CC3K project documentation

Ruohan Jin, Yidan Wang, David Dong

Introduction

CC3K is a game where a player is randomly generated in one of the 5 chambers on a floor, and the floor on which the player was generated is denoted the first floor. The player needs to reach the fifth floor alive in order to win the game. Throughout each floor, there are enemies, potions, and golds spawned on each floor. If the player steps into an enemy's attack range (within one block radius of the enemy), it will be attacked by the enemy. A player may choose to attack the enemy, but after the player's turn, the enemies will attack the player back. Potions can be picked up by the player within one block of radius, they could give you positive effects, or negatives, so be careful! A player may also pick up gold that is spawned on the floor and can also collect gold after it slays an enemy. The only way to go to the next floor is through the stairs generated on each floor, so the stair is your ultimate target! The number of golds you have determines your score after you complete the game.

Overview

- **class `Item`**
    - This class has two subclasses: class `Potion` and class `Gold`, both inherit from their base class `Item`.
    - In class `Potion`, we have used Decorator Design Pattern, and details will be explained below.
    - In class `Gold`, there are multiple integer fields representing number of golds store in every "gold". The constructor of `Gold` has all fields set to their default value. When generating golds on the floor, we just set the specific field (the type of gold we want it to be) to its default value and other fields zero, so whenever a

player picks a gold, the corresponding gold number will add to player's gold storage through `getgold()` function, which checks which field is non-zero.

- **class `Character`**
    - This class uses Observer Design Pattern, with two subclass: class `Player` and class `Enemy`.
    - Player is the subject and enemies are the observers(details below).
- **class `Cell`**
    - This class is essentially representing anything that can be stored in one unit area throughout the floor, including wall, floor tile, passage, and doorway. It has a boolean value that checks whether the current cell is occupied by an item or is an valid area that the player can step on, as well as accessors that get the corresponding potion, gold, or enemy on the cell.
- **class `Chamber`**
    - This class contains a vector of shared pointers with type `Cell` that saves the empty floor tiles in the chamber. The primary role of this class is to handle the spawning of items on the floor. To spawn the items, the `randomFloorTile()` function is called to randomly selects and returns one empty floor tile from the list. Once an item is spawned, the corresponding cell's pointer is removed from the vector to ensure it only contains pointers to cells that are empty floor tiles, ensuring the subsequent spawns will not use the same cell space. There is also a boolean value to determine whether the player is in the current chamber to help generate stairs, making sure they are not spawned in the same chamber.
- **class `Floor`**

- This class is in charge of the distribution among players, enemies, items etc. through the entire floor. It has an int field represent the current level of floor the player is within. It contains one vector of shared pointers with type `Cell`, one vector of shared pointers with type `Chamber`. It uses several “spawn” functions including `spawnPlayer(), spawnStair(), spawnPotion()`, `spawnGold()`, and `spawnEnemies()` along with the `PRNG` to randomly determines the type of the items and their position when generating player, stair potions, gold, and enemies on each floor. Additionally, the Floor class is also responsible for managing movements within the game grid. It includes a `moveBetweenCell()` function, which transfers information from one cell to another. This function is called whenever a player or enemy moves, effectively updating their positions and the states of the involved cells.

- **class `GameFlow`**
  - This essentially acts as the controller of the game. It contains a shared pointer with type `Player` and a shared pointer with type `Floor`. It uses functions including `updatePosition()` which renews player’s position after it makes a move, `checkPlayerDeath()` checks whether the player is still alive, `validDirection()` whether the direction player moves towards is valid and the engine to start and restart the game if needed.

Design

We used multiple inheritance, decorator design pattern, and observer design pattern for the overall structure of the project.

Decorator Design Pattern

In class `Potion` we used decorator design pattern with core object called *`BlankPotion`* and `Decorator` class including different types of potions. Whenever a player picks up a potion, the `Decorator` will add its corresponding features on the *`BlankPotion`*. Inside every subclass (different types of potions) of class `Decorator`, there's a function called `usePotion()` which takes a player pointer as parameter and apply its effects to hp, atk, def accordingly. In class `Player`, it stores a potion pointer which is essentially a linked list of `Potion`. Every time a player picks up a potion, it is added to the top of the linked list, and whenever the player tries to attack, it goes through the linked list and calculates the effects on atk and def brought by the potions it has picked up.

