CC3K Report

Introduction: About ChamberCrawler3000

ChamberCrawler3000 is a game that consists of a board 79 columns wide and 30 rows high (6 rows are reserved for displaying information). The player can choose one of six characters who have different abilities according to different roles. A dungeon consists of five floors which each floor consists of 5 chambers connected with passages. The player character moves through a dungeon and slays enemies and collects gold until reaching the stair on the fifth floor to escape.

Overview:

The game has nine different significant classes: Controller, Grid, Unit, Observer, Character, Item, Player (with six subclasses), Enemy(with eight subclasses), Gold.

Class: Controller

We designed the Controller class as the subject class in the **Observer design pattern** and the central controller of the program. It aggregates Grid Class and composites the class Unit. The controller class is responsible for creating everything on the game board, enabling the player/enemy to attack/move, printing out the annotation of command, and notifying the Grid class with new changes. All the commands that the player enters will be accepted through the Controller class and pass to other classes. We plan to use the **model view controller** as the overall structure. For the method auto_attack(), we use **Iterator design pattern** such as std::vector<int >::iterator ite = std::find(arrindex.begin(), arrindex.end(),idd); to deal with the situation that the player character stands in the area which both the same dragon and dragon hoard should attack him, but the dragon only attacks one time.

Class: Grid

We designed the Grid class that inherits from the abstract class Observer. The Grid class contains a pointer to the class Controller to receive updates. The Grid class also has a field of a vector of vectors of char to store and print out the game board. The overridden notify method initializes every cell in the game board with the data stored in Controller Class.

Class: Unit

We designed the Unit class that stores all necessary information for each cell, including the symbol at the position, the previous symbol at that position, the col and row of the unit, the pointer to character/gold on the cell, and whether the enemy has auto moved in the current turn.

Class: Observer

We designed the class Observer as an abstract class. It has a method called notify() that enables the class Grid to receive updates from Controller Class. The Observer class is part of the implementation of the **Observer design pattern.**

Class: Character

We designed the Character Class, which is a base class for both two subclasses Player and Enemy. It contains the main properties for Player and Enemies in protected fields such as (HP, Atk, Def, etc.). It also has many accessor and mutator methods such as get_HP(), change_HP(double value), and set_HP(double value). When we want to modify the values in each character class's fields, we need to use these public methods. For example, change_HP(double value) will add value to the HP field in Character, while set_HP(double value) will set the HP field in Character to value. These mutator methods facilitate our implementation and increase the cohesion of our program.

Class: Player and Enemy

We designed the Player Class (with six subclasses) and Enemy Class (with eight subclasses). Each subclass has a different constructor to set their values, and each of them has a method that is inherited from the Character Class called name(), and it returns a string as the name of the subclass.

