CC3K Final Design Report

Team Bald Warriors

Introduction

This report presents our design process of ChamberCrawler3000. We tend to focus on building our game by applying fundamental OOP concepts and C++ concepts learned in CS246 to develop a functional, organized and memory safe program that on the other hand minimize coupling and maximize cohesions.

Overview

With the use of the Object Oriented Programming ideology. We structured this project into various components, each has separate and independent responsibilities, and each provides only functionality for its own duties.

Subject is a super class to all objects on the floor. Subject class keeps track of information such as positions, object type and observers. All are necessary properties that any object would have, despite their type. Subject is inherited by three subclasses, **Tile, Character** and **Item,** each representing a general type of objects on the floor.

Tile is a subclass of **Subject**, representing the floor tiles of the dungeon map. **Tile** class is not a very informative class and it only keeps track of the type of a specific tile, such as chamber floor, walls or tunnels.

Character is a super class to all character objects, and is a subclass of **Subject. Character** keeps track of character type, health point, attack strength, defence strength and max health point. These are vital properties that each and every character would have, despite camp and race. **Character** class is inherited by two subclasses, **Hero** and **Enemy**, which represents the two major camps of all character races.

Hero is a super class to all hero races, and a subclass of **Character**. **Hero** class keeps track of information unique to hero races such as gold earned. Moreover, **Hero** class provides methods for a hero to attack enemy objects of specific races, to defend general **Enemy** objects (the implementation of combat mechanics will be addressed later in this report), to use **Potion** and to pick up **Treasure**. **Hero** class is inherited by 5 different subclasses, each represents a particular hero race, they are **Shade, Drow, Troll, Goblin** and **Vampire**. Structure

wise, these subclasses are no less the same, with different statistics. Thus we will not expand further.

Enemy is a super class to all enemy races, and a subclass of **Character**. Like the **Hero** class, **Enemy** class keeps track information and provides methods that are unique to enemy races. **Enemy** class keeps track of whether an enemy is neutral, to separate permanent hostile races and occasionally hostile races, **Merchant** in particular. **Enemy** class provides methods to attack hero objects of specific races, to defend general **Hero** objects. **Enemy** class do not interact with **Item** class, thus, they have less methods compare the **Hero**. Like **Hero**, **Enemy** class is inherited by 7 different subclasses, each represents a particular enemy race, they are **Human, Dwarf, Elf, Orcs, Halfling, Dragon** and **Merchant**. They will not expand this section.

Item is a super class to all item objects on the dungeon floor. **Item** class keeps track of item type and provides a virtual effect method that applies the effect of a particular item on a **Hero** object. **Item** class is inherited by 2 subclasses, **Potion** and **Treasure**, each represents a general type of item that can be used on the floor.

Potion is a super class of different potion objects. **Potion** class keeps track of information such as potion type and whether a potion's effect is revealed by the player. **Potion** class provides an effect method that will apply an potion effect to an **Hero** object that uses the potion. Either magnifies or minifies a **Hero**'s ability. **Potion** class is inherited by 6 subclasses, each represents a potion of particular effect, they are **RestoreHealth, BoostAtk, BoostDef, PosionHealth, WoundAtk, WoundDef**.

Treasure is a super class of different treasure objects. **Treasure** class keeps track of information such as treasure type and whether a treasure can be picked up, a **Dragon Hoard** that is guarded by a living **Dragon** cannot be picked up. Similar to **Potion**, **Treasure** also provides an effect method that will increase the amount of gold of the **Hero** object that picks it up. **Treasure** is inherited by 4 different subclasses, each represents a treasure of particular size, they are **SmallHoard, NormalHoard, MerchantHoard, DragonHoard**.

When implementing objects on the dungeon floor, we chose to use a large inheritance tree to group classes that share common properties together to reduce redundant code. We treat each class as an independent object with independent focus, to reduce coupling and increase cohesion.

Floor class represents the dungeon floor and serves the purpose of organizing different objects on the dungeon floor, controlling objects' movements and interactions between each other. Interactions and movements are relative to

positions on the floor. Thus **Floor** keeps track of 2d vectors the same size of the actual floor map of floor objects as a virtual map. This idea came from the implementation of Conway's Game of Life. In particular, **Floor** holds a **Hero** object, which is our player character and 4 virtual maps, for **Tile** objects, **Enemy** objects, **Potion** Objects, **Treasure** objects, as they have different behaviours and are hard to generalize. **Floor** provides a spawn method that initializes each floor object on each of the 4 maps, with specified probabilities. **Floor** also provides methods for different **Hero** movements and interactions, such as move in a particular direction, use **Potion** at a particular direction, attack an **Enemy** at a particular direction. As well as a turn method, that takes care of the movements of the **Enemy** objects on the floor, which is automated from the player's point of view. There are many details being taken care of in the **Floor** class to accommodate special characteristics of different classes and will not be further discussed in this section.

