CS246 Group Project - CC3K

Kabir Ashish Wahi, Thanh Le, Haichun Kan

Overview

Our project, CC3K, embodies an object-oriented programming mindset. The objects in the project can be basically generalized into the following three categories: ***Game***, ***Character***, and ***Item***.

Game is the interface that allows the user to interact with the CC3K game. The two most important fields in the *Game* class are *player*, which is a ***Player*** (from the *Character* category) pointer, and *displayGrid*, which is a field to store the floor map. *Game* reads from user input and then controls the *Player* to do things like moving in the map, attacking ***Enemies*** (also from the *Character* category) and using ***Potions*** (from the *Item* category). *Game* also includes the initiation of the game, for example, randomly *generateItems* and *generateEnemies* on the map, generate an invisible stair, etc..

Character is a pure virtual class that includes fields like *HP*, which represents the *Character*'s health, *atk*, which represents how hard it can hit others, *def*, which represents how hard it can be hit by others, and *gold*, which is the amount of gold that *Character* is holding. It can be divided into two subcategories, which are ***Player*** and ***Enemy***. *Player* is one of ***Human***, ***Elf***, ***Dwarf***, and ***Orc***, and Enemy can be ***Vampire***, ***Werewolf***, ***Troll***, ***Goblin***, ***Merchant***, ***Dragon***, and ***Phoenix***. Each class (race) may have some special traits or abilities. *Player* and *Enemy* can attack each other. For *Player,* there is also a *usePotion* function as mentioned in the above paragraph.

Item is what can be used by *Player*. *Item* is also a pure virtual class, whose subclasses include ***Potion***, ***Gold***, ***Compass*** and ***BarrierSuit***. *Potion* onced used, will affect the *Player*'s *HP*, *atk*, or *def*. *Gold* when picked up, its value will be added to the *Player*'s gold. After *BarrierSuit* is used, all damages to the *Player* afterwards will be halved. If *Compass* is picked up, the stair then becomes visible.

Updated UML

The largest change we made on the structure is the *Gold* class. We did not implement the *Normal*, *SmallHoard*, and *MerchantHoard* classes because these classes don't have "special abilities" except for their different amount of *value*. To deal with the different values, we just need the constructor to accept the *value* as a passed parameter. If we want to have other values of *Gold*, we also don't need to specifically create a new class.

The other small changes are for the convenience of implementation. For example, we added *addHealth* and *addGold* functions in *Player* although we have *setHP* and *setGold* functions inherited from *Character*. This is to better deal with the *Elf, Dwarf*, and *Orc*'s special abilities.

Design

Inheritance and Polymorphism

The *Character* and *Item* are base classes. *Player* and *Enemy* inherit from *Character*. Other concrete classes inherit from these classes. What is common is implemented in the base class and what is special is implemented in the concrete class.

Design Pattern

The design pattern we apply is the ***Decorator Pattern***. Since the effect of the *BA, BD, WA, WD Potion*s won't be brought to the next level while the effect *RH* and *PH Potion*s will last to the end of the game, we design a new class, ***Buff***, for the effect of the former four potions, in which the effect is denoted by the *value* field. *Buff* is the abstract "**Decorator**" inherited from *Player*, and it has two child classes, ***AtkBuff*** and ***DefBuff***. So *Buff* itself is a *Player*. *AtkBuff* is *atk* related potion effect and *DefBuff* is *def* related potion effect. As a decorator, *Buff* has a *Player* pointer. When *Player* uses one of those four potions, a new *AtkBuff/DefBuff* class will be created with the *player* pointer pointing to the original *Player*. To achieve the functionality of *Player*'s *getAtk/getDef* function which should return the affected *atk/def* value, we implement (and override) the *AtkBuff/DefBuff*'s *getAtk/getDef* function as *value* + the return value of the *player* pointer's *getAtk/getDef* function. Finally when *Player* goes to the next level, all these temporary effects can be removed through the *Game*'s *player* field's destructor, calling its *player* pointer's destructor, until the last, undecorated *Player*.

Resilience to Change

If we want to …

1. Add a new *Player* or *Enemy* race or a new *Item*.

   Since we have abstract *Player*, *Enemy,* and *Item classes,* it would be easy to add other child classes with some special traits within only a few changes.

2. Add extra enemies or items in the current floor.

   The helper function that we use extensively in *generateItems* and *generateEnemies* is the *randomPosn* function, which takes in a parameter of the chamber index and generates a position in the chamber that was previously unoccupied. If we want to add more enemies or items, we just need to change in the for loop statement how many times we are going to loop through.

3. Have a map where the positions and types and amount of items are already assigned.

   Our program allows this by accepting command input through filestream input, and runs a mapDetection algorithm to allow for correct random spawning of enemies/items.

