CC3k+ Final Design Document

1. Introduction

2. Overview

3. Techniques we used to solve the various design challenges in the project(Details)

4. Resilience to Change

5. Answers to project specifications

6. Extra Features Overview

7. Final Questions

1. Introduction

We have got the basic Text-based version of cc3k working, as well as implementing an extra Window-based version of cc3k using some external library. In this document, we will explain how did we handle different problems during each phase of developing our program, and what we have changed with respect to the original design of our program firstly. Secondly, we will answer what technique we used, how do we make our program resilient to change. Then, we will provid answers for project specifications, and what we learnt from this project. (Our updated UML class model can be found at the last page).

2. Overview

We defined several classes to achieve abstraction, and to make our code more resilient to change:

- Classes to provide background layout and interactions: Floor, Generator, Chamber, Cell.
- Classes to provide player character: *PC*, Shade, Drow, Goblin, Vampire, Troll, *Decorator*, Effects.
- Classes to provide enemies: *Enemy*, Orc, Elf, Dwarf, Human, Dragon, Halfling, Merchant.
- Classes to provide items: *Item*, Potion, Treasure.

We used Floor to incorporate all other classes to provide a way to let all classes interact with each other. Each Floor owns a Generator, and has Cells, a *PC*, several enemies and items. The Generator class provides functions to create *PC*, *Enemy*, and *Item* objects. Therefore we do not

directly use their constructor in the Floor object, making it easier to revise implementation(i.e changes of STAT) and debug. Each Cell obejct contains a char, and it has many useful methods to help us identify its type based on the content it contains. The *PC* and *Enemy* class are quite similar, with simliar fileds and methods except that *PC* has an abstract decorator, *Decorator,* with concrete crete decorator, Effects. Both of *PC* and *Enemy* have several concrete obejcts to provide constructor that set default values for each race of charaters. The Effects class is used to model the effects of potions, eliminating the need to keep track of which potions the player used before entering the next floor. Lastly, the Item class provides method to identify Potion and Tresure, along with their positions.

3. Techniques we used to solve the various design challenges in the project(Details)

This is the order of our implementations:

1. Basic Floor --- to correctly display our floor exactly as the same as an empty input file
2. PC objects --- to see if user can correctly win or can properly move inside the floor we built
3. Item objects --- to see if PC can have desire effects of potions and can correctly remove all the effects after moving to the next floor.
4. Enemy objects --- to see if Enemies can randomly generated and do not conflict with our former classes, and basic features of Enemies(attacking, moving behaviour)

Initially, displaying the floor as the given input file is not as easy as we thought. We realized that simply using a vector of vector of char to store the contents in a floor would lead to many if–else blocks to determine the type of content at a specific location, such as whether it is a wall, a tile, or a passage. To address this, we introduced a new class called Cell to replace a single char. We used a vector of vector of Cell pointer to store the content in a floor. The Cell class provides several convenient methods like: isValid(), is_stair(), and other simple but useful methods to help us determine the type of content at each location of a floor. After that, we started working on the Chamber class, which helps us to determine the valid tiles in a floor. The Chamber class has an int field to indicate which chamber it is(range from 0 to 4 inclusive). It also has a map field with an int key and a pair<int, int> value to keep track of the range of a chamber at each row. With Cell and Chamber classes, our implementation for Floor became much easier. In our design, a Floor class has a private ifstream field, which allows it to store the input file with up to 5 floors. In it's constructor, it takes a floor number as an argument to indicate which floor we want it to represent, and a const string & as another argument to indicate the file name from which we want to read the input text floor.

