CC3K Design Document CS246

By: Jatinvir Matharoo, Marzuk Mashrafe and Jennifer Zhu

Introduction

For our final project, we decided to implement CC3K. In this document, we will describe how we have built our program, the design choices that were made, what we have learned throughout the entire process, and what we could have done differently.

Overview

Initially, when we first set up our program, we tried to classify different parts of the game into different sections. We created a Floor class that includes everything to do with the Board, and then have subclasses for elements of the floor, such as Chamber and Cell. We also have a class to keep track of the statistics of the player, called Stats under which we have the BaseStats subclass and the Potion subclass. To implement these classes, we decided to use the decorator design pattern, which is explained more in our Answers to Questions. For the players, we created an abstract Player class under which all the different players, such as Shade, Drow, Vampire, Troll and Goblin are subclasses. Similarly for the enemies, we first have an abstract Enemy class, and then the various enemies such as Human, Dwarf, Elf, Orcs, Merchant, Halfling and Dragon. To implement both Player and Enemy (combat in particular), we utilized the Visitor design pattern, which is explained more in the Answers to Questions. The factory method design pattern was also used to create all of these enemies. To implement the factory method, we added the Creator and ConcreteCreator classes. We describe more of the reasoning for using the Factory Design method in the Answers to Questions. We also added a RandomMove abstract class, which has two subclasses: RandomMoveOneBlock and RandomMoveZeroBlock. The following section will go into more detail about our design.

In our design, we have utilized low coupling and high cohesion, as we have separated many different areas of our code depending on if they're related or not. For example, the different enemies are related to the Enemy class, which is why we have them connected, but since the potions enemies are not related, we have them separated and they do not have a large connection.

Design

Floor (floor.,h)

To create our floor plan, we have a Floor class, with a Chamber object as well as a Cell object. Each Cell object keeps track of an (x,y) coordinate and what type of object is in that location. The purpose of the Chamber object is two-fold. One - it ensures enemies know what chamber they belong to and two - allows random generation to occur (further discussed below).

The Floor class is used to generate the floor with Cells and Chambers while also placing the required items on the floor such as Cells, Chambers, Enemies, Gold, and Potions.

In the Floor constructor, it goes through the entire floor plan that is passed in and sees what and where the potions, enemies, staircase, etc are. This allows us to understand where everything is and see what type of items we are dealing with.

Also in this class, there is functionality related to the Enemy and Player. When the enemy attacks a Player, the program sees what type of Enemy is on the floor at that cell, and then applies the appropriate strike functions so that the Enemy strikes a Player, from which the action message also displays on the screen. The Floor class also moves the enemies across the floor. The Floor class will check to see what is located on a cell and its neighbouring cells, and then move the Enemy after determining that the new cell is a valid location to be in. For the Player, the Floor class moves the Player to new cells and also adds functionality for the Player to collect Gold. Implementing this in the Floor allows the program to see the contents of the neighbouring cells and see what needs to be done based on that information. Floor is the sole class in the program responsible for location management including randomized location generation. Thus it maintains high cohesion and low coupling.

Cell (cell.h)

We have a cell subclass as each cell itself is part of the Floor and makes up a majority of what you see on the display. This class has a constructor and creates the individual Cell based on the coordinates given and what needs to be placed at that position.

Chamber (chamber.h)

Since there are multiple different Chambers, our Chamber class contains the individual cells that a chamber has, as well as the id of the Chamber. The id of the Chamber allows us to identify which specific Chamber items are in.

Stats (stats.h)

Stats is used to apply and modify the various statistics for the Player, such as defence and attack, as well as modify those values with the different types of potions that are in the game. This is an abstract class consisting of two pure virtual functions: int calcAtk() and int calcDef(). Under stats there are two subclasses: BaseStats and Potions.

BaseStats (baseStats.h)

The BaseStats class is used for initializing the statistics for the players, such as atk and def. It also overrides the calcAtk() and calcDef() functions from stats.h, which return the base atk and def for the player, respectively.

Potion (potion.h)
The Potion class is an abstract class which acts as the decorator for all the potions that are required. Inside of Potion however, we have calculated the atk values and the def values for the players, when utilizing a potion.

potionBA, potionBD, potionWA, potionWD (potionBA.h, potionBD.h, potionWA.h, potionWD.h)
Inheriting from the potion parent class are potionBA, potionBD, potionWA and potionWD. These are concrete subclasses in the decorator pattern. The potionBA class is the boost attack potion, which will boost the players atk statistic by 5. The potionBD class is the boost defense potion, which will boost the player's def statistic by 5. The potionWA class is the wound attack potion, which will decrease the player's atk statistic by 5. The potionWD class is the wound attack potion, which will decrease the player's def statistic by 5.