Answers to Questions

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

   We made ***Player*** class abstract , and all the player races as child classes of ***Player***. This will make generating each race easy because the common functions like *getHP*, *getAtk* are already implemented in parent classes, and in each race class the only things we need to worry about are the special traits and special abilities. This means in each race class we only need to rewrite the constructor as they have different *HP, atk, def* values and the ability-related functions, which for *Elf* is *addHealth* and for *Dwarf* and *Orc* is *addGold*. For *Elf*, why we don't have *addAtk* or *addDef* functions like *addHealth* to make the potion effect always positive is that we deal with it in the *Buff* class. If *Player*'s *getRace* function tells us this is an *Elf*, in *Buff*'s constructor the *value* field will be changed to its absolute value. It's just two lines of code, so if we want to add another race whose special ability is related with *atk/def* potions, there would still be very few changes needed. In addition, as we initialize the player in ***Game***, we will just do "new *Race*(...)".

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

   The idea is the same because ***Enemy*** is also an abstract class containing common features of all enemies. One small difference is that when we generate enemies we need to generate more than once and each race has a different probability of being spawned. So in ***Game*** we have a vector to store all ***Enemy*** pointers and there's also a *randomNum* helper function that can decide which race to be initialized. When initializing an ***Enemy***, it's nothing different from initializing a ***Player*** race. We still do *new Race(...).*

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.?

   As mentioned in the previous two questions, we will implement that in each race class. The first thing is to identify the place to put these codes —- do we write a new function for each? Obviously we are not going to write virtual *stealGold*, *regenerationHealth* functions in parent classes. We know that these actions take place when the *Enemy* is interacting with the *Player*, i.e. attacking. Therefore, we write the special abilities in the *attack* function. Since the *attack* function takes in a *Character* pointer parameter, representing the *Player* to be attacked, then the effect of *stealGold*, *stealHealth* on *Player* can also be easily done.

4. 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?

We used the decorator design pattern. This is already discussed in the Design Section.

5. How could you generate items so that the generation of Treasure, Potion, 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 hordes and the Barrier Suit?

   We have two helper functions, *randomPosn* and *randomNum*. The function *randomPosn* accepts an integer passed, which is a chamber's index, and will produce a position that is not occupied in that chamber. The *randomNum* function also takes in an integer, which is an upper bound, and will produce an integer between 0 and that upper bound. For instance, if we want to randomly spawn a *Gold*, we first use *randomNum(5)+1* to get a chamber's index where the *Gold* will be placed. Then we use *randomPosn(chamberIndex)* to get an unoccupied position. Finally, since there's ⅝ probability to spawn a normal gold, ⅛ probability to spawn a dragon hoard, ¼ probability to spawn a small hoard, we use *randomNum(8)* to get an integer. If that integer is less than 5, then that means a normal gold should be generated. If the integer is 5 or 6, it means a small hoard should be generated. Otherwise it's a dragonHoard. The process of generating enemies is similar, except that the probabilities are different. But as the *randomNum* function takes in an upper bound parameter, it's not a problem. These two functions allow us to reuse code. To reuse the code of a dragon protecting an item, our *Item* class has a *guarded* field representing if the item is being protected by the dragon. In addition, there's also *isGuarded* and *setGuarded* functions to return and set the *guarded* field. Moreover, the *Dragon* class has an *Item* pointer. If the player wants to pick up an item, the first thing to check is *isGuarded*. In *Game*'s *update* function, when the dragon is slain (*HP* is 0), the item's *guarded* field is set to false through the item pointer's *setGuarded* function. This code can be reused because, the *Dragon* class has an Item pointer, and all items have the *guarded* field and those two functions.

Extra Credit Features

1. Added 3 new player characters
   1.1 Magic Archer - has ranged attack, fires projectile in a particular direction damaging everything in that direction.
   1.2 Valkyrie - has AOE attack, does not have to specify directions, damages all enemies in a one cell radius.
   1.3 God - slains everyone on the floor, do not have to specify direction.
2. Added 1 new enemy race
   2.1 Reverser - attacks the player with a damage equivalent to the player's attack stat.
3. Added special abilities for enemies

3.1 Vampire - lifesteal, heals ⅕ of the damage it does
3.2 Goblin - has a ¼ chance to steal 1 gold from the player if the player has any, can be taken back by slaining the goblin
3.3 Troll - has ¼ chance to attack twice

4. Added 1 new item
   4.1 Potion Book - spawns only on the 1st floor, protected by a Dragon, gives knowledge of all potions to player when walked over

Final Questions

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

   Communication of ideas and plans are important. Sometimes the work we do is duplicated. And when using GitHub, it's better if each person has a branch and has a person proofreading the code before merging branches. We only had a master branch and some code was overwritten.
2. What would you have done differently if you had the chance to start over?

   We would have a short meeting everyday to share what everyone has done, what's the difficulty, etc. We would also make branches and have the other members proofread the code before merging branches.