Secondly, we moved onto to implementing the *PC* class for player characters. All five possible characters inherit from the *PC* class, and they do not override any of the methods in *PC*. Instead, we made changes only to their corresponding constructor to implement their default STATs. The specific features for different races of player character are implemented together in the *PC* class to avoid duplicate code. When calling the attack method, the race of current *PC* is checked, and the corresponding feature is implemented. For example, if a Vampire successfully attacks an enemy, the method would first check the player character's race(int this case, Vampire), and then checks the enemy's race based on the Enemy pointer we passed to the method. Lastly, it determines if 5 HP is gained or losed(e.g., enemy is Dwarf). The *PC* class has methods to get its STATs for a player character so we can use it to check if a player character dies or to decide the effects of potions to a player character. There are also methods to set the STATs, location. In addition to these concrete classes, the *PC* class also has a decorator with a concrete decorator called Effects, which allows us to easily delete all the effects when the player character goes to the next floor(the details of Effects can be found below).

Then we shifted our focus to implementing items. In our original design, we considered both potion and treasure to be concrete decorators for a player. However, during the actual implemtation of this idea, we realized that if they were decorators, they would not be able to exist on their own, and thus, we could not generate them in the floor, which was incorrect. As a result, we decided to make changes to our UML dirgram. We introduced a new class called Effects as the concrete decorator and seperated the Potion and Treasure class from the abstract decorator. This modification allowed them to be standalone instances in a floor. With this changed, handling the effects of potions became much simpler, as we only need to override methods in the *PC* class related to getting the STATs or act_message of the player character. After this modification, we proceeded to write the implementation for Potion and Treasure as standalone classes. We provide them with position, STATs fields, and a string field "type" to indicate if it is "RH" potion, "PH" potion, "Small Pile" treasure, and so on.

Lastly, we implemented the *Enemy* calss, which is similar to *PC* class. All concrete classes of *Enemy* inherit its STATs fields, and there are methods to get their STATs. Additionally, we implemented the different features of different enemy races in a pure virtual method called "attack". The Elf and Orc classes overrides this method to achieve special attack features. There is also a boolean field called "hostile" which defaults to true for all races except for merchant, to indicate if a particular enemy is hostile(attackable) to the player character. There is a method to

set this field to true and if a player character ever attack a merchant, this field is set to true. Consequently, all merchant would then attempt to attack the player character. For Halfling class, we added a method called "success" in the *PC* class to randomly return a boolean value from a vector called "case" which contains {true, false}. Since all enemies have a 50% miss rate, we also added this feature to the entire *Enemy* class. As for dragons, which is associated with a dragon hoard, we added an extra argument in its constructor to allow it to take the position of a Dragon Hoard. Then once the dragon associated with the dragon hoard is killed, the Dragon Hoard's postion is revealed.

As for random generation, we utilized "rand" and "srand" from the <cstdlib> library. In the main function, we initialize srand with time(0) to use the current time as a seed for random number generation. This allows us to have a different seed each time the program is executed. However, if a different seed is provided by the user, we use that provided seed instead.

To achieve the random generation of all the objects, we follow this algorithm:

1. A random chamber is randomly chosen
2. A valid position is randomly chosen from a set of valid positions at that chamber.
3. Note: Since stair cannot be generated in the same chamber as the player, we exclude the chamber in which a player character spawns when generating stairs. For valid positions, we created a helper method in the Floor class to count the number of positions where an object can be generated. This method exclude those tiles that are already occupied by another object, even if it is an invisiable dragon hoard.
4. To generate enemies/items with probabilities, we used a global constant enemy pool(a vector of string) and a global constant item pool(a vector of string). For example, "por_types = {"RH", "BA", "BD", "PH", "WA", "WD"};". We randomly generate an index and use por_types[index] to generate the corresponding item. Similarly for the larger enemy pool. "enemy_types = {"H", "H", "H", "H", "W", "W", "W", "L", "L", "L", "L", "L", "E", "E", "O", "O", "M", "M"}; ". This time, we use enemy_types[index] instead.

Our Floor is responsible for moving the player character and enemy charatcers through the "move_pc" method and "Eact" method respectively. To move a player character, the floor first determines if the given direction contains a valid tile, and then decides whether to perform the move or output an error message if the tile is invalid. For moving enemies, we combined enemies attacks and moves in a single method called Eact. Since we traverse through the floor in a "from leftmost to rightmost and line by line" fashion, we added a field called "acted" in the *Enemy*

class to check if an enemy has already moved or attacked during a particular turn, to avoid duplicate moves.

