CS246 Spring 2022 Project – CC3k+

Final Documentation

Alex Lin / David Wei / Hilbert Yang

Introduction

The project is called ChamberCrawler3000+, a more approachable version of the rogue-like video game Rogue, implemented and designed by David Wei, Alex Lin, and Hilbert Yang. After the four-day design period starting on July 12th and ending on July 15th, ChamberCrawler3000+ was implemented successfully after ten days of coding and debugging starting on July 16th and ending on July 26th. The project possesses all the features and functions, listed in the CC3K+ requirement document.

Overview

Our team's endeavour is the creation of ChamberCrawler3000+, a more user-friendly version of the rogue-like computer game Rogue. The board is 79 columns wide and 30 rows high, and it has five chambers on it, each of which represents a floor in the game. It has a total of five floors with five of these boards. In order to go through the floors and win the game, the player will control a player character who will kill monsters, pick up buff potions, and gather coins(called gold piles in the game). The player can choose from one of four races for their character: human, dwarf, elf, or orc. Each race has a different hitpoint, attack, defence, and special trick. Additionally, different types of opponents will appear, such as merchants, dragons, phoenixes, werewolves, trolls, and troll-like creatures. Each of these opponent types has distinct stats, just like the player character.

Furthermore, dragons and merchants both have unique features that add to the game's complexity and excitement. Indeterminate chambers will generate random enemies. To simulate the randomness of rogue-like games, the adversary type and chamber selection will be based on the given likelihood. Within one block of the player's present location, the player

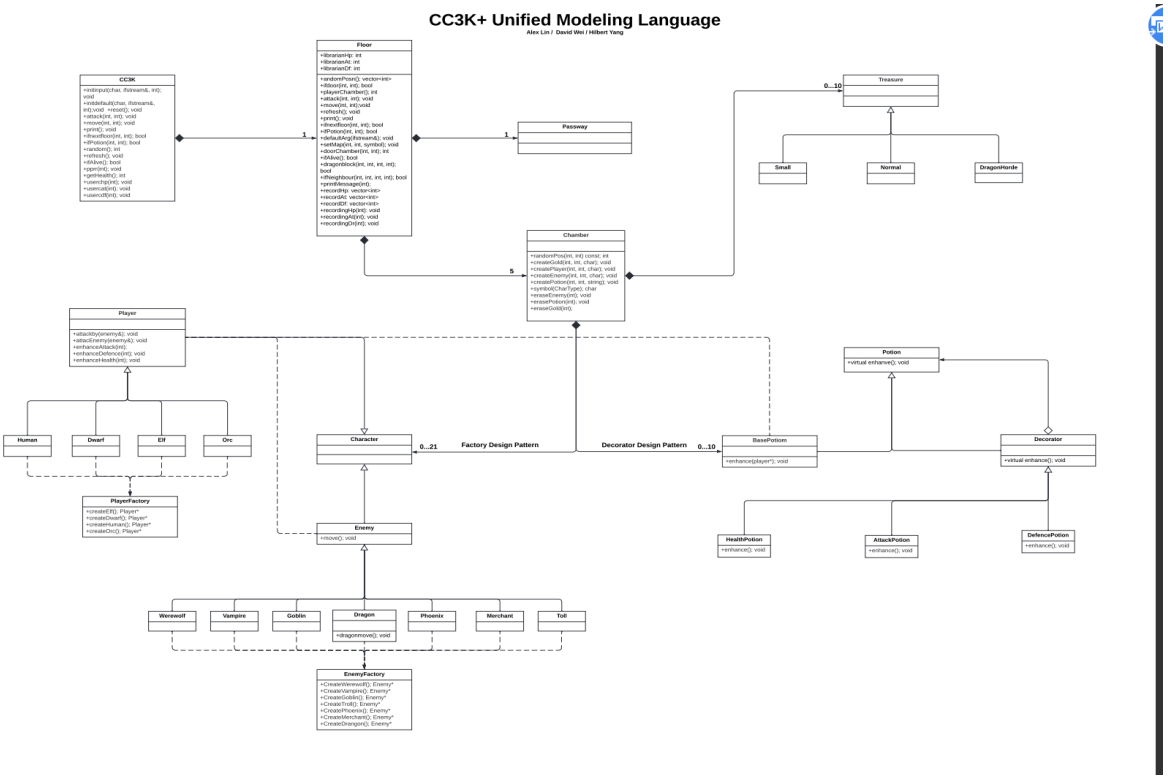
will attack the enemy, who will then strike back. The assault and defence will be used to determine the damage done. Additionally, the player will only absorb half the damage if they are wearing the barrier suite. The player may utilize potions to boost the hitpoint, attack, and defence using random potions that spawn across the game.

