CS246 Final Project
CC3K
Documentation and Design

Prepared By

Yuhang Guo
Ruowen Wang
Yanran Zhang

July 31, 2023

1. Introduction

In this project, we produce the video game ChamberCrawler3000 (CC3k), which is a simplified rogue-like game.

In CC3K, the player controls a character as he/she explore a series of interconnected chambers, each filled with different challenges. The dungeons are populated with various types of enemies, each with unique abilities and attack patterns, as well as treasures and potions that can aid the player's journey. Player navigates through the dungeon, battling foes and collecting treasures, all the while keeping an eye on their health and other vital stats. PC can enter the next floor when it walks through a stair represented by '/'. If PC's health value is negative (drops below zero), it dies. If it thrives until the end of the 5th floor, it wins.

CC3K, developed in C++, is a complex project embracing object-oriented principles like polymorphism, inheritance, and encapsulation, along with modern C++ features such as smart pointers, significantly reducing the risk of memory leaks. Implemented with Factory Method Pattern, the random generation of the game elements as well as the utilization of virtual functions contribute to maintainability and flexibility in design.

2. Overview

The program passes in the map as an argument. `GameController` integrates the whole CC3K game, combining two essential parts: `Floor` and `Display`. It handles player interactions by reading in inputs and calling corresponding methods in Floor. `Display` takes the responsibility of printing, including displaying the map and player information.

The Model-View-Controller (MVC) design pattern is used in the implementation.

Model: `Floor` works on the game's logic. It manages the behaviors, player and enemy data, and other essential information related to the game functioning.

View: There is a separation between the user interface(`Display`) and the game logic (`Floor`), while `Display` handles only the task of displaying the game map and panel without containing any business logic.

Controller: `GameController` is responsible for handling user input, processing it, and updating the Model or View. It acts as an intermediary between the inputs and the game information. It processes the user's input, updates the data (e.g. modifies hp, map), and communicates with `Display` to show the updated game version to the player.

3. Updated UML class model

3.1 UML (on next page)

3.2. Current Structure

We start from the `GameController`, and divide the main body of the game into two parts: `Display` and `Floor`. `Display` is used to print the user interface, and `Floor` is a manager of all interactive elements of the game.

`Floor` builds a management system, which divide the game elements into two main parts: *`Character`* and *`Item`*, and use an integrated class *`Object`* to encapsulate them. *`Character`* manages all actors (Player and Enemy) of the game, and *`Item`* manages all entities (Potion and Gold) of the game.

- *`Character`* Part:

`Player` manages all heroes that can be chosen in the game, and *`Enemy`* manages all enemies in the game, we built different kinds of heroes and enemies.

- *`Item`* Part:

`Treasure` manages all golds in the game, and `Potion` manages all potions in the game, we built different potions and golds.

During the running time, we use `PlayerFactory`, `EnemyFactory` and `ItemFactory` to spawn player, enemies, golds and potions, and finally all elements will be updated into the game map, which is built with the format in `Chamber`, and printed by the `Display` class.

3.3. Current Memory Management

Current version mainly use smart pointers (`unique_ptr<T>` and `shared_ptr<T>`) to ensure that memory for dynamically allocated objects is properly deallocated, which not only reduces the risk of memory leaks, but also polymorph our game for lower complexity.

3.4. Current Functions

We add some interest functions based on the original game mechanics to improve the interactivity of our game:

Merchant Store: A new shopping feature has been added. Players are able to use golds they acquire to choose and purchase potions with merchants (M) in the game, each potions has a random stocks and price in merchants' warehouses

3.5. Different From Previous Design:

1. **Remove Observer Pattern:** previous version uses Observer Pattern to update each floors and client interface, but in the latest version we remove it, and use a 2D array to model the map, and implement methods in `Display` for generating the user interface.
2. **Remove Cell Class:** previous version uses *`Cell`* class to build a format of the map, but in the latest version we finished building the format in the process of building the 2D array.
3. **Better Memory Management:** previous version uses simple pointers to manage memory, but in the latest version we use smart pointers for the management