Our algorithm for Eact is:

1. Iterate through all cell in the floor from leftmost to rightmost and line by line
2. If an enemy is found, check if the player character is in the enemy's attack range.
    a. If the player character is in the attack range, the enemy will perform attack.
    b. If not, the enemy will randomly move instead.
3. Repeat the above steps

4. Resilience to Change

We divided our project into several directories: Enemy, Item, Player, and the main directory. The Enemy/Item/Player were meant to provide specific functions for enemies, items and player character to use, and their correctness were checked within each directory. If they passed our tests, we included them in the Floor class to avoid general dubugging. We also encapsulated our Enemy/Item/PC constructor in a Generator class, which is responsible for generating correponding objects on the floor. It means that no plain constructors are used in our Floor class. Consequently, whenever we make changes to the original constructors(e.g., the arguments they use or the default values they have), we do not need to modify any code in the Floor class. Additionally, we utilized classic c++ features like inheritance, polymorphism to further achieve encapsulation and abstraction. As a result, changes in one module do not affects the others. In most cases, concrete Enemy/PC calsses have inherit methods from the abstract base class, and only a few methods are meanted to be overridden by some concrete subclasses when specific features are needed. Overal, all these approaches together result in a maintainable and modular code structure.

1. If the roles of player and enemy were switched, we could first turn original *PC* class into an *Enemy* class, and the *Enemy* class into a *PC* class, as their features are encapsulated in their respective class. Then we would only need to make a few adjustments in the Generator to change the way they generated, without modifying the Floor class.
2. If more new items were introduced or some items were removed, we could first add/remove item types in the *Item* abstract class intead of modifying every occurrence of items in the Floor class. Then we would simply make small modifications to the Generator class, which is responsible for creating them.
3. If the rules were changed, e.g. the player wins if a certain amount of gold is gained or enemies are killed instead of reaching the stair on the fifth floor, we could add a field and

then modify the specific method "is_win" int the Floor class without modify any other code or any other classes.

4. If a whole new feature was introduced, and it only involves interactions between *Enemy* and *PC*, then only *Enemy* and *PC* classes need to be modified to accomodate this feature. If it involves interactions between *PC* and *Item*, we would add more virtual methods in *PC*. Then only the concrete decorator class would need to be modified.

5. Answers to project specifications

Q: How could your design your system so that each race could be easily generated?

A: We created an abstract base class *PC*, with basic STATs, some common methods to get the STATs and set them. All of them inherit from the *PC* class, and use the regular constructor for PC. We only needed to set default values to each concrete class to stand for different races.

Q: Additionally, how difficult does such a solution make adding additional races?

A: Additionaly, we used the factory design pattern to easily generate these objects, we abstract the constructor for each concrete class into a "Generator" method called "createPC". This method is responsible for creating *PC* objects so when we want to add a race or change the default values of some races, we just need to modify this createPC method instead of modify all the occurrence of the constructor.

Q: How does your system handle generating different enemies?

A: We created different kinds of enemies using an abstract base class, *Enemy*, with pure virtual method to implement special features. We employ a factory pattern to create enemies, just as what we did for generating players.

Q: Is it different from how you generate the player character? Why or why not?

A: Yes. Generating enemies is total randomly, and dragons can only be generated besides a dragon hoard. We wrote an algorithm to find all the Dragon Hoard positions in the floor to correctly generate dragons near it. For other races of enemies, we used rand to randomly pick a position for that enemy and place the enemy at that position.

Q: How could you implement the various abilities for the enemy characters?

A: We implemented Dwarf's feature in the *PC* class, since its feature only works on Vampire, especially when it gets attacked. Similarly, Halfling's feature is also included in *PC* class to stand

for a 50% miss race when Halfling is attacked. For Elf and Orc, we let them override the attack method from the abstract base class so it performs special features when attacking. Dragons, a special case, is associated with dragon hoards so when we create them, we give them an extra position parameter to connect the dragon and the hoard.

