# CS246 A5 Final Project (CC3K+) DDL2 Report

*Contributors: Sirius Hou, Emily Chen, David Mo*

## Introduction:

ChamberCrawler3000+ is a game where the player character moves through a dungeon and slays enemies and collects treasure until reaching the end of the dungeon. A dungeon consists of different floors which consist of chambers connected with passages. Traditionally, each game contains 5 floors where each floor consists of 5 chambers. The player character (PC, for short) can either be Human, Dwarf, Elf, or Orc. The PC needs to wander among chambers, combat with enemies (non-player characters, or NPCs) to gain coins and unlock treasures, and find the compass to reveal the “stairs” for each floor which can take the player to the next floor. The player can also use potions scattered at random locations on the map or purchase potions from merchants. The exact design of our program is outlined below.

## Overview:

Since this project is all about interactions between each game element (e.g.: combats between the player and NPCs, trading potions with merchants, and collecting treasures and items in maps, etc.), we adopt the strategy of using Object-Oriented Programming and several design patterns learned in this course to develop this program.

## Design:

We split the game into many different modules to simplify the design structure of our implementation, where each module will take care of processing data corresponding to the specific module. For example, the NPC class is responsible for the structure of all NPC objects, the “attack” functions for calculating the damage to the PC and the “attacked” function to respond to the PC's attack. We also have the PC class that is responsible for processing the interaction between the player and other game elements, such as the “applyEffect” function that applies different effects to the player when the player uses a potion or collects an item on map, the “attack” function” which calculates the damage to NPCs and many more. The Grid class acts as a control centre for all the game features, which can coordinate, control and calculate the status of each game element. It also inherits from the subject class to update the text display using the Observer Design Pattern. We applied the Observer Design Pattern on interactions between concrete class objects as well. An example would be when the dragon’s HP falls to 0, we would call the “notifyObserver” function to notify the dragon hoard or the Barrier Suit that it is guarding to unlock them. We also used this for the compass class as when the compass is picked up by the player, the stair will be notified and change its state to “visible to player”.

Our initial plan of attack was to use the Factory Design Pattern to create and delete NPC objects and items. Nonetheless we realized this is too tedious to implement. For Factory Design Pattern, the client does not need to know how objects are created and is prohibited from calling constructors directly, which poses severe limitations to our design. Sometimes a derived class has several constructors, and the client would choose the most appropriate one to use. If we were to employ the Factory Design Pattern, we would need to define several functions to create this object, which is more complicated than calling the corresponding constructors. Thus we did not use this design pattern.

Moreover, we planned to use the Decorator Design Pattern for applying potion effects and picking up treasures, but in the latter implementation we took into consideration that everytime we need to calculate the attack or defence, we need to go through the whole stack from top to bottom and apply each effect one by one, which is quite inefficient. However, we only care about the final results. Therefore, our solution is that we created two private integer fields in class PC, namely, potionAtkEffect and potionDefEffect. Each time we use a potion, if it boosts attack, we add 5 to the potionAtkEffect field; if it wounds attack, we subtract 5 from it. Similar cases for boosting and wounding defence. In this case, the design is much simpler.

## How we implemented each command in our program:

**(1) PC movements: (command no,so,ea,we,ne,nw,se,sw)**

In the main.cc file, it keeps reading in commands until EOF, and it distinguishes whether each command is valid and classifies them into cases. For instance, if it detects characters ‘n’, ‘s’, ‘w’, or ‘e’ from the input stream, then it reads in another character and determines whether this direction is valid. If so, it calculates the coordinates the player will reach after moving 1 unit in that direction and calls the “moveTo” function in class Grid to make the PC move to that direction. The “moveTo” function will first determine whether this block can be “move on to”, (i.e.: it’s not a wall and is unoccupied). And then, it moves the PC to that block and returns whether it has moved onto a stair (i.e.: it’s entering the next floor). If the input direction command is invalid or it’s asking the PC to move on to an invalid location, then an error message will be printed to the output stream.

**(2) Using potions: (command ‘u’)**

In the main.cc file, if a command u is read, it will continue to read in two more characters representing the direction of the potion locates relative to PC. It calls the “usePotion” method in class Grid. Similar to the PC movement, “usePotion” first checks whether there is a potion located in the given direction. If not, an error message will be printed and the program returns back to the stage of waiting commands from the input stream. Otherwise, Grid calls the “state” method in class

Potion and get the potion's effect code (specified in the instructions, e.g.: 0-RH, …, 5-WD). Then it calls the "applyEffect" method in class PC and passes the effect code as the parameter to the function. The "applyEffect" method applies the effect of each potion correspondingly to the PC.

