CS 246 Final Project Design

Xinyan Lin, Hanyu Xu, Rivers Chen

August, 2020

1 OVERVIEW

The game of ChamberCrawler3000 is a roguelike game in which the player's goal is to reach the exit at the top floor of the five-floor map.

The game follows the OOP design principle, where all actions from clients should be called in the GameController class.

2 DESIGN

2.1 System Components

2.1.1 Game Element

All objects are inherited under an abstract class `GameElement`. which has game element's x and y positions and their char to display as attributes. `GameElement` has three direct subclasses `Living` and `NonLiving`, and `Architect`.

2.1.2 Living Characters

The `Living` class is an abstract superclass for `PlayerCharacter` and `EnemyCharacter`. `Living` contains character's race, HP, Def, Atk as attributes, and provides their corresponding accessors and mutators.

Under `PlayerCharacter` and `EnemyCharacter`, each distinct race is implemented as a subclass of them. For player characters, the different basic stats (hp, atk, def) of different races are initialized with default value in the constructor of each race respectively. For the special ability of different races, the template method design pattern is applied. For example, shade has its score magnified by 1.5. This is achieved by overriding the score accessor of `PlayerCharacter` in the `Shade` class and return 1.5 * score instead of actual score. For another example, goblin gains 5 gold for every slained enemy. This is achieved by overriding the `slain` function of `PlayerCharacter` in the `Goblin` class. Beside the normal function flow, 5 extra gold is added to the score of the goblin. Using the template method allows each race to override and modify certain steps to implement their special abilities, but also reuses code for common features. The implementation of different enemy character races is similar to player characters, which also applies template method.

2.1.3 NonLiving Items

Both `Potion` and `Treasure` inherit the `NonLiving` superclass which inherits from `GameElement`.

The `Potion` classes use the decorator design pattern. The potions player character drinks exist as decorators to the player character's status, and PC holds a pointer to a concrete component class of "no effect" potion initially. This way potions can be easily added and removed when needed, and their effects can be calculated independent of player character. Since potions affecting attack and defense are only effective in the current floor, they are removed when the PC reaches the next floor.

The `Treasure` class uses a similar design as the potion class. For each of the gold size, a specific enum class type exists in the header file which integer value corresponds to its size. When constructing a treasure, the client does not need to know necessarily the actual size of the treasure type. Unlike `Potion` class, the player character is responsible for adding the value of the treasure to its score field. A special case is for Dragon Hoard, since it can only be picked up after the dragon guarding it is slain, a dragon hoard will return 0 from its getAmount method, indicating that it cannot be picked up yet.

2.1.4 Architecture

The `Architect` class, for `Architecture`, is a direct subclass of `GameElement`. It is used for floor tiles, walls, passages, doorways, and also empty spaces. Except the three attributes inherited from `GameElement`, `Architect` also contains chamberInd as an attribute. The chamberInd for each floor tiles (' .') is initialized to the index of chamber it is in, and other architectures it's just the default value 0.

2.1.5 Board

The `Board` class stores all gameElements as shared pointers. It contains a 3d vector floor, which each floor[i][j] represents a square on the board, and is implemented as an 1-dimensional vector of game element shared pointers. The first position is always an `Architect` object, and any other game elements "standing" on this position is pushed back onto this 1-dimensional vector.

`Board` also contains vector of shared pointer of `EnemyCharacter` named enemyList which is renewed at each level. This vector has all enemies on current floor that moves (so dragons will not be in this vector). When we move enemies at the end of each round, each enemy in the vector that are not beside pc will be removed from floor in order then randonly assign a new position. After that, this enemyList will be sorted according to new position.

2.1.6 GameController

The `GameController` acts as a controller of the whole game, it is a friend class of `Board`. The functions in `GameController` will access the game elements on floor to handle tasks such as setup of floor, spawn of PC and enemies, move PC and enemies, etc. The specific design of how these are complished will be in section 2.2.3.

`GameController` holds a shared pointer to our player character and also the x and y position of the stair of current floor.

2.1.7 Display

By design, everything relating to output of the game components should be handled by `display`. `display` defines a set of flags corresponding to various possible states of the game. Only one display object exists in the program as a global variable, so that any other objects can pass one of the flags to `display` when needed. In addition, to increase flexibility, `display` can accept a string directly and append/prepend it to the game message.