Enemy (enemy.h)
The enemy is an abstract class designed for an enemy character. Enemy is a part of a visitor pattern along with Player class. Visitor pattern is used for the player strike functions. There are methods that are used for a general enemy, such as the strike or movement functions.

The fields x and y are used to represent the enemy's position on the floor. The field hp, atk and def are to represent the health points, attack and defense value for each of the enemy. The goldAmt field is used to represent how much gold the enemy has. We have also introduced many different methods in the enemy class.

There are also two functions used to calculate the gold properties for the enemy: dropGold() and getDropsPile(). The dropGold() function is used to determine what type of gold the enemy will drop, whether it is small or normal, and has a return type of int. The getDropsPile() is used to determine if the enemy drops gold or not. If it does it will return true otherwise false.

For the movement of the enemy, there are also two functions: getMovement() and moveTo(int x, int y). The getMovement() function is used to randomly move the enemy around the floor. The moveTo(int x, int y) updates the location to which the enemy must move to, setting the x coordinate to the x integer passed in, and setting the y coordinate to the y integer passed in. These two functions allow us to move the enemies around the floor.

The modifyHealth(int diff) is used to modify the enemy's hp value.

There are then various strike functions depending on the player the enemy is striking. The calculation given to us in the project documentation was used to calculate the damage applied.

Under enemy, there are subclasses for each of the different types of enemies. These subclasses override the required functions that will be different depending on the type of enemy.

The following classes are the different Enemy types.

Human (human.h)

For the human subclass, we have a dropGold() function and a beStruckBy function. The dropGold() function returns an integer of 4 as a Human has that much gold on it. The beStruckBy(Player &) function is used to calculate what the hp of the Human will be after it has been struck by a Player character.

Dwarf (dwarf.h)

In Dwarf, we have a beStruckBy(Player &) function that is used to calculate what the hp of the Dwarf will be after it has been struck by a Player character. Dwarf itself does not have any specific special features itself, and none of the strike functions needed to be overridden.

Elf (elf.h)

For the Elf enemy, we have overridden the strike functions in the Enemy class, as when an Elf strikes anything but a Drow, it strikes the Player twice. Since this is different from the average Enemy, this was required. To implement this special feature, we decided to essentially make the damage be twice than what it usually is against every Player other than Drow.

Orcs (orcs.h)

Since Orcs attack Goblins with 1.5 times the amount of damage, we have overridden the function where an Orc attacks a Goblin to implement this change.

Merchant (merchant.h)

The major difference implemented here was the amount of gold the merchant drops. Since a Merchant drops a merchant hoard, or gold with a value of 4, the dropGold() function returns 4 here.

Halfling (halfling.h)

For a halfling, the special feature it has is that it has a 50% chance of causing the Player to miss during combat. So essentially when any Player tries to strike a Halfling, instead of the Player always being successful, there is a 50% chance that the player misses. So in this class, we generated a random number, and if the random number was 0, the player would strike, otherwise it wouldn’t.

Dragon (dragon.h)
Since a Dragon is the only enemy that is stationary for the entirety of the game, our program does not move the Dragon while it is moving all of the other enemies, which is one of the major differences. The Dragon class also handles the gold

<u>Player</u> (player.h)
The Player class is an abstract class which is designed for the player character. It is a parent class to all of the different types of players: Shade, Drow, Vampire, Troll and Goblin. Player is the other part of the Visitor Pattern along the Enemy class. It also utilizes Visitor pattern for its enemy strike functions.

Since a Player can use many different types of Potions, we have methods that will allow the Player to use a particular function and have its statistics modified. One of these methods is useHealthPotion(int effect), which will add the health potions effect and either restore or boost the player’s hp. There is also usePotion(std::unique_ptr<Potion> potion), which will modify the players atk and def values for the Boost Atk, Boost Def, Wound Atk and Wound Def potions. We then have usePotion(int x, int y, Player *pc, std::string &potion), which will see if there's a potion at that coordinate where the Player is and have the Player use that function.

The score that should be displayed is included in this class. Depending on how much gold the player has, the score will be calculated.