GameController
- display: unique_ptr<Display>
- floor: unique_ptr<Floor>
- status: String
- canRandomMove: Boolean
+ start(): void
+ readCommand(string input): void
+ nextFloor(): void
+ validDirection(string input) const:Boolean
+ getFloor() const: Floor *
+ getStatus() const: String
+ getDisplay() const: Display *
Display
-action: String
+ printMap(Floor *floor) + const: void
+ printPanel(Floor *floor, + char race): void
+ printPCInfo(): void
+ printLose(): void
+ printWin(): void
+ getAction() const: String
+ setAction(string ac): void
Floor
+ isMNeutral: Boolean
+ level: Integer
+ map[25][79]: character [][]
- defaultMap[25][79]: character[][]
+ player: unique_ptr<Player>
- enemies: vector<unique_ptr<Enemy>>
- potions: vector<unique_ptr<Potion>>
-isPotionKnown[5]: Boolean[]
- treasures: vector<unique_ptr<Treasure>>
- chambers: vector<unique_ptr<Chamber>>
- isFrozen: Boolean
+ initMap(string file): void
+ redrawMap(): void
+ initFloor(char race): void
+ clearFloor(): void
+ initNext(): void
+ generateChamber(): void
+spawnPlayer(char race, int &playerChamberID): void
+ spawnStair(int &playerChamberID): void
+ spawnItems(int &dragonNum): void
+ spawnEnemies(int &dragonNum): void
+ movePlayer(string dir): string
+ pickGold(Position p): string
+ seePotion(Potion *p): string
+ usePotion(string dir): string
+ purchase(string dir): void
+ randomMove(): void
+ sortEnemies(): void
+ attackDir(string dir): string
+ autoAttackPlayer(): string
+ getValidPos(int chamberID): Position
+ generateDragonPos(Position *dhPos): Position
+ regainFive(): void
+ frozenEnemy(): void
+ getIsFrozen() const: bool
+ direction(string dir): string
+ purchase(string dir):void
Chamber
+ chamberID: Integer
+ rowStart: Integer
+ rowEnd: Integer
+ colStart: Integer
+ colEnd: Integer
+ checkRange(Position *pos): Boolean
+ getPosition(): Position
Object
- pos: Position *
- type: String
+ setPos(Position *pos): void
+ getPos(): Position *
+ getType() const: String
+ playerWithinRange(Position *playerPos): Boolean
Position
- x: Integer
- y: Integer
+ newPos(String dir): Position
+ getX() const: Integer
+ getY() const: Integer
+ setPos(Integer x, Integer y): void
Shade
Drow
Vampire
+ modifyHp(int newHp): void
Troll
Goblin
Character
- hp: Integer
- atk: Integer
- def: Integer
+ getHp(): Integer
+ getAtk(): Integer
+ getDef(): Integer
+ modifyHp(Integer deltaHp): void
+ modifyAtk(Integer deltaAtk): void
+ modifyDef(Integer deltaHp): void
Player
- maxHP: Integer
- gold: Integer
- defaultAtk: Integer
- defaultDef: Integer
+ getGold(): Integer
+ getMaxHp(): Integer
+ attackEnemy(Enemy *opponent): String
+ pickGold(int amount): void
+ getDefaultAtk(): Integer
+ getDefaultDef(): Integer
+ modifyMaxHp(int newMax): void
+ modifyHp(int newHp): void
PlayerFactory
+ createPlayer(char race, unique_ptr<Position> pos): unique_ptr<Player>
Enemy
- isNeutral: Boolean
+ attackPlayer(Player *opponent): String
+ getIsNeutral() const: Boolean
+ move(): Position
+ notNeutral(): void
EnemyFactory
+ createEnemy(char &type, unique_ptr<Position> pos): unique_ptr<Enemy>
Orc
+ attackPlayer(Player *opponent): String
Human
Dwarf
Halfling
Elf
+ attackPlayer(Player *opponent): String
Merchant
- store: map<unique_ptr<Potion>, int>
+ sell(string p, Player &player): String
+ printStore(Player &p) const: void
+ printData(Player &p) const: void
+ printFunction() const: void
Dragon
- dhoard: DHoard *
+ freeHoard(): void
getDHoard() const: DHoard *
+setNeutral(): void
ItemFactory
+ createPotion(int price, int typeIdx): unique_ptr<Potion>
+ createItem(char &type, ItemType itemType, unique_ptr<Position> pos): unique_ptr<Item>
Item
Treasure
- Value: Integer
+ getValue() const: Integer
Potion
- price: Integer
+ affectPlayer(Player &player, double magnify): void
+ getPrice() const: Integer
Small
Normal
MHoard
DHoard
- isGuard: Boolean
+ freeGuard(): void
+ getIsGuard(): Boolean
RH
- price: Integer
+ affectPlayer(Player &player, double magnify): void
BA
- price: Integer
+ affectPlayer(Player &player, double magnify): void
BD
- price: Integer
+ affectPlayer(Player &player, double magnify): void
PH
- price: Integer
+ affectPlayer(Player &player, double magnify): void
WA
- price: Integer
+ affectPlayer(Player &player, double magnify): void
WD
- price: Integer
+ affectPlayer(Player &player, double magnify): void