2.2 Features

2.2.1 Setup

If we are at the initial level, the game will ask player to give a valid player character race char.

Each level, `floor` in the board will be reset, so all game elements of last floor will be discarded except pc. The enmeyList will also be reset.

If no command line argument is provided, then, a default layout of floor which only contains architectures will be used to build to fstream. `gameController` will use this fstream to setup an initial empty board, then use the char player provide to spawn PC. Afterwards, `gameController` will spawn other game elements, when spawning enemy, each enemy will be added to the enemyList. And, each dragon hoard will have its dragon as a field.

If a single command line argument which is the name of a file that specifies the layout of each of the 5 floors is provided, then, readFloor of `gameController` will be called to change each char in the file to an object on our board.

2.2.2 Beginning of one round

At the beginning of each round, the player input a char to indicate the action they want to take.

If that input is used to do game settings. For example, if to reset the game, the `gameController` will reset floor level to and break the inner loop. If the input is to quit the game, then displayLose() of `gameController` will be called and game is quit.

Or the player may play the game by move pc(which may include pick of gold), attack enemy or drink potion, which is discussed below.

2.2.3 Middle of one round

Move: Check the direction is unoccupied, then revert where it was on board floor to a floor tile, and push back pc on its new position on floor. If that direction has a gold of type not dragon hoard, then pick gold and move to that position. If that direction is a dragon hoard, then check if dragon is dead, pick gold and move to that position.

Combat: When implementing the combat system between player character and enemy character, observer, visitor and template method are all in use. When the player character gets within 1 block radius to an enemy, observer pattern is used to notify the enemy so that the enemy will attack the player character. Since the damage dealt by an enemy varies based both the race of the enemy and the player character, visitor pattern and double dispatch are used to implement an attack. Whenever a character A is trying to attack a character B, it calls the function A.attack(B), then A.attack(B) calls B.attackedBy(A), which is an equivalent visit function that behaves differently based on the concrete class type of A (races of A in this case). Such design pattern makes the combat system clear and straight forward. For different races, damage will be calculated differently. Furthermore, some races have special interaction between each other. And template method is applied to handle these interactions. If pc dies during combat, lose message will be displayed and player will be asked if they want to start again. If enemy is dead, it will be removed from enemyList and floor.

Use Potion: again, `gameController` will check there is a potion at the direction inputted. Then, it will call pc's drink(std::shared_ptr<Potion>) to attach potions to pc as decorators. How potions work has been discussed in section 2.1.3. The potion will then be removed from its position.

2.2.4 End of one Round

After these actions are completed, MoveEnemy(int, int) will be called, then `gameController` will use enmeyList in `Board` to move enemy as described in section 2.1.5. Also, `gameController` will call flushDisplay() to display the board and stats of pc and action taken.

3 RESILIENCE TO CHANGE

The `Living` class involves player characters and enemy characters. It is designed to follow high cohesion and low coupling. In the `Living` class, each distinct race has its own header and implementation files. Thus, a header and an implementation serves on a single purpose of implementing the specific race. It demonstrates the high cohesion in our code. The communication between player character and enemy character happens in combat only. In a combat, function calls only takes others in as basic parameters without knowing the actual implementation. It demonstrates the low coupling in our code as different modules has very low dependencies between each other.

Since the potion subclasses use the decorator design pattern, which allows us to easily add new types of potions to the game, as well as designing more complicated types of potions, such as ones that have dynamic effects depending on the progress of the game. Its functionality is also independent from the player character, which is easier to make changes to the potion classes only.

The treasure class uses a enum type to define the various size of gold, if other sizes of gold are need, or if the current size of gold has to be changed, one simply has to define additional members or modify existing ones in the enum type and pass the type into the constructor of treasure. The existing code of generating treasures may not need change at all. If other types of treasure are need, subclasses of Treasure can be added to accommodate the change. How treasures are picked up by Player character is defined on the PC's side, who is a friend of treasure class, thus no other changes are needed.

The display class handles all the standard message the game has by having a set of flags corresponding to various possible states of the game. It also dynamically read in the game board when printing its content, and thus there is no need to change the code of display if other player characters and/or enemies are added to the board.

