buff to player, after player use WA, WD, BA, BD potion, after the remove the buff after the next player movement.
    - attack(shared_ptr<Enemy>& e)
        - Return a string that represents the damage that PC did.
    - use(shared_ptr<Item>& i)
        - Return a string that represents the type of potion that PC used.
    - beAttacked(Vampire* x)
        - Return a string that represents the damage that the enemy did.
- Dwarf
- Elves
- Human
- Orc

Resilience to Change

Since we used a pure virtual method for the enemy and player, if we want to add more player character races or more enemies. All we need is to add the more base classes inherited from the enemy or player, and override the pure virtual method. Our design might nor be the best, but with separate enemy and player classes, we are able to add different skills for different types of enemies or players. Furthermore, if we want to add a feature that allows the player to go back to the last floor, we could just simply increase the number of arrays to save different grids.

Question 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 classes?

Answer:

Since races have the same Attributes with different Abilities, we created a base class Player and all races are derived from Player construct with their own “HP”, “Atk”, “Def”, and different abilities. It is very easy to add a new race in this way.

Question 2:

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

Answer:

Firstly, generating different types of enemies is similar to players, constructing an abstract base class *Enemy* and having each race derived from it with their unique abilities. The differences

between Player and Enemy objects is how they move on the chamber; by the description, enemies are moving randomly within the confines of the chamber where they spawned in, whereas player moves by user's command. Thus, in order to have this kind of behaviour, there would be a different move() method applied to each enemy and using a <random> library to achieve that.

Question 3:

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

Answer:

Depending on whether it is passive skill or active skill, for instance, skills like gold stealing for goblins will be more likely implemented as a passive skill as generally known. Which means once a goblins arise an attack, this ability applies. In this case, override the attack() method of enemies who have passive skills such as vampires-life stealing, trolls-health regeneration, so that we need to track specific enemies. In terms of active skills (for example Merchant-item sell), it has to be an individual method since each of them acts totally different. We applied a Visitor Design Pattern to deal with such a case, allowing the enemy and player to interact with each other.

Question 4:

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

Answer:

The preferred design pattern here would be decorator design pattern. Since potions change the status of the play with either good or bad effects, we construct a decorator class *Buff* to

generate *BA/BD/WA/WD*. To differentiate positive and negative effects, two decorations would be derived from *Buff; Buffs* and *Debuffs* , buff can be applied multiple times and once the buff is activated, the original data of player's Atk and Def will be recorded though method. Alse there would be a clearBuff() method which will be applied once the player switches the stair while there Atk and Def will be recovered.

Question 5:

How could you generate items so that the generation of Treasure, Potions, and major items reuses as much code as possible? That is for example, 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 hoards and the Barrier Suite?

Answer:

An abstract base class *Item* would be constructed, *Treasure and Potions* are derived classes from *Item*. These two Items have the same functionality that modifies Player's attributes in different ways (one temporarily, the other permanently). Thus, in order to avoid code duplication, they are all set with "name", "value" and overrides method "use" to have different performance. In terms of major items, compass does not need to be an individual class, it becomes a field of Enemy, since it is created with an Enemy and used when Enemy dies. BarrierSuite and dragon hoards are exactly the same Treasure with the only difference that BarrierSuite can only be defined once in a game. In this case, we need a global value to record the presence of BarrierSuite.