4. Design

`GameController` is responsible for handling the inputs for every round and controlling the start and end of the game. It has a `Floor` and `Display`, where Floor deals with the player's input and updates the map and level, while Display prints the map, PC's information, and action. At the beginning of the game, the player will be asked to choose a race by input. The input (char) will be passed to Floor and calling its method `initFloor(race)` to initialize the game. While generating objects, every cell is stored as a char (specific potions and treasures are denoted as *0 - RH, 1 - BA, 2 - BD, 3 - PH, 4 - WA, 5 - WD, 6 - normal gold pile, 7- small hoard, 8 - merchant hoard, 9 - dragon hoard*) in a two-dimensional array of chars called `map` in `Floor` instead of pointers to save memory. Then the player can enter input to play the game. Commands are taken into `GameController` and it is responsible for calling corresponding functions in `Floor` to achieve the effect (e.g. player moves, use potions, attack). `Floor` will also return a message to describe the previous action (e.g. PC moves to East), where `Display` will fetch it. If the PC either win, die, restart, or quit, the status is updated to GameController and the game will end or restart again.

Valid commands to the command interpreter:

[no,so,ea,we,ne,nw,se,sw]: call `Floor::movePlayer(input)`.

[u direction]: call `Floor::usePotion(dir)`.

[a direction]: call `Floor::attackDir(dir)`.

[f]: call `Floor::frozenEnemy()`.

[r]: set the `status` in GameController to "restart".

[q]: set the `status` in GameController to "quit".

All other invalid commands will receive a warning, and be ignored. After an valid input, it comes to enemies' turn, if they're not frozen and PC is within range, then they'll attack PC by calling `Floor::autoAttackPlayer()`. If PC is not around, enemies will move randomly by `Floor::randomMove()`. After processing the input and enemies' action, `Troll`'s ability is handled here. After each valid input, for player whose race is "Troll", it will call `Floor::regainFive()` to regain five hp. `GameController` uses a while loop to continuously take player's input, processes the commands, and performs actions accordingly until a certain condition is met (the player dies, reaches the stairs on floor 5, restarts the game, or quits). If one of the above status expect quit is achieved, the loop would break an back to main function for further instruction (restart or quit). Updated map and panel is displayed during each iteration of the loop.

`Floor` works as the core of the program. All objects are placed in `Floor`. It manages all objects, handles player movement, interactions with objects, and supports combat mechanics. Some Important functions in `Floor` are listed below:

`initFloor(race)`: Takes the responsibility of spawning objects (discussed below) randomly through the factory, spawning player based on the player's input race and set a random position to it.

`movePlayer(string dir)`: Checks if the target cell at the direction is valid to move, if yes, swap'@' with the cell on the map, along with their positions and return a string indicating which

direction the player moved. If no, return a sign "Invalid move" and avoids any change to the map. If the target cell is a unguarded gold, it will be picked when player passes by and added to the player character's total. Note: Only a unguarded gold, floor tile, doorway, or passage will be considered as a valid target cell.