Class: Gold

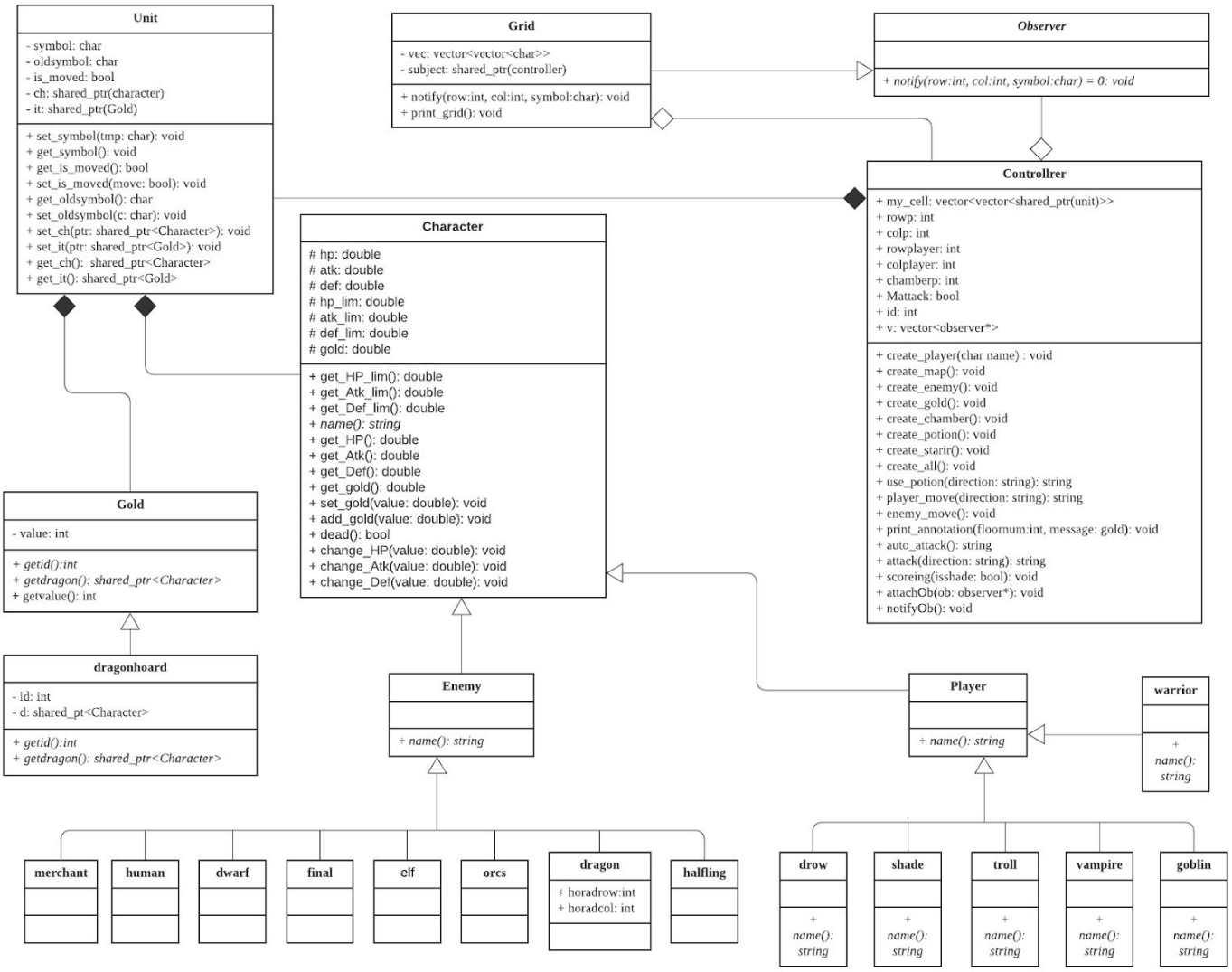
We designed the Gold Class for the following situation.

1. The Player will obtain 1 or 2 gold randomly as the reward of killing a normal enemy.
2. Replace the dead enemy with gold as the reward of killing a merchant or a human.
3. The player character can use gold to trade potions from the merchant.

Class: dragonhoard

We designed dragonhoard class for dealing with the situation that the player character stands in one block radius of the dragonhoard. The dragon will attack the player character. And after the player character kills the dragon, the player can pick up the dragon hard.

Updated UML:



Design:

Design Patterns:

We apply several design patterns to this project to achieve better cohesion and a more comfortable process for debugging.

1. Model View Controller Design Pattern:

   To simplify the process of reading commands from the user and outputting information about the game board, we decide to use the **Model-View-Controller design pattern**. The Controller Class is the Controller Class, and the Grid Class is the View Class, the main class is the Model Class.

2. Iterator Design Pattern:

   Since we use a vector of pointers to observers to store the objects that we need to notify, we decide to use the **Iterator Design Pattern** in Controller Class to notify each object in the vector. We use the default Iterator of vector to traverse the vector and notify each object inside.

3. Observer Design Pattern:

   We choose to use the **Observer Pattern** to link the classes since the observer design pattern can make different classes easily receive and send updates between each other. The Observer Class is the abstract Observer Class, the Grid Class is the concrete Observer Class, and the Controller Class is the subject class. When the Controller Class receives commands from the user, it notifies the Grid Class with the updates and the Grid class prints out the game board based on the changes that notify from the Controller Class.

Resilience to Changes:

1. For deadline one, we planned to use the Visitor Design Pattern for the interaction between a player character and the enemies. But we don't have the confidence to use

this design pattern correctly, so we use another complicated method which works but requires more codes.

2. We used to plan to use the Decorator Design Pattern for creating six kinds of potions. However, after the discussion of the group, we deleted the potion class and created the influence of potion in the Character class and showed their impact by directly changing the value of the player character's HP, Atk, Def. When we go to another floor, we set back the value of Atk and Def to the original one represented by HP_lim, Atk_lim and Def_lim. We believe this method may be easier to implement than using Decorator Design Pattern or using the Strategy Design Pattern.
3. We used to plan to use the Factory Design Pattern for creating the player characters and enemies. But then we realize the simple inheritance method can do the same thing when we use shared pointers for support. That is why we do not use the above three design patterns in the end.

Answers to Questions:

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?

A: We made the Character class as a base class, and all the six races implemented as its subclasses. Due to the inheritance, all the characters share common fields, such as health, damage, defense, and gold. These subclasses can override the constructor and methods from the base class. The Character Class can generate new subclasses. For making an additional race, warrior, we just need to add another subclass to the Player class with providing a little extra code in Character Class and Controller Class.