There are then various strike functions depending on the enemy the player is striking. We used the calculation given to us in the project documentation.

The following classes are the different Player types.

Shade (shade.h)
This is the default player class, Shade. This class inherits the strike functions from Player, as they are the same, and has a function: beStruckBy(Enemy &e) to modify its hp from being attacked by an Enemy. We also calculated the score for Shade depending on the amount of gold it has.

Drow (drow.h)
The changes in Drow was to account for how each potion has its effect multiplied by 1.5 when used by Drow. So everytime a potion is used by a Drow, the statistics are multiplied by whatever effect that specific potion had by 1.5.

Vampire (vampire.h)
The special thing about Vampire was that it would gain 5 hp everytime it would strike any Enemy other than a Dwarf. When it strikes a Dwarf, it loses 5 hp. To implement this we

overridden all the strike function from Player, and modified it to handle increasing or decreasing the hp value for the Vampire each time.

Troll (troll.h)
A Troll gains 5 hp every time it takes a turn. So if the Player was a Troll, a function is called each time it takes a turn and it adds 5 hp to its total hp.

Goblin (goblin.h)
For Goblin, it's special feature is that it steals 5 gold when the enemy is slayed. So in our Goblin class, we introduced a function that will add 5 gold pieces to the Goblin's gold depending on if the enemy is slayed or not.

Creator (creator.h)
For Creator, there is the Creator, concreteCreator, potionBox and Gold classes as well as each Enemy class. We decided to use the Factory Method design pattern here as it allows us to create enemies on runtime depending on the floor. Creator itself is an abstract function, containing pure virtual functions to create an Enemy, Potion, and Gold.

concreteCreator (concreteCreator.h)
This class is a subclass of Creator and has the implementation to create an Enemy, Potion and Gold. There are two ways we decided to create an Enemy. For the first way, since there is a probability to each Enemy being generated, we created an Enemy based on that. For the second way, we created an Enemy based on the type found on the floor, such as "H" for Human or "E" for Elf. Potions also have either a probability and a type on the map, so a similar model was followed from creating an Enemy to creating a Potion.

RandomMove (randomMove.h,)
Since the player playing does not have control over where the Enemy moves on the floor, these classes allow us to do that. For these classes, we used the Strategy design pattern. The strategy pattern was used as we needed to determine which algorithm of the same function to use at runtime, which is where Strategy helped. To do this we have an abstract parent class called randomMove, which has a pure virtual function to move the enemy to an intended destination. Since we don't know if the enemy will be able to move there or not, we set up two subclasses, randomMoveOneBlock and randomMoveZeroBlock, which both implement the pure virtual function move found in the parent class.

randomMoveZeroBlock (randomMoveOneBlock.h)

This class overrides the move function from its parent class, but does not modify anything. This function will be called when the Enemy is not allowed to move to a Cell.

randomMoveOneBlock (randomMoveZeroBlock.h)
This class also overrides the move function from its parent class, but will move the enemy to a random new Cell that is one block away.

<u>Resilience to Change</u>

Our program is able to handle changes to the program specification. We put in a considerable amount of time towards the start of the project to design and plan a program that can accommodate change as easily as possible. However we do acknowledge that our solution is far from perfect and has a lot of room for improvement.

Let us Discuss the Floor class first. Floor is the biggest class in the program, It is responsible for everything to do with location. Everything from random location generation to keeping track of all player, staircase, enemy, potion and gold location is done inside of the floor class. Thus if any changes need to be made that are location related, it is most certainly going to be inside the Floor class.

Now for the Enemy and Player types, we used abstract parent classes to simplify the code as much as possible. By overriding only the differences, we make the code much shorter and easier to change. We employed the Visitor pattern to simulate combat. This way, we can further simplify and modularize the code. Now if we need to add a new type of character, we don't have to write strike functions for all the other classes, only just a few.

For potion usage, we implemented the decorator pattern as the changes made to the player stats are limited to a floor (except HP). By having the floor own the potions and applying them using the decorator pattern, we can control how the stats are manipulated and we can always reset the stats. Adding new potion types is also very simple. We only have to change the concrete creator. Unfortunately we did not have the time to move all potion responsibility from Floor class to the creator class.

The fact that Dragon is a type of enemy that doesn't move while other types of enemies move randomly is also accounted for using the strategy pattern. This is in order to simplify what move algorithm is used by what enemy. This allows new Enemy types to choose their move functionality. Furthermore, changes to the move algorithm of all Enemies is much simpler. Even more, it is relatively easier to add new algorithms to many Enemies with little change.