`usePotion(string dir)`: Checks the availability of potion at the target cell first, and return a message based on the action. After knowing it does have a valid potion, iterate over the potion vector to get the potion that matches the target cell's position, call `Potion::affectPlayer()` to act the corresponding function on player.

`attackDir(string dir)`: Handles both player and enemy behavior, including checking availability of enemy at target cell, converting neutral "Merchants" to hostile, calling `Player::attackEnemy()` to let enemy loss hp. If an enemy's hp drops to 0, it performs specific actions, such as stealing gold for Goblin, freeing "Dragon Hoard" or dropping gold piles. It also accounts for special interactions with the "Vampire" race, affecting gaining or losing hp.

`autoAttackPlayer()`: Firstly, calls `Floor::sortEnemies()` to sort the vector named enemies that holds all the pointers to enemies based on their positions, from left to right, up to down. For each enemy, determine if the player is in its attack range. If yes, call the `Enemy::attackPlayer(Player *opponent)`.

`Object` is an abstract base class of two abstract classes: *`Character`* and *`Item`*. *`Character`* and *`Item`* are distinguished by whether they are movable and whether they have hp, atk, def or not. *`Enemy`* and *`Player`* both inherit from *`Character`* and each of which has different races with each race has its own characteristics. Each *`Object`* has a position so that we can get the particular object by comparing its position.

`Enemy` is an abstract class that serves as a base class for its concrete subclasses(`Human, Dwarf, Elf, Orc, Merchant, Dragon, Halfling`). It contains a virtual member function `attackPlayer(Player *opponent)`, which allows concrete subclasses to override this method to implement their unique attack abilities when attacking the player. `Enemy` provides a common interface and functionality that can be shared among different types of enemies in the game.

5. Resilience to Change

First of all, our program applies robust modulization. It enhances resilience to change by dividing a software system into separate, self-contained modules. This isolation allows for changes in one part of the system without affecting much of others, enabling more efficient and targeted development, testing, and maintenance.

We also have implement inheritance so that some shared parts will remain and differentiation can be achieved in its subclasses. For example, we implement various classes that represent the different

elements required in the game, which includes four abstract classes *Player*, *Enemy*, *Potion*, and *Treasure*. We also have concrete elements that inherit from these four abstract classes, which maintains certain parts from super class while adding some new features that differentiate them or overriding the virtual functions in their superclass. By the abstract interfaces and the virtual methods in the superclass, we are assuming that these are some shared characteristics or abilities that all the subclasses can override or just remain unchanged (e.g. we have an virtual `attackEnemy()` method in *Enemy* class, where all concrete enemies can directly use or override it). Thus, if we want to introduce a new type of either player/enemy/potion/treasure, we only have to follow few steps. Firstly, we have to construct a subclass that inherits one of the four superclasses. Then we have to decide whether or not to override the virtual functions in the superclass. If extra features need to be supplied, just add new fields or new functions.

Since most of the concrete classes are independent of each other (eg. A potion seldom related to a treasure in this game), we have successfully achieved low coupling by nature. Thus introduction of new element won't won affect much in the original code, as long as you include the header files in the required module. If the new element may somehow interacted with our existing concrete classes (e.g. A newly added player may be uneffected by certain potion, or a newly added enemy may be hit by player and freezes for three turns), we will consider to modify some specific functions in our Floor class. Since we give explicit methods names in Floor, we will clearly see which function to change. For instance, the above two scenarios should modify the `usePotion()` and `autoAttackPlayer()` methods, while you might need to add new fields in its own class.

If we want to add new features that are common to all elements in a particular abstract class, for example, if we want to add an 'energy' field in players, you can simply add this field in the abstract `Player` class, which is efficient, maintainable, and flexible. You can easily tailor the behavior of this field in each subclass as needed, while still benefiting from a shared implementation.