Observer Design Pattern

We applied observer design pattern when implementing class `Character`. Essentially, class `Player` is the concrete Subject and class `Enemy` is the concrete Observer. Whenever a player moves, it will notify all enemies through the `notify()` function. If player is within one block radius of an enemy (enemy's attack range), the enemy will attack the player.

Inheritance

We applied inheritance on multiple classes including class `Character` and class `Item`. We collected all the fields and methods that would be used in all of the subclasses and put them in the base class to avoid duplicate code.

Forward Declaration

We applied Forward declaration in our code as well. For example, we added the forward declaration of class Enemy and class Potion to the class Player, and added forward declaration of class Gold, Potion and Enemy to the class Cell so that we can avoid the potential issues caused by including cycle. By using forward declaration, we can also reduce the C++ build times.

RAII Idiom

We strictly followed the idea of RAII through the development of our project. By employing RAII, resources requiring explicit clean-up, such as heap-allocated objects, are automatically managed and cleaned up. Our code eliminates the need for the

`delete` keyword, utilizing `unique_ptr` and `shared_ptr` instead, ensuring our program remains free of memory leaks even if exceptions occur. This significantly reduces the complexity and frequency of debugging during game development.

Low coupling and high cohesion

In our project, the dependency between modules is low. Each module interacts with others through a stable interface, without needing to know the internal implementation of the other modules. There is also no existence of friends classes or public fields in our code, fields are all being accessed or mutated through accessor or mutator. Additionally, we maintain high cohesion by adhering to the Single Responsibility Principle. More details see the “Resilience to Change”.

Resilience to Change

To achieve high cohesion, we implement our base classes using abstract classes. Concrete classes inherit from these abstract classes, so when we need to add new functionalities, we only have to introduce a new derived class. Here are some examples that illustrate our code’s resilience to change.

For now, we only have text display which basically we call class `Cell`’s `print()` method from class `Floor` to print different kinds of elements that could be contained in a cell such as wall, floor tile, passage, and doorway. If we want to add a graphical display, we don’t need to change our current code since we already have a frame on how to print different elements. Instead, we could simply add a new class called `GraphicalDisplay` and make class `Floor` aggregates `GraphicalDisplay`, hardly any change will be made to our current code.

Moreover, it is possible that this game introduced more races to players, and we can perfectly fit in these new skills to our current Player without changing much code, reasons is as follows. We have written the combat() function for the basic player and enemy. For certain player or enemy races with special abilities, like Vampires being allergic to Dwarves and losing 5 HP instead of gaining, we just need to use function overloading to write a separate combat() function for them. If we want to add a new race or skill to player, all we need to do is to use function overloading and write new `combat()`. By doing this, we avoid changing existing code.

We also might, in the future, add equipment to players such as clothes or weapons that make them stronger. We can add this feature by using the same strategy as potion's decorator design pattern and make class `Equipment` a subclass of `Item`. In that case, the only change we need to make to our code is adding a subclass to `Item` and add one more `Equipment` pointer in `Player`'s field, and of course, everything else remain unchanged.

We adhere to the Single Responsibility Principle by dividing our project into multiple classes, each responsible for a specific aspect of the project with its own distinct functionalities. This approach minimizes module dependencies during implementation. By ensuring high cohesion and low coupling, we have achieved the goal of resilience to change.

CC3K specific question

2.1 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 races?

Answer. First, we establish a class named `Character` that serves as the base class for both `Player` and `Enemy` classes. This allows us to store common fields like `HP`, `Atk`, `Def` and common virtual methods such as `move()` into the base class, which reduces duplication of code. This inheritance hierarchical structure is mirrored for each race, like drow or vampire, by creating individual derived classes under `Player`. In this setup, adding a new race is not overly complex; We simply need to add a new derived class, override different methods as necessary, since common fields and methods are already stored in the base class.