Q: 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.

A: We think the Decorator Pattern is better, although there are potential performance issues caused by frequently constructing objects using heap memory. The Decorator Pattern allows us to "link" an effect to the Player, and remove it from the Player when needed, providing flexibility to add more effects dynamically. On the other hand, the Strategy Pattern imposes an effect on the Player, and adding combinations of effects may require writing more algorithms than necessary. In the future, if we want to introduce more effects, we can easily "link" the effects to the Player by using the Decorator Pattern, avoiding the need to write additional algorithms to handle different combinations of potions for the Strategy Pattern.

Q: 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?

A: As mentioned above, we modified Tresture and Potions so that they are no longer concrete decorator classes(they can now exist on their own). Like what we do for *PC* and *Enemy* classes, we created an abstract base class *Item* from which both Potion and Treasure inherit. There are only non-virtual method in *Item* class to get the effect of current item(Potion --- hp/atk/def; Tresure --- gold). A string field "type" stores what item is it so we would not mess up. Through this, Potion and Treasure use the same non-virtual method and hence avoid duplicate code as much as possible.

6. Extra Features Overview

1. Real-time Actions with ncurses:

We implemented real-time actions using **ncurses.h**, allowing the game to respond immediately to player input without requiring the player to press Enter after each action. This was challenging

because we needed to handle user input and update the game state continuously. To achieve this, we used the getch() function in a while loop, which allowes us to process single-character input in real-time and update the game accordingly.

2. Combining Attacks/Use and move key in Two Hits:

We designed the game to enable players to perform attacks or use items with two keys press within a set time duration. This feature required handling different input scenarios based on the timing of the key presses. If the second key press occurred within a short time after the first key press, the action was executed; otherwise, it was treated as two separate key presses. Implementing this behavior was complex as we needed to manage multiple variables and track the timing of each key press accurately. We used time() function calls to measure the time elapsed between key presses and determine whether the action was a attack/use or two separate actions.

3. User Interface and Visual Enhancements:

We designed an intuitive user interface using ncurses to create a visually appealing terminal-based game. We added colorful text, designed clear layout structures, and used ASCII art to enhance the visual experience.

4. Game Score and Records:

In addition to a scoring system to track the player's performance, we also added implemented a record system to keep track of the highest scores achieved in the game. The challenge was to manage records persistently. We solved this by saving the highest score in a text file and updating it whenever a new high score was achieved.

5. Inventory system/merchant selling system

We created a shop where the Merchant's inventory is randomly generated. Ensured the player's gold balance updates accurately after transactions and handled scenarios where the player cannot afford an item. Implemented a bag system to store purchased items until they are used or the player successfully completes the game.

7. Final Questions

Q1: What lessons did this project teach you about developing software in teams?

Writting code together was a totally new experience to us, and we made some efforts to accomodate it. Initially, we send our code to each other through email. This approach was not efficient, and it was hard to work at the same time since we do not know what did each one of us changed to the code. Lately, we learnt Git and created a Gitlab project. This greatly boosted

our efficiency. Now, we have the same version of code through "git pull" command, and we only needed to make sure files are not modified by us both at the same time. In addition to learning how to use Git, we are deeply aware of the importance of writting comments(annotations) to clearly explain what each function does and the logic behind a method. With well-written comments, we can understand the purpose of a function without reviewing the complete code from scratch, saving us a significant amount of time.

Q2: What would you have done differently if you had the chance to start over?

We would pay much more attention to the guideline content next time. We managed to complete our code two days ahead of the deadline with all basic features implemented. While reviewing the guideline again, we realized that we must support a command-line argument to take an optional input file. Fortunately, incorporating this feature would not require significant changes to our design structure. If it had been something that necessitated a complete restructuring of our design, it would have been very dispointing.