In addition, since we use the Factory Method Pattern to generate all these elements, we have encapsulate the functionality of generating concrete elements within the factory method. These factory methods are highly extendable. Thus, we don't need to change the basecode or the logic in the spawn methods from the `Floor` class where we call the factory methods, we only have to make edits on those factory methods by adding one or two lines creating that element in the heap by calling its constructor. Also, the factory methods can extend easily. If a common attribute or behavior must be applied to all objects of a type, incorporating this logic within the factory method ensures that it's applied consistently. If that common attribute or behavior needs to change, it can be done in one location.

Besides, since we use Strategy Pattern in different concrete Potion classes to achieve the potion's effects on PC, it is easy to modify their effects without affecting others's. We can also add new effects simply by creating new strategy classes, without altering existing code.

In conclusion, with the combination or modulization, inheritance, factory method pattern, and

strategy pattern, our code achieves high cohesion and low coupling, therefore enhancing resilience to changes.

6. Answer to Questions

6.1. How could your design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional races?

A: We create an abstract base class *Character* with two abstract subclasses *Player* and *Enemy*. The races (Shade, Drow, Dragon, Human etc.) will all be created as subclasses and inherit from either *Player* or *Enemy*. *Character* has all the fields and methods common to *Player* and *Enemy*, including HP, Attack, and Defense. All the player subclasses would override and add methods to enable their specific behaviors of that race. Adding additional races only requires creating a new subclass and then inherit from *Player* or *Enemy*.
(*new) We also implement a Player Factory and Enemy Factory so that each race can be easily generated. Player factory takes a specific race type and a valid position as inputs, while enemy factory takes a reference to a `type` (which will be modified later) and a position. They both create a Player or Enemy object on the heap and returns a unique pointer, thereby transferring ownership of that memory to the main program. By utilizing this factory design pattern, we can easily introduce additional races with minimal changes to the existing code, simply by adding a few new lines within the factory itself while maintaining the original logic.

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

A: Both ‘player’ and ‘enemies’ are generated within the chamber utilizing factory method. However, there are some major differences between how enemies and the player character are generated. First, PC is chosen at the beginning of the game by the player, so we will only call the factory method once based on the choice of that player. Whereas enemies are generated randomly according to the given amount and probably using `rand()`. Thus, which enemy to generate depends on that random number generated. For the spawn location, we first need to randomly choose a location for PC, then we generate stairway, potion, and gold, at last we will choose 20 random locations for the enemies within the chamber.
(*new) Since every dragon is generated near a dragon hoard, when each dragon hoard is spawned, a dragon guarding that hoard is subsequently generated without calling the enemy factory method. After spawning all treasures and dragons, we should calculate the remaining number of other enemies to be

spawned. Then, we will call the enemy factory method to generate all those enemies.

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

A: ~~The method of implementing various abilities for enemy characters is similar to implementing player characters. That is, by overriding the methods in their base classes, *Player* and *Enemy,* to achieve additional logic. Take elf and dwarf as the example. First, their base class named *Enemy* declares a method dealDamage() that is common to all enemy characters. Then its subclasses elf and dwarf would override method dealDamage() as needed based on their own Defense value.~~

(*new) No. The ability of different races for *Player* are implemented in functions in Floor or GameController. Since Player is fixed for every round and Floor stores that Player, we can easily handle Player's special feature by entering the if statements within certain functions. For instance, if the Player's race is a vampire, 5hp are recovered for each valid attack at the end of the `attackDir` function. To achieve Drow's ability, we will first determine if the player's race is a drow at the beginning of the usePotion function in Floor, and if it is, we magnify the potion effects by 1.5, and 1.0 otherwise.

However, for enemies, race-specific abilities are achieved by overriding the methods in their base classes, *Enemy. Enemy* declares a virtual method `attackPlayer` that is common to all enemy characters and handles normal enemy attacks on the player. For those enemy races with special features like orcs and elfs, we have overridden the `attackPlayer` method in the subclass Orc and Elf to achieve their features.

6.4. 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.