4 ANSWERS TO QUESTIONS

Question: 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?

Answer: The answer for this question is basically the same as answered in due date 1 design.

As discussed in section 2.1.2, to generate each race easily, we apply the template method. We would create a class named `PlayerCharacter` as the abstract base class. A player character must be one of the race type, thus an object of the parent PC (without race) will never be initialized, which means PC will have pure virtual method. Then, based on the template method, we will make each race a distinct sub class of PC which inherit accessors and mutators form `Living` and `attack` and `getAttacked` from `PlayerCharacter`.

Such solution makes adding additional races easily. If an additional race is to be added, then we would make the race a new sub class inherited from the base class PC. We would create a header file and a separate implementation file for the race. We can override any method required in the sub class. Doing so results in high cohesion and low coupling. The new header and implementation file serves solely for the purpose of creating a new race. Also, such implementation will not affect the base class or any other races.

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

Answer: For how our system handle generating different enemies, the answer is same as answered in due date 1 design.

Like for player characters, use template method to generate different enemies, `EnemyCharacter` is the abstract base class which will never tbe initialized, `EnemyCharacter` will also contain methods called "attack" and "attackedBy". "Attack" is a method called upon the enemy attacking a PC whereas "attackedBy" is a method called upon the enemy being attacked by a PC.

Based on the template method, make each race a distinct subclass of `EnemyCharacter` like we did for player characters, then these subclasses all inherit accessors and mutators form `Living` and `attack` and `getAttacked` from `enemyCharacter`.

For the difference between generating player character and enemy character, our answer is different from our due date 1 design.

We have `GameController` responsible for game flow and management of pc-related behaviours. When user input their desired race, GameController has function `spawnPC()` which generates the required race PC.

We have `Board` responsible for game component generation and management of pc-unrelated behaviours. At the beginning of initialization of every floor, enemies will be spawned. For the enemy generation, the Board will call its method "spawnEnemy". SpawnEnemy has a helper named spawnOneEnemy that can spawn one specified enemy. SpawnEnemy determines an enemy race based o n the probability distribution, and calls spawnOneEnemy to generate such enemy. It repeats the process 20 times to get 20 random enemies.

On due date 1, we proposed to use Board class to controll the entire game flow. But in actual implementation, we decided to have Board and GameController taking different responsibilities of the game flow. Doing so increases the cohesion within the classes. We put PC generation in GameController, as its generation is related to user inputs. We put enemy generation in Board, as it belongs to part of the floor initialization and is unrelated to user's behaviours. Such reason causes the generation of player character and enemy character to be different.

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.

Answer: The answer for this question is slightly different from the answer in due date 1 design.

The various abilities for the enemy characters will be implemented differently. We sort the abilities of enemy characters into three categories: attack priority (for halfling), extra damage (for dwarf, elf, orcs), and others (for human, dragon).

First, we would expose our implementation of combat. In a combat, a character A attacking a character B has the following procedure: calling A.attack(B), A.attack(B) will call B.attackedBy(A), "attackedBy" calculates the actual damage and change the HP of B consequently. Here we exploits the observer pattern and visitor pattern. The observer pattern is used in such way: when a PC gets within 1 block radius of an enemy, it notifies the enemy so that the enemy get notified. As long as an enemy gets notified, it tries to attack PC. The visitor pattern is used so that the damage will be calculated differently based on the races of player character and enemy character.

To implement attack priority for halfling, when PC tries to attack a halfling, a 50Also, halfling has 100function of EnemyCharacter is overridden by halfling. Normally, when an enemy gets notified by an PC, it has 50the probability check is removed so that an halfling always attacks the PC when getting notified.

To implement extra damage, the template method will be applied here. In default, the base PlayerCharacter class has 7 "attackedBy" methods that each takes in a different enemy race with respect to the visitor pattern. Howevery, the enemy races belong to the extra damage categories do extra damage to specfic PC races. Specifically, vampires react to dwarves, all races except drow react to elfs, and goblins react to orcs. Thus, we will override the distinct "attackedBy" for the mentioned enemy race in the character race class. For example, when attackedBy takes in an orc, goblin class will override attackedBy(orc) and special damage calculation will be done. When attackedBy takes in other races, it follows the default attackedBy defined in the PlayerCharacter class.