TextDisplay is an observer class. **TextDisplay** keeps track of the floor map and hero statistics that are used for game display. **TextDisplay** provides methods to retrieve information from floor objects and update displays as well as a method to output the display through the outputstream.

The final execution of our program is done by the **Main** program. **Main** will initialize a **Floor** object and is responsible for taking in player's commands and executes corresponding operation through **Floor** methods. All of our game operations are designed to be encapsulated in **Floor.**

Design

In the middle of the design process, we met several challenges and managed to solve them using design patterns.

Combat (Visitor Pattern)

The combat system is one of the more complicated systems we have designed in this project. Different races have different attack mechanics, elf attack twice against most enemies and vampires restore 5 hp through each success attack. Moreover, combat between different races can be unique, orcs deal 50% extra damage to goblins and elf only attack once against drow. Thus, to accommodate these features, attacks cannot be done between the **Hero** and **Enemy** super class but rather between the specific subclasses. However, for the ease of management, we store these particular subclass objects in their super class pointers. Thus to resolve this challenge, we decided to use the visitor pattern.

Hero class has a virtual defend method that takes in an **Enemy** subclass and calls its attack method on the **Hero** itself. The **Enemy** has a similar virtual defend

method. Moreover, each of **Hero** and **Enemy** has different virtual attack methods for each other's subclasses, where it deduces hp of the subclass object. In such a way pof design, if we have a **Hero** subclass object and an **Enemy** subclass object stored in their super class pointers. To incur their special combat mechanics, all we need to do is to call the defender's defend method on the attacker. The virtual method will help detect their subclasses.

TextDisplay(Observer Pattern)

Displaying texts is an important part of the game, since all the user sees is the text display. Thus having an organized text display system is vital to our game. Thus, we had chosen to use the observer pattern. We break down the game display into blocks on the map. Each block either displays the corresponding floor tile (if nothing is on the tile), a player character, an enemy with specified race, a potion or a treasure. These are all subclasses of the **Subject** class. Thus, we write and implement an Observer class with a notify method that takes in a **Subject**. We also made the **Observer** class a component of the **Subject** class in the form of a vector of pointers and wrote a notifyObservers method for **Subject**.

In the basic version of this game. We Implement a **TextDisplay** class that inherits from the **Observer** class, the focus of this class is described in the overview section. Upon notification, **TextDisplay** will update the corresponding string to the position of the **Subject** object on the display map. In the spawning phase, we attach the **TextDisplay** object to each floor object. At the end of each turn, we will have each floor object to notify their observers of their new status, and the display will automatically update.

Resilience to Change

Character

Most of our classes have decent resilience to changes. We consider the change to character classes in two cases.

1. Change to the basic properties of the character. For example, a change in the health of a race of hero character, or a change to the atk value of an enemy character, these can be easily achieved by altering the field of the **Hero** super class. Since we follow inheritance strictly, each layer of superclass and subclass are separated clearly and only handle its own fields. Thus this change is not going to harm our code much.
2. Adding/Removing a raceType/enemyType. This requires us to add/remove the specified subclass while also changing all overloaded attack functions on the

other party(changing a raceType requires the change in enemies' attack function).

Item

Second, the item classes are having a pretty good balance. The potion class has four functions, and three of them only interact with its own properties. The **effect** function takes in a hero parameter and applies the potion/treasure effects on the player character. For any possible change(that makes sense), we consider it with two approaches:

1. Change in potion effects. Since the effect function takes in the hero as a parameter, it has access to the player character's basic properties(since the hero provides access and mutator functions). If there is any change in the specified fields, we would only change the effect function and no other change on any other functions/classes. If any new potion is added, we can simply add another subclass to the potion class. The same thing applies to treasure classes. Currently, we only have effect function (treasure effect, add gold to hero) except dragonHoard. We manage to put the dragon hoard associated with a dragon, with a field to keep in track the life of the dragon. If it dies, the dragon notifies the hoard. Thus if there is any change to our hoards, we would follow the steps of mutating our potion classes. The dragon hoard follows pretty much the same, except if there is any change after the death of the dragon. This can be done with mutating the notify death function of the dragon.
2. Change with hero property. If any basic properties of the hero changes, we would just alter the hero, which is not going to affect our potion class. Consider if there is a change in drow, for example, an alteration on the scale of potion effects, we have a **getScale** function on in the class, which gets the scale of enhancement in the potion effects. Thus, we can accomplish this change just by altering the getScale function.

Floor

We are not randomly generating the map, but reading from a file to draw the map is fine. However, if there are further changes to the map(either size or more tile types), the floor class may face a large change. But on the other hand, the floor class does not overload much character class and item classes and has a high cohesion with low coupling. Thus a change in those two classes is not affecting the floor much.

Observer/TextDisplay

Our observer has a high cohesion and low coupling. It only focuses on the change in the text display and does not share any information with other classes. One thought is make text display to visual display. This is an easy implementation since there is not much to change other than some coordinates and tile displays.