The player cannot discern if a potion is positive or negative before using it and some can also debuff the player's stats. The floor's stairways will similarly appear at random and they are always invisible until the player obtains an enemy's compass. When the game is over, the player's score is determined based on the gold they have earned through overcoming foes and gathering treasure.

Last but not least, the game board can also be altered by the player. This will be designed and implemented by our group using C++ and Object Oriented Programming ideas. More details are shown in the design part.

UML

(the clearer version of UML :file:///Users/davidwei/Desktop/UML%20Final%20(1)%201.pdf



Design

To make the project more sophisticated and able to utilize the functions and features easily, we used various techniques to solve the design challenges in the project.

Compared to the assignment that students normally do, CC3K is a relatively huge group project and the biggest challenge for us is to divide the whole project into different parts (i.e. making different classes) and each class should have its own function and feature.

CC3K class serves as the game's container and top class. Only the method in this class will be called directly by the client. The class contains a Floor pointer and a few methods to deal with different situations(i.e. has a default argument or needs to generate the map randomly). If there's no default argument, the map(i.e. a 2D vector used to print the image of the floor) will be pre-configured or modified randomly. The user initialization,map initialization, refreshing the floor and score calculation are all tasks that fall under the purview of the CC3K class.

The Floor class has a composition relationship with the CC3K class and contains all the information and the features on each floor including a vector of Chamber pointers and Passways, the player's current health and the number of enemies in each chamber etc.

The enemy, treasure, and potion vectors are then included in the Chamber class. Originally, the Floor class was created to house all of these vectors, but some modifications have been made to make the design simpler and the overall structure more comprehensible. The Floor class has three boolean variables: ifAttackMerchant, ifBarrierAppeared, and

ifCompassCarried that will affect the five chambers on a floor. For instance, if a merchant is attacked, all other merchants in each chamber on the floor will suddenly be hostile. As a result, when we initialize the map for the following floor, accommodations will be made in case the play strikes a merchant on this floor. The Chamber class has the methods RandomPos, Createplayer, enemy or gold. A random location inside a chamber will be returned via the RandomPos and different Create methods. Characters and objects in a chamber can be formed with the help of the Create methods while carefully adhering to the corresponding possibilities. To minimize code duplication, specific treasure classes are created via inheritance.

Resilience to Change

In addition to these fundamental classes that control the game, we also use two design patterns to assist with the implementation of Characters and Potions. Two classes, Player and Enemy, are derived from the abstract Character class. Human, Dwarf, Elf, and Orc are all subclasses of the Player class, which is derived from a class called PlayerFactory. The methods for constructing these objects can be found in the PlayerFactory class. Programmers may utilize virtual constructors to add additional characters to subclasses thanks to this use of the Factory design pattern. Also, the Potion class's design benefits from the Decorator design pattern. Three decorator potion classes could theoretically embellish the BasePotion class and a BasePotion pointer in the player class links the BasePotion class with the Player class. Accordingly, the classes of Decorator and Potion are both abstract. The player can therefore have the effects of potions added to them at run-time by the software.

Answers to Questions

Q1: How could your design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes?

Even if each race may be generated in a variety of ways, we still need to find a means to make the process more feasible and approachable. The Factory Method design pattern offers an interface (PlayerFactory on the UML) for producing objects, and it gives the subclasses the freedom to determine which race object to produce. As a result, we opted to utilize this pattern to enable the generation of each race. The Factory design pattern, which we will employ, allows us to build each race without having to specify their specific classes, which increases the program's adaptability. So, by taking this technique, we could easily manage situations where classes were to be added or removed.