(3) Combating with NPCs: (command 'a')
First, when a character 'a' is read, the main.cc file continues to read in two more characters which represents the direction the player is attacking. Then it calls the "PCAttack" function in the Grid class. This method checks if the attack is valid, that is, if there exists an NPC in the specified direction and throws an error message if it is the NPC is not there. On the other hand, if the attack is valid, it calls the "getDefence" method for the NPC's defence. It then passes this value as a parameter to the "attack" function for PC which returns the PC's damage. Finally, the "attacked" function for the NPC will be called and the NPC will react accordingly.

(4) Restarting the game: (command 'r')
When the character 'r' is read, the program will reset the current level to 1. It then determines on which floor the barrier suit will be generated. Just like when you first start the game, it will also ask for an input of your choice of character and creates a new grid.

(5) Quiting the game: (command 'q')
If the program reads in a 'q' as a command, it will print out a message saying the player has quit the game and break out of the loop asking for inputs, which terminates the entire program.

## Resilience to Change:

Our codes have high resilience to change and a high degree of freedom to add new features to game elements without copying large pieces of code. We can easily add new features or abilities to player characters, NPCs and other elements that affect the player's attributes. We created an abstract base class called NPC, where we defined most of the functions needed to attack the PC, move around, and react when under attack in most cases. And most of those functions are virtual, which means that we can simply override them in the concrete derived classes if they have some distinct features differing from the ones we defined in class NPC. This design system provides us with a great degree of freedom to innovate and achieve polymorphism through inheritance. In this way, we can easily add new elements to the game if we want without using a large amount of duplicated code, which endows the program with high resilience to change.
Furthermore, our program also supports reading in saved game states. What's more interesting, our program can also take maps with different floor layouts (i.e.: the

player can customize the floor structures). This is because we do not use the same floor layout everytime. Instead, we developed an algorithm to identify what each character read in represents and classify them into walls, passages, floors, player, NPCs, potions, etc. When the grid is constructed, we apply this algorithm to "translate" the map we read in into a 2D vector of Cell pointers (where Cell is an abstract base class for all elements in the game) and store it as a field in class Grid. Later when we need to manipulate the grid, we can access and modify each cell through Grid. In this way, we can use any map we create or any saved game state to continue the game on, which shows high resilience to change and innovation.

## Answers to Questions:

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

**Answer:** We create an abstract base class called PC and we implement some common attributes and methods for all races of PC. For instance, class PC has fields HP, Atk, Def, maxHP, etc., and it contains member functions such as "applyEffect", which takes an integer representing the potion effect or which kind of treasure the PC consumes and applies that "effect", and also function "attacked", which is called when PC is attacked by an enemy, etc. These functions and fields set up an abstract framework for PC. When we are creating some concrete derived classes of class PC, such as class Human, class Dwarf, etc, we only need to override one or two functions that are different from the one implemented in class PC, such as the constructor, which sets up the initial player attributes of each race. In this way, it's relatively easy to create new PC races since we only need to specify where it differs from the framework defined in PC. For example, if we wish to create a new PC race called "Warrior", which has higher Def and Atk but lower HP, and its Atk attribute can gradually increase overtime as it kills more enemies. In this case, we only need to create the child class Warrior inheriting from class PC, implement its own constructor setting up its attributes, and override the function "attack" to set up its unique skill. Then we can create this new race without copying a large piece of the same code from other classes.

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

**Answer:** In the Grid constructor, we first read in the map (the default empty map), classify each element into its corresponding type, (e.g., if we read in a '|' at row 2 column 3, then we create a Wall object and store its pointer at theGrid[2][3]) and use

an algorithm we developed to classify cells into different chambers, and store the Cell pointers into each chamber (a vector of Cell pointers for each chamber) (note, all game elements inherit from class Cell, an abstract base class.) Then, we randomly generate the PC, stair, treasures, and potions, and the barrier suit (if it's on the current floor). After that, we begin to randomly generate NPCs of each race. The project instruction specifies that totally 20 NPCs are generated on each floor and each one of them is generated into some specific type based on the probabilities specified in the instruction pdf. So, we create a vector of integers containing numbers from 1 to the number of chambers (by default, it's 5). We write a loop for 20 times that creates a NPC one by one. For each iteration, we shuffle the number list using the function "shuffle" from "algorithm" library and pick the first number from it. That number indicates the chamber number this new NPC will spawn at. Then we shuffle the corresponding chamber (a vector of Cell pointers) and pick the first element in it as the location the NPC should spawn at. If that cell has been occupied (a non-floor element is on it), we'll assign the second element in the chamber to it. Repeat this process until one empty spot has been found for that NPC. After we decide on the exact location the NPC will spawn at, we then randomly decide which race it should be. We use the similar strategy as we used to randomly choose a chamber. We create a vector of integers from 0 to 17, and shuffle it. Then we pick the first number out of it to decide which race it should be. For instance, as the instruction specified, a werewolf's spawn probability is 2/9 (i.e., 4/18), then if the picked number is in range from 0 to 3, then this NPC should be a werewolf, and a Werewolf object will be created at the chosen location. Similar cases for the other races. (e.g., If the number is in range from 4 to 6, then a vampire is generated; if the number is in range from 7 to 11, then a goblin is created; etc.) This strategy is the same as the one we used for randomly generating the PC except that we do not need to decide which race it should be since the player has made their choice before the game starts. Therefore, we can simply create the race the player chooses.