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

Answer. The implementation of players and enemies is similar, with the main difference being that the player's race is determined at the start of the game and is generated only once, whereas enemies are randomly generated, with both their number and races being unspecified. We will utilize a similar mechanism of random generator as the provided example. Additionally, I have added a private field called `isHostile` to the

enemy class to determine whether specific enemies, such as `Dragon` and `Merchant`, are aggressive. Another field, `isAlive`, is used to check if an enemy is still active in the game. This contrasts with the player, for whom such a check is unnecessary, as the end of the `Player`'s life directly results in game over.

2.2 Question. How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

Answer. In implementing the abilities of enemies, I utilized `attack()` and `move()` methods, where `move()` is used to simulate the movement of the enemy, and `attack()` represents the aggression towards the player. This is also similar to the implementation for the player. I planned to use dynamic casting to determine the enemy type to implement various abilities at first, but I chose to use function overloading to write combat() finally, which improves the resilience to change. (details above)

2.3.1 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?

Answer: I can use a decorator pattern to model the effects of temporary potions. Using a decorator design pattern for `Potion` class with an object called `Blank` as the core object and decorators including Restore Health, Boost Atk, Boost Def, Poison Health, Wound Atk and Wound Def. In the `Potion` class, we should write several methods such as `usePotion()`, `attachNext()`, `setPotion()`; reasons will be explained later in this paragraph. In the `Player` class, we should include a Potion pointer called buff which records the potions taken by the player on a particular floor. Therefore, whenever a player is able to obtain a potion, no matter what kind of potion that is, it will be attached to buff in the Player class using the `attachNext()` method and set the latest potion obtained as the "top" of the buff using setPotion method. Finally, whenever a player tries to attack an enemy, we will loop through its buff along with usePotion to evaluate its final value of HP, Atk, and Def.

2.3.2 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?

Answer: We will first implement a class called `Item`, then we will implement two classes called `Potion` and `Treasure` which they inherit from Item class. Inheritance means both of `Potion` and `Treasure` shares fields and methods from `Item`, (such as coordinates, display by a single character etc.), which is logical since their behavior are similar (both are generated on floor and collected by the player within certain range). Therefore, if I implement a generation method in Item class, both `Potion` and `Treasure` class would be able to use it. Using such technique, whenever I want to add a potion or treasure, I only need to adjust the generation method from Item class instead of changing code in both `Potion` class and `Treasure` class.

Extra Credit features

Usage of Smart Pointer

We have used smart pointers throughout the entire project to avoid the use of `new` and `delete`. The advantage of using smart pointers is that we don't need to track all the memories that go out of scope, since it would be very tedious to do in a large project like this. The key challenge of using smart pointer is our unfamiliarity with it, since we just learned this concept recently, so we need to make sure we fully understand how smart pointers actually work before applying it to our code, which took us a period of time.

Final Questions

1. 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?

Throughout the project, one of the lessons we learned is the importance of establishing connections between each group member's idea, where communication plays a big role. Since a large project requires dependencies and inheritance between different classes, person A's code may require information from person B's code, so they need to

communicate between each other to make sure the logic between one's code and the others is correct and reasonable. One other thing we learn from doing this project is time management. Since every group member has different schedules, we need to come up with a plan that matches each other's time slot. Thus, we all need to somehow sacrifice our daily routine, instead, putting more effort on our project. We learned that the biggest difference between team project and individual project is that team project needs everyone to participate and adapt to each other's personalities and habits, with a detailed plan we can follow and provide support to each other whenever needed.

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

One of the things we would definitely do if we had the chance to start over is to test every group member's code before combining together, since debugging will be considerably easier when dealing with a relatively small amount of code compared to the code for the entire project. During our project, we hardly did any tests on the code that we wrote before combining them altogether, we basically went through the code and made sure there's no compiling errors. As a result, we spent a considerable amount of time debugging when we try to run our combining code. For example, a bug that occurs in the controller class may potentially indicate an error from a leaf class. Depending on how "far" between the origin of error and where error occurs, our debugging time may increase or decrease. Overall, there were countless bugs in our code and we would have spent significantly less time if we tested and debug first before combining codes together.