To implement the ability of human, upon the death of a human, extra gold pile generation is required. As mentioned before, we have a Board class responsible for management of pc-unrelated behaviours. Specifically, it is responsible for removing enemies from the board when the enemies are slained. Thus in Board's function enemyDeath(), it checks the race of the dead enemy. If it's determined to be a human, two piles of gold will be generated.

To implement the ability of dragon, its associared dragon hoard maintains a pointer to its dragon's address. It means the dragon generation is isolated from normal enemy generation. Instead, whenever a dragon hoard is generated, a dragon will be spawned consequently.

The implementation of enemy abilities is different from of PC races. PC races has no special interaction with enemies. Thus, they can be implemented through changing the information of PC (I.e. fields and methods). But most enemy abilities have special interaction with PCs (they do damage differently), thus we use different design patterns and implementations to complete such features.

Question: The Decorator and Strategy patterns are possible candidates to model the effects of potions, in your opinion, which pattern would work better?

Answer: To adhere the decorator design pattern, potions would be concrete classes of Decorator. While any player character or mobile who can use potions owns a concrete class of Component corresponding to the "base" potion effect, which defaults to doing nothing. At the creation of a character it is initialized with a concrete class of Component which returns 0. When using a potion with non-permanent effect, the Decorator is attached to the corresponding component. When using a potion with permanent effect, the value is directly added to the player's status. When calculating a player's final status values, the chain of Decorators is called, which return value is added to the player's own status value. In this way, one can detach any of the non-permanent potions at ease. Any special ability or non-standard event during combat is not handled by the Decorators and is directly calculated in the character class.

To adhere the strategy pattern, one has to create a concrete Strategy for each player status. The character owns one pointer to a virtual Strategy object for each of their stats. When encountering a potion, the effect of the potion is passed to the corresponding object. All the strategy object have to do is to record the effect, and hold a collection of all the potions that it has encountered during the game. When calculating the character's final stats, the original value is added to the value returned by the Strategy object, which gives the combined effect of potions. In addition it can provide a reset function which erases the effect of non-permanent potions.

Both design pattern have their advantage and disadvantages. The decorator pattern allows easier management of each potions, which provides extensibility when we are required to access each individual potion. The strategy design is easier to implement in the default case and uses less resource. Although it can only handle a small amount of combinations of potions, which is not a

problem in this case.

In my opinion, the decorator pattern would work better, as it allows us to easily add functionalities.

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 generation of treasure does not duplicate code?

To reuse as much code as possible, Potions and Treasures should inherit the same base class. Note that treasure and potion share many similarities. They can only be picked up when players are near them, and they are immobile items generated at the start of each floor. To do so we need them to both inherit a virtual base class of Items (in our design UML, it is called NonLiving). The constructor controls where the object is placed, as well as the specific type of the potion/treasure, which is determined by an enum class type predefined. Then we encapsulate the generation of these items in a method and let it handle the details of generating each type of items. If we want to extent this functionality, we have to provide another method with different parameters that controls things such as number and proportions.

5 EXTRA CREDIT FEATURES

In our project, we uses smart pointers to manage memory without leak. No delete statement exists in our program.

6 FINAL QUESTIONS

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

Answer: Communication is very important to facilitate a smooth working flow. A good design does not only increases coupling, it also makes the classes easier to implement. If the interface of each classes are determined beforehand, each team member only has to deal with the classes they are responsible for, not worrying about how their classes are used by other classes. Also, the classes that has more clients should be implemented first, since it is likely that the design will be changed during development; except for the main loop of the game, which should be written first to give a better idea of how the game should be run in general.

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

Answer: If I could start over, I would change the design of board to be a nested class of game controller, this allows multiple boards to be created during the game, so player may go back to the previous floor. I would also eliminate the use of friend classes so that all operations are made from public methods to reduce cohesion. The display classes should be made such that it has an inclusive list of all possible game events, so that the newAction method is not needed, and all call made to display should only contain flags. This increases the complexity of the class, but it also allows several game messages to be combined into one sentence, making it more user friendly.

7 Conclusion