**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.?

**Answer:** We implement those features in the corresponding concrete derived classes. To be more specific, for the gold stealing feature for goblins, we simply add a few lines of code at the end of the overridden "attack" method in class Goblin. The function "attack" is triggered only when it is attacking the PC, therefore at the end of the method, we choose a number at random between 0 and 2 representing the number of gold the goblin is stealing from PC. Since we set the PC's coin as a public static variable, we just minus the amount of coins the goblin steals.
And for the health regeneration ability for trolls, we design it as follows: if a troll is attacked but is not eliminated, then when the PC is away, the troll regenerates its health for 10 HP each round. We add it in the "updateGrid" method in gird.cc. This

function scans the whole grid and checks the status of each element. There, we add a condition that checks whether this element is a troll and PC is not 1 unit around it. If the conditions are met, we call the addHP method in NPC class and add the troll's HP.
For the health stealing ability for vampires, we implement it in the "attack" method in class Vampire. In the overridden method, we add that the vampire's HP is increased by 5 points. Thus, when the vampire is attacking the PC, and if its HP is currently not full, then it "steals" some HP from the PC.

**Q4:** 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:** We could have used the Decorator Design Pattern for this feature since each potion can be treated as a decorator and can be added to the "top" of the PC's cell. However, when we were implementing class PC, we realized that we can simply add 2 integer fields in PC: "potionAtkEffect" and "potionDefEffect". When the PC uses a potion, if it restores health or poisons health, we can simply modify the PC's HP field for the corresponding effect. If it boosts attack, then we add the corresponding amount to the "potionAtkEffect". Similarly, for boost defence, wound attack, and wound defence we modify the two fields above to reflect their effect. Later, when calculating the PC's attack value and wounded value based on its attack and defence attributes, we add its original attack value with "potionAtkEffect" value and add its original defence value with "potionDefEffect" value to calculate its true attack and defence. When the PC reaches the stair and enters the next floor level, we delete the Grid object and create a new one. In the constructor, we reset the "potionAtkEffect" and "potionDefEffect" fields to 0, which means that all the potion effects, except for the health-effect potions, are removed when the PC enters the next floor. Thus, we do not need to explicitly track which potions the PC has consumed on each floor and whether they should have an effect on the next floor.

**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?

In the project CC3K+ instruction, it specifies that each type of potion and treasure has an integer as its code. (e.g.: 0 represents Restore Health potion, … , 5 represents Wound Defence potion, 6 represents a normal gold pile. … , 9 represents a dragon horde.) For each item, the random generation process is the same: we first randomly choose one chamber out of all chambers and choose one particular

unoccupied cell out of all cells in that chamber. Then we create the corresponding item at that location. Therefore, we only need to add a few if-statements that check which kind of items they are and call the corresponding constructors. And for the random location selection part, we can use the same codes for all potions and treasures (i.e.: no duplicated codes).
For the dragon hordes and the Barrier Suit, we use the Observer Design Pattern on them. To be more specific, when a dragon horde or a Barrier Suit is generated, we scan the 1-unit area around it and look for an unoccupied cell where we can place the dragon on. In the dragon's constructor, we pass an additional pointer to the item it's guarding and set it as its observer. When the dragon is eliminated, its observer is notified and sets its state to "collectible". Since the dragon doesn't care what item its guarding, we can use the same piece of code for both dragon hordes and the Barrier Suit.

## Extra Credit Features:

**Feature 1: Use only smart pointers and STL:**
We used only shared pointers in our implementation to manage heap allocated memories to avoid deleting pointers manually. It would also help us prevent the potential memory leaks and segmentation fault problems due to forgetting to delete pointers or using pointers after freed.