For the Gold functionality, we initially planned to have a separate dragonhoard class inherit from a gold parent class. However, that idea was scrapped due to the complication not being worth the little flexibility it provides. We ended up with one gold class that both acts like regular gold and dragonhoard.

Finally, our program can accommodate any floor plan that is 79 columns wide and 30 (25 if you exclude the stat display) rows high. This is due to the way the Floor class reads in a given (or not) file. The program determines the chambers dynamically rather than statically. As such you can play on cool boards that are not the default one. So if the floor plan specifications ever change, we do not have to change our code very much.

Answers to Questions

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?

Assuming this question is only for generating player classes, we have designed the system in a way that in order to add/remove new player classes, we only have to modify the main.cc file. Just by adding a conditional return of the new type in the createPlayer function, we can add a new player race to our game.

As for creating new Player classes, we simply have to inherit from the player abstract class and implement the changes we want for the player. We did end up with two different classes for Player and Enemy as we had planned.

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

The enemy generation is handled by the creator class. Using the Factory method design pattern, we can simplify the addition of new enemy types much easily. It is different from generating a Player character because it is randomized unlike the player character. Our implementation is very close to what we had planned for in the beginning.

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

For enemy attack abilities, we implemented them using the Visitor design pattern. Most of these abilities depend on which enemy is attacking which character race, and the Visitor pattern lets us use double dispatch to determine which enemy and character race are involved in an attack so we can decide which abilities to apply. We used the same technique for the player character attack abilities as well, since most of them also depend on which character race is attacking which enemy. If an enemy or player character has special features, then that function was overridden in the race class.

For enemy stats, we used an abstract enemy superclass with each type of enemy as a subclass. Each enemy type's constructor can then initialize the object with the appropriate stats. We used the same technique for player character stats, with an abstract player superclass and each race as a subclass. For the player's race attributes, we did inherit the hp, atk and def from another stats class. So since there are similar attributes between an enemy class and a player class, we decided to use similar techniques to implement them. The answer we had for DD1 is really similar to the answer provided here.

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.

Our answer for this question is mostly the same. We decided that the Decorator pattern would be more effective because we don't have to keep track of the types of potions currently active and how they stack together. In our code, we simply add new potions to the vector of potions inside Player, and leave Decorator to do the rest. If we had used the Strategy pattern, we would need to have kept some record of which potions were used so that we'd be able to select the right algorithm when needed.
The disadvantage is that the Decorator pattern requires more memory, and as we have learned is prone to memory leaks even with smart pointers. We had to modify the pattern to use a vector rather than a linked list for this reason. The advantage of the Strategy pattern is that we don't have to deal with memory and there aren't as many potion objects we need to handle.

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?

Both generation of Potion and Gold are done in the creator class. Their location is generated using the same function in Floor. This is exactly as we had planned for.

Changes from DD1 to DD2

One of the changes we made was removing the dragonHoard class. Instead of having a separate class for it, we decided to implement the functionality directly into Gold. Another change that we made was adding a potionBox class. This class is created from the concreteCreator we used to see the type of potion at a location, such as Boost Atk, Boost Def, Wound Atk and Wound Def.

Extra Credit Features

One of the extra credit features that we have decided to implement was the use of smart pointers. One of the areas we used unique_ptr was in our Floor implementation. We used unique_ptr alongside vectors to include the various items that will be used on the floor, such as Cells, Chambers, Enemies, Gold and Potions (floor.h, line 20 to line 24). Utilizing a smart pointer here allowed us to avoid memory leaks easily.

Another feature that we added was the layout of the floor. Our program does not have any restrictions on the number/layout of chambers and there can be any number of chambers on the floor. In the question specifications, there are always 5 chambers, but our program

<u>Final Questions</u>

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?

One of the major lessons we learned as a team was how to properly use git. Some of us had never really utilized gitlab or even git in general in a team environment. Using these tools allowed us to learn how to manage all of our code together in a central location.

Another lesson that we have learned is how valuable communication is. When you are working in a team, and on a project that has a lot of files and tasks to be done, clear communication is needed to understand what is being done and needs to be done.

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

If we had the chance to start over, we would spend more time on our UML diagram, making sure that we have covered all the bases and it is clear. Our final UML has some methods that we did not consider beforehand.