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

A: Generating enemies will be mostly the same as generating player characters because they share three common fields (HP, ATK, def). But it is different from how we generate the player character because the player character is created only once for each game, but the different probability makes the enemies.

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.

A: The various abilities implemented by the auto_attack() and attack(std::string direction) method in Controller Class. Because most of the abilities occur when the player character attacks the enemy or the enemy attacks the player character. We use similar techniques as for the player character races. For Vampire and Goblin, we use almost the same technique. For the other four, we place their ability into specific methods, such as Drow's ability is included in the methods relating to picking potions.

Question. 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 design patterns.

A: In our opinion, the decorator design pattern would work better. The advantage of the Decorator Design Pattern is that it provides a more flexible way to add responsibilities to objects, just like changing the HP, Atk, and Def of the current player character object. The disadvantages of the decorator design pattern are that it might complicate the process of instantiating the component. The advantage of the Strategy Design Pattern is that it prevents the conditional statements extendable. The disadvantages of strategy patterns increase the number of objects in the application. It needs to detach the strategy, which needs more code

than reset two fields back to zero. But actually, we didn't use any design patterns; instead, we just created a new player character and replaced the gold amount and HP to the old one. That works well and seems more efficient.

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 the production of treasure does not duplicate code?

A: Initially, we planned to write an abstract base class Item containing the collective fields of gold and potion, which are both derived classes of Item. However, in the actual implementation, we find out it is extremely useless to write a class for the potion since we can directly make the effect on the fields of player character. Then we realized that because we do not need the potion class, we only need to write the Gold Class instead of the original three classes. Besides, whenever a dragon hoard is spawned, a dragon must be generated. The dragon hoard cannot be picked up unless the dragon is killed. So the dragon hoard has extra code such as a pointer to show the dragon is alive or dead.

Extra Credit Feature:

Smart Pointers:

We use shared_pointers throughout the whole program without using any delete, and there is no memory leak and segmentation faults.

Extra Player character and Enemy character:

If we use -warrior as one of command line arguments, we can enable the additional player character warrior and enemy character final. Because we use inheritance for creating characters' subclasses, it is easy to add the additional "warrior" class as the inheritance of the Player Class and the additional "final" class as the inheritance of the Enemy Class. Then it will change the minimum of original code in our temporary files, and we only need to

implement the constructor since all the other methods and functions can be inherited from the base class directly.

Extra Merchant's ability:

If we use -trade as one of command line arguments to enable the additional merchant trade feature. We added the setting that the player character could use the gold to buy HP, Atk, and Def increase potion, by specifying command t ne(direction) hp/atk/def (type of potion), and then press y to get potion effect. If the player does not have enough money, then the player will receive a message "You do not have enough money" and the transaction is thus unsuccessful. The trading feature will significantly enhance the experience of actual Players, since we can actually buy some blood when we want to continue the gameplay.

Extra Welcome and Bye Bye page:

To increase the sense of ritual and interest of the game, we added an interface for the player to enter the game and exit the game, and at the same time, it showed that our team members made the game.

Final Questions:

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

Working as a team is an exciting experience for every teammate in our group.

First of all, we learned to share our ideas as much as possible in our group.

- When we faced a problem, different teammates came up with different kinds of algorithms, and then we walked through these algorithms and combined their strengths to make the best method to solve the problem.
- When the compilation error arose, everyone tried their best to find out the location of the errors, so we saved a lot of time in debugging too.

- Sometimes people will be restricted in their logic and ignore some edge cases or potential errors. Other teammates may point out these logic flaws and improve our program.

What's more, it is significant to assign each member of the team what they are good at, which helps to improve the work efficiency and satisfaction of the team.

Furthermore, we also learned to separate the massive program into different small pieces, like different inheritance classes, so we can let different teammates do different things to increase overall efficiency.

Last but not least, we learned to do our work early since the coding of the massive program has an enormous potential of unexpectable bugs, so we need to ensure more debugging before the final deadline.

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

First of all, we will compile every time and download a copy when we make any improvements. There was one time there were so many compilation errors, and we had to modify our main.cc file to test from the base case and to add test cases step by step for locating the bugs. The debugging process is extremely time-consuming, and we will prefer to compile step by step to prevent expensing a lot of useless time at that step.

Secondly, we will implement all header files before writing implementation, so we can minimize the extra work for adding features in base cases due to fatal needs in the process of writing the implementation of the program. For example, if we plan more before writing actual implementation, we can apply the Visitor Design Pattern to maximize cohesion in our code.

Lastly, we will start our implementation early. Although we already implement our program with enough time, it will still be great to have more time to implement more bonus features, like command-line arguments and graphical display.