**Feature 2: Add and remove customized cheat codes during runtime stage:**
We defined some cheat codes in cheatcodes.txt for demo and debugging which makes it easier to beat the levels during presentation. For instance, we defined the "MOREMONEY" cheat code, which sets the player's coins to 999 so that they do not need to spend a great amount of time on combating with NPC and going through the map to collect coins needed to purchase from merchants. This way, it would be more efficient and much easier for us to test and present the buy potion function of the implementation. Moreover, we defined two additional difficulty levels: one is easy mode (where there will be more random treasures generated in the map, and all NPCs are spawned with half the default HP), the other one is hard mode (where there will be less treasures, and all NPCs are spawned with 1.5 times their default attack). Furthermore, there are two ways to apply these cheat codes: you can enter them as optional command line arguments after you have entered the filename and seed number (e.g. ./cc3k defaultFloor.txt 999 MOREMONEY EASYMODE), or you can apply them during run time by entering the command "+" and "-" to manually activate and deactivate the cheat code (e.g. enter +MOREMONEY during runtime to activate the MOREMONEY cheat code; this will maintain the player's coins at 999 and will not decrease even if the player purchase potions from the merchant. This

code can be deactivated if you enter -MOREMONEY, then the coins will decrease as you purchase more items from the merchant).

**Feature 3: colour scene for text display:**
In our termcodes.h file, we defined several different color codes which are used to set the color of texts we print to the standard output stream. We used this to make different game elements stand out in the text display. For example, we used the color yellow to show the player, red to show all the NPCs, cyan to display potions, green to show treasures and the barrier suit, etc. We also printed the following additional information below the grid: 1. the current state of the player character, which includes the HP, attack, defence, and coin; 2. the area around the PC (the NPCs and potions within two units of the PC); 3. the action of the player for the current round; 4. the combat details with NPC if there were any along when printing the grid.

**Feature 4: “buy potion” option from Merchant:**
We implemented a feature of buying potions from merchants. Initially, every merchant carries randomly 3 kinds of potions. By default, each potion costs 5 coins. To make the game easier, we set that each merchant has an unlimited number of potions and each type can be repurchased unlimitedly. Player can gain coins by collecting treasures scattered all over the map or gaining rewards by eliminating enemies. Additionally, we designed two more difficulty levels: the easy mode (cheat code: BIGSALE) and the difficult mode (cheat code: INFLATION). Player can input command +BIGSALE or +INFLATION to switch to the corresponding game mode during runtime stage. In easy mode, each potion only costs 1 coin while in hard mode, each potion costs 8 coins. Similar to other cheat codes, we can deactivate them and return back to the normal game mode by entering command -BIGSALE or -INFLATION respectively.

## 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?

**Answer:** Before starting coding, we should first come up with an overall design of the project including as many details as possible. This will help us develop our code based on the same design patterns and strategies. For instance, we need to first specify what areas each module should be responsible for and how they are interacting with each other. Then we should also determine what kinds of member functions each class should contain to interact with other classes. Also, we should start with abstract base classes and then implement its derived abstract classes, and

finally, externalize all the concrete derived classes based on the same design pattern and logic. Most importantly, we learned that we need to specify each team member's work before we start writing code. People are always more familiar with code they write and understand them better than those written by other people. In this case, when a bug shows up, we'll quickly identify which part of the code went wrong and fix it instead of spending a lot of time trying to understand other people's code.

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

**Answer:** Before writing any code, we should pay close attention to as many details specified in the project instruction as possible. Otherwise, we might have spent a lot of time writing code that didn't match what we were required to do. Or we might suddenly realize some overall structural defects and might have to start over from the beginning again. For example, when we were implementing the feature of reading in maps from the file, we misunderstood the meaning of the format of input files. We thought that the sample cc3kfloor.txt file is mistaken and instead, we created our format of reading maps: we add some additional lines below each map showing where each game element will be placed at and what effect each of them will have. After 6 hours, we finally implemented this feature. However, after checking Piazza, we realized that we misunderstood the requirement and we had to delete the code we wrote and start over again.

## Cohesion & coupling:

We divided the program into several modules, such as PC, NPC, Items, and Map Elements. For each module, we subdivide it and concrete each derived class element. In this way, every module is only responsible for one particular element in the game. All functions in each module serve the same type of element. Therefore, our program has high cohesion. Also, since our program adopts a top-level management mechanism, that is, the Grid class is the console of each game element and all the manipulations of cell elements are controlled by Grid only, therefore, each derived class does not need to access fields or call methods of other derived classes directly. Hence, our program also has low coupling. This method of design makes the structure and logic clearer and easier for us to debug.

## Conclusion:

As the first group CS project we have at university, we all enjoyed the precious experience it brought us. The CC3K project is not only a coding problem, but also a great opportunity to consolidate our understanding of Object-Oriented Programming and the skills and design patterns learned throughout this semester. Generally, learning how programs work by watching can never have the equivalent effect as

writing codes and reading those error messages by ourselves. After this project, we had a deeper understanding of the Observer Design Pattern and how exactly inheritance among classes should work, meanwhile gaining higher proficiency in using smart pointers. Overall, the impact this project has on us is much higher than what we learned in class has.