Another change that could alter is displaying the action. Contributed by our high cohesion, this does not require any change to the observer class. Since all our mutation to action is handled by the **floor** class and **hero** class. On the other hand, if we add more classes that require a display and need to notify the observer, we would consider adding those classes under the **Subject** class. Since the notify function takes in a subject as its parameter, adding new classes under the Subject allows us to finish the change without any effect on the observer class.

Answers to Questions

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 races?

We have a superclass named “Hero” with all basic properties such as health, atk, def, with some abstract functions like attackEnemy, move, etc. Then for each race, we design a concrete subclass with the race’s unique properties. We believe this solution makes adding new races easier, since we only need to manage a new subclass with minor adjustments. It would be possible if we implement a decorator design pattern which simplifies future adjustments, which is something we are still thinking on.

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

We generate enemies by random placing them on our floor tiles according to the given probability. This gives enemies an equal chance of spawning on any given valid tile, which is simpler than generating them in different chambers before specifying the tiles. We were using rand() and srand() in <cstdlib> to generate random numbers.

This generation is similar to the generation of the player character. Since we have the movement of all characters controlled by the floor class, generating using similar functions makes the program organized.

Question 3: How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

Since the special abilities apply while an enemy attacks, we incorporate it in our attack and defend strategy. We use a design pattern similar to the visitor pattern. where Enemy class has an attack method that takes in a Hero pointer, and calls the Hero's defend method on this enemy. Hero class overloads different defend methods for different enemy races, that's where we modify the attack effect base on each enemy races' special abilities. We do the same for Hero(player character) races.

Question 4: The Decorator and Strategy patterns are possible candidates to model the effects of potions, so that we do not need to explicitly track which potions the player character has consumed on any particular floor. In your opinion, which pattern would work better? Explain in detail, by discussing the advantages/disadvantages of the two patterns.

In our opinion, the strategy pattern works better for this class, as we add a double dispatch function to the potion and hero class; then change private fields of our 'hero' with respect to the effect of the potion. This idea views potions as algorithms that mutate the attributes of the player. We have the view that strategies tend to be easier to implement, and requires less coding, saving us time to do other important tasks. At the beginning of the game, after selection of character race, we record the character's default properties, and reset some properties (like Atk or Def) when the 'hero' enters a new floor. This could be difficult for the strategy pattern as it is quite hard to reverse the effect of strategies.

Additionally, we may face some difficulties working on DLCs for potions. For example, if we add a permanent boost atk potion, the effect may conflict with the reset function at the beginning of each floor. Additionally, there could be potions with time limits, or with special effects that do not mutate the player's attributes, but rather do something else. This creates exceptional challenges towards the strategy design pattern. We might change to decorators in the future.

Question 5: 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?

Our structure stores potions and treasures with two 2d vectors. This results in the fact that we only need to randomize the position and type of the potion/treasure. Since the probability of generation functions are different, there is not much duplicated code of these. We considered the use of template functions to reduce duplicated code, but in our implementation of the structure, there is not much we can do to make a template function.

Extra Credits Features

During the implementation of our project, we choose to follow the RAII idiom to ensure the memory safety of our program. In our project, we did not use any raw pointers or instance of arrays, since we believe that manually deleting heap allocated memory in such a large project has high risk of incurring memory leakage. Instead, we used shared pointer and vectors, both following the RAII idiom and will automatically delete any heap allocated memory after the entire program returned. We are aware that there are downsides to using shared pointers, such as cyclic referencing, so we are very careful in ensuring that no such issue occurs. During our testing phase, no memory leak is discovered at all.

Final Questions

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

1. Github is a great tool while developing in teams.

   Since we did not work in a professional team before, this is a chance we explore our way of working in a team. We all heard Github before, but part of our group does not have practical experience with Github. This project gives us an opportunity to use this great application to enhance the quality and efficiency of our project.

2. Documentation is important throughout developing a project

   Although this project is not too large, and we all put effort into reading and fully understand each other's parts, we still realize the importance of having formal documentation. We spent a lot of time trying to comprehend other members' work while they are sleeping, while this can be done more efficiently with documentation and fully commented code.

3. Structure graph(UML) need to be considered deeply before writing the code

   When adding or removing a class, we may need to change some functions in other classes. if we are able to think fully and objectively, we could avoid some changes and save us some valuable time to look for bugs and even develop DLCs. However, due to a lack of knowledge in developing such a big project, we do not have the ability to foresee all difficulty in writing the code,

which this project teaches me to think deeper and take it more seriously when designing the structure of a project.

4. Importance of having good teammates

   Although it is important to have a strong personal skill set, it is also significant to have good team members while working in a team. Imagine you wake up in the morning and realize all the bugs you found last night disappear(fixed by teammates). There is no better feeling.

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

Speed up the process and add more content to the project

Our plan is kind of interrupted by other exams(which shouldn't happen). We would expect to speed up the process of basic classes and build more features with our creativity.