~~A: We choose to use the Decorator pattern, where the Player class is the Component, the Potion class is the Decorator. Potion 'has a' a Player and Potion inherits Player.~~
~~The advantage is that the Decorator pattern allows us to add effects to the player dynamically at runtime, which can wrap the original player with the new effects every time the player consumes the potion. The disadvantage is that we have to create many concrete potion classes that may increase the complexity. On the contrary, using the Strategy pattern is easier to understand, and the relationship between PC class and Potion class is less complicated compared to Decorator Strategy. The drawback~~

~~is that we will have to write more code, and we have to add a cosumePotion method in the Player class to apply the effects of potion on the PC. However, if the Potion is the decorator class, it has a Player object, and will directly override the mutator methods, which is more straightforward.~~

(*new) We choose to use the Strategy pattern. It has the following advantages. First, it provides great flexibility and maintainability. By encapsulating the behavior of applying potion effects on PC, you can easily add new types of potions without altering existing code. Each new potion type simply requires a new class that implements the `affectPlayer` method. Also, if you need to change how a specific potion works, you only need to modify the corresponding class. In addition, it is simpler than Decorator pattern under this context. We do not need to implement the inheritance and aggregation relationship between Player and Potion class, which is not that reasonable and more error-prone.

On the other hand, the Decorator pattern offers extensive flexibility by allowing you to layer various effects on the PC without changing the foundational player class. This pattern works best when you want the core class to remain as is, while continually modifying the decorated versions. In contrast, the Strategy pattern provides a more direct method to modify specific attributes of the Player (e.g., Health Points, attack), which fits well when using a potion.
While the Decorator pattern can introduce new functionalities without affecting the underlying structure, its application may not be suitable here, as the goal is to directly alter particular aspects of the Player. Therefore, the Strategy pattern might be a more efficient approach in this context, offering a clearer way to enact these changes.

6.5. 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?

A: We let Potion and Treasure as subclasses of *Item* and inherit from it. Then, we choose to implement a single Item Factory that handle both the generation of Potion and Treasure by passing different type signature `Itemtype`, which indicates whether to generate a Potion or Treasure. Then, based on the item type, we have a conditional structure that generates potion and treasure respectively according to the given probability. It returns a concrete potion or treasure which is pointed by a unique item pointer. When we called the `spawnItem` method in `Floor`, we will first spawn 10 potions. So we will call the item factory method 10 times, once we get the unique item pointer, we will first release it and then static cast it into a Potion pointer, then accept it as a unique Potion pointer. Similarly when we spawn 10 treasures. Therefore, we can create different types of items (potions and treasures) without duplicating code.

7. Extra Credit Features

7.1. Utilization of smart pointers

By using unique pointers in our program, the lifetime of all the `Object` is automatically managed. We do not include any `delete` in the code, and there is no memory leak.

7.2. Purchase Potions from Merchant

We implement the function of purchasing potions from Merchant. Player will input [b direction] to query the merchant, than a potion store will be displays with all six potions with randomly generated price and amount. PC can buy all six potions if they are not sold out and enter [exit] to leave the store. A message will displayed if PC do not have enough gold to buy or the entered good is not valid or sold out.

8. Final Questions

8.1. 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?

Developing software in a team has taught me the significance of planning ahead and earnestly following the plan. A reasonable division of tasks among team members helps bring into play individual strengths, making cooperation more efficient and allowing members to learn from each other. Effective communication is an essential key in teamwork, it not only avoids duplication of work, but also minimizes misunderstandings. Besides, follow a consistent style and write clear documentations. It gives other members the chance to understand and use the method you implemented without reading the implementation from line to line. It also facilitates future modifications and debugging. It also taught me to decide on the framework of the program before writing implementation, for example, drawing an UML class mode will help a lot on making logic clearer. It's also a good idea to break large and complex tasks into smaller parts and solve them one by one. Remember to test and debug often so that it will not end up with too many bugs and further lead to more elusive problems.

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

First, spend more time carefully considering the overall structure and interconnections of the program, for example, what design patterns need to be used, rather than realizing its inadequacy after implementation. Another point is that I will think carefully about the logic, special cases and executability of each method, rather than relying on improving it during debug time.