Final Design Report
ChamberCrawler3000+

1195 CS 246
Assignment 5

Siyu Liu (s565liu), Sining Jia (s9jia)

July 30, 2019

Table of Contents

Introduction

In this project, we produced a game called ChamberCrawler3000+ (cc3k+), which is a game where player needs to move through a dungeon, slay enemies and collect treasure until reaching the end of the dungeon. Score is calculated based on the total amount of gold collected during the game.

In this document, we include the overall structure of our project, an updated UML, some detailed explanations of the design we used, answers to the questions, extra credit features, what we learned from the project and what we can improve.

Overview

- Our program starts with the **main function (in main.cc)**, after initializatializing a new **Game object g**, it reads user inputs. Depends on the user inputs (commands), the main program decides which game function (in game.cc) will be called. Commands include move, attack, use potion, restart, and quit.

- The **Game class (in game.h)** is the main control of the game. We call init()/init(string fileName) method to have the game object created in main.cc get initialized. It calls the **Player class** constructor to initialize the myPlayer field given the race provided by the user, and then calls private method initFloor()/initFloor(string fileName) which subsequently calls the **Floor class** constructor (which takes myPlayer as one of the parameters) to initialize the currFloor field. After the currFloor field is initialized, it calls several methods on currFloor to initialize myPlayer and stair position, and then generate potion, gold, Barrier Suit, enemies, and compass in order.

- The Game object g calls its methods move(Direction), attack(Direction), and usePotion(Direction) to move myPlayer, attack enemies, use any potions or pick up other treasures. Inside these methods, the corresponding Floor methods are called with some conditions, which modifies the state of myPlayer the currBoard fields.

- The main function displays an output to users after executing each command, it calls the Game's printGame(string) method to return a string representing the game board to the user, which is done through calling the printFloor() (which returns a string representing the current board display) on currFloor, and getHP(), getAtk(), getDef(), getGold() on myPlayer along with some conditions and the string parameter which describes the action occurred.