Q2: How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

Since various foes can also require the Factory design pattern, in general, the way we want to utilize to manage to produce them is identical to its equivalents (i.e. each race). Nevertheless, we did make certain adjustments in light of various opponents' abilities and characteristics. Dragons, for example, have several unique characteristics that other characters do not have, such as the ability to defend a dragon hoard as soon as it spawns at random. In addition, we need to create a few fields and methods for the player abstract class,

such as the Boolean value ifBarrier that indicates whether the race has a barrier, the potion that the race might acquire, and some methods to get the HP, attack points, defence points, and the amount of gold, among other things. In contrast, the abstract enemy class has just one function that causes the adversary to travel at random.

Q3: How could you implement special abilities for different enemies. For example,
gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?

Our factory method design pattern makes it simple to build various special abilities for various opponents. You can see from the UML of our project that the Factory design approach will be used to build both the Player and Enemy classes, making the creation and modification of the objects quite simple. We can simply add a virtual ability method to the abstract creator class (i.e., the enemy class) and override the ability in each concrete creator class, which is the same way we used to provide distinct abilities for various opponents (i.e Goblin, Toll, Merchant etc.). For the aforementioned example, if we wanted to give the greedy goblins a gold-stealing ability, we could simply override the ability method in the goblins class and do some subtraction operations on the player's gold int in accordance with the relevant rule.

Q4: What design pattern could you use to model the effects of temporary potions (Wound/BoostAtk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?

The decorator design pattern is a good choice for assisting with potion implementation. The effects of potions are applied to the player at run-time, hence various

potion objects must be made and removed in accordance with the player's movement. Fortunately, the Decorator design pattern enables programmers to add functionality to an object at run-time as opposed to the class as a whole. Thanks to the pointer in the decorator class, many similar potion objects may be arranged in a linked list. Therefore, the effects the player will experience are determined by how the potion objects connect in the list; as a result, the computer does not need to explicitly monitor how much of each component the player has consumed on a floor. After accessing the next floor, we must make fresh potion objects and, if required, clean out any outdated ones in order to counteract the effects of the attack and defensive potions.

Q5: 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 hordes and the Barrier Suit?

To prevent code duplication, the production of treasure and potions will be accomplished using inheritance. It is decided, for instance, that the Treasure base class will have three fields: the x and y coordinates on the map, as well as the quantity of gold a Treasure holds. As a result, the common fields and methods may be shared by all additional Treasure classes descended from the base class without duplicating any code. In addition to being reusable, the decorator design pattern used to construct potions makes it simple for programmers to add new functionality. No new class will be developed as the Barrier suit because it frequently occurs in games as a dragon horde. The information about whether or

not the dragon horde is a Barrier suit will instead be stored in a new boolean type property called ifBarrier that will be included in the dragon horde class.

Final Questions

Question 1: What lessons did this project teach you about developing software in teams? What lessons did you learn about writing large programs if you worked alone?

The first lesson we learn is that good communication between team members is the key to smooth project development. Since we first have a plan of attack, we try to stick to the plan and do the assigned part. However, we soon learned that almost everyone encounters problems that are not easy to solve. Therefore, good communication is required between teammates to either inform each other if one's part is not ready or help a team member solve the problem.

Moreover, another important lesson we think is worth discussing is the importance of debugging, or rather, making a reasonable plan for debugging. At first, we never thought that debugging would take this much time. We believe the coding is the hard part, so we dedicate more time to doing the coding. Nevertheless, our code did not even compile on the first several runs. Then bugs started to pop up when it was finally compiled. After we fic one bug, other bugs pop up because of the changes and accommodations we implement for the previous bug. Therefore, if we ever have a chance to work on a group project, we will invest more time in debugging.

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

The first thing we will do differently is starting compiling and debugging as quickly as possible. As we reflected, the time-consuming part is the compiling and debugging part. We will arrange frequent meetings and group debugging where we think of solutions to bugs and problems together.

In addition, we will make sure our code at least compiles and then combine them. We first thought that since we each were in charge of different parts, there was no way for each of us to test our code. Now we will write some simple main functions and compile them. This way, much time will be saved when our code is combined.