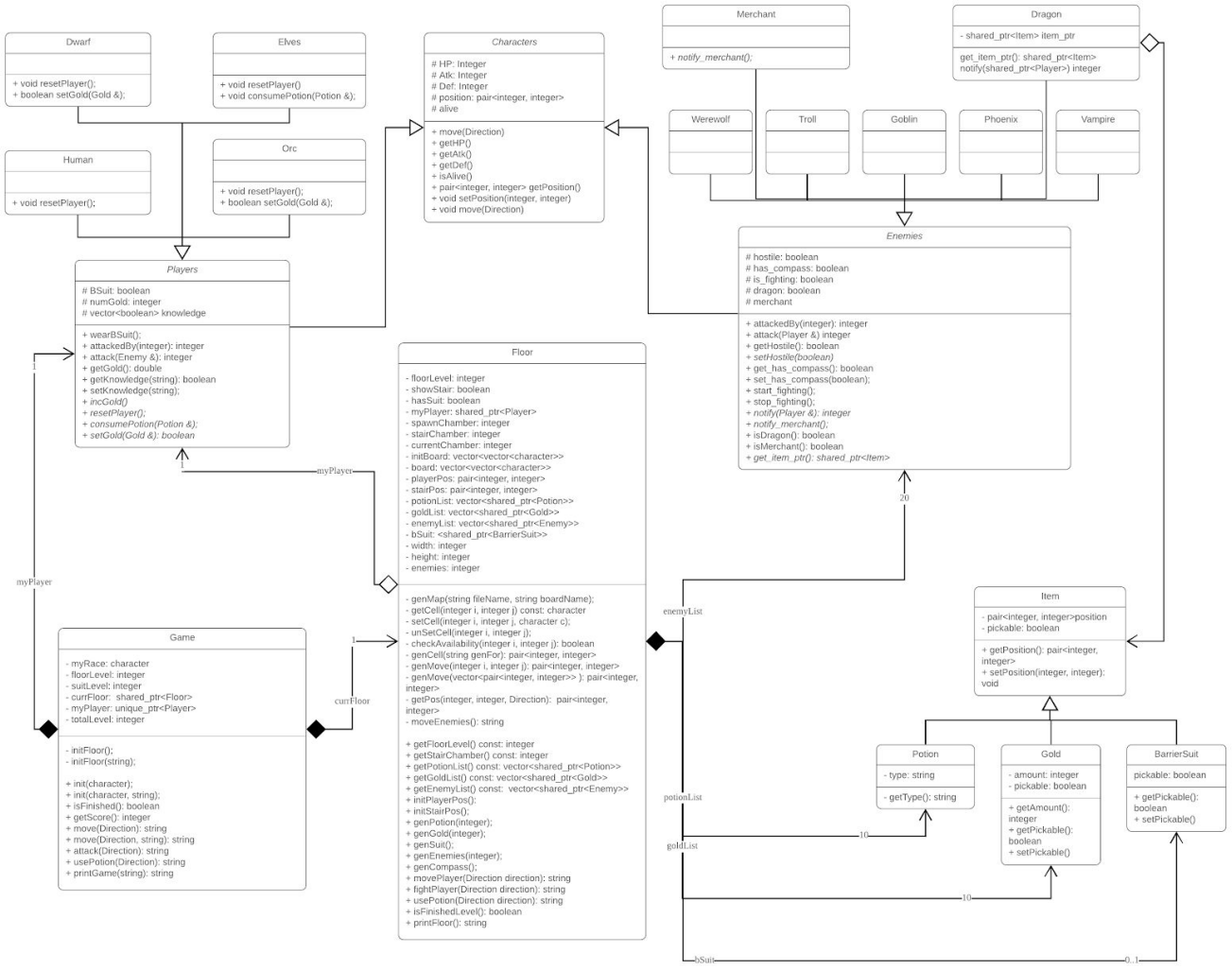
- The Game's method getScore() calculates the score by calling the Player method getGold(); The Game's method isFinished() determines if the game is finished

through myPlayer's getHP() method and currFloor's isFinishedLevel() method. The main function continues to take user inputs until the game is finished and displays the score when it is finished.

- The **Floor class (in floor.h)** except for initializing the board and generating **Items** and **Enemies**, the class also implements how myPlayer is interacted with them through the methods listed below:
    - string movePlayer(Direction direction);
    - string fightPlayer(Direction direction);
    - string usePotion(Direction direction);

- Details on the classes breakdown will be discussed in detail in the later section.

Updated UML



Design

- Template Method Design Pattern:

- We used template method pattern in this program. Especially, for Character class and Item class.
- **Character** class has fields including the character position, HP, Atk, Def, and the related setter and getter methods.
- Character class has two children classes: Player and Enemy.
- **Player** has fields including:
    - "numGold": keep track the number of gold the player has;
    - "knowledge": tracks whether the player has used this potion before,
    - "BSuit": telling whether the player wears a barrier suit or not,
- Player has methods including:
    - setter and getter methods for private fields
    - "attack" which takes an enemy's pointer, attack the enemy that is given
    - "attackedBy" which takes the Atk of the attacker.
- The player has subclasses **Human, Elves, Dwarf, and Orc.**
- Using template method pattern, each of them has certain methods overridden.
    - Dwarf and Orc has their gold-related methods overridden
    - Elves has its potion-related methods overridden
    - Every race has its own reset() function which resets their atk and def, eliminates the temporary potion effects from the last floor.
- **Enemy** has fields including:
    - "hostile": enemy attacks player only when it is hostile,
    - "is_fighting": when the enemy is fighting, it won't move;
    - "is_dragon": return whether the enemy is dragon
    - "is_merchant" return whether the enemy is merchant
    - "Attack": takes a player pointer and attack the player given
    - "attackedBy": takes the Atk of the attacker.
- The enemy has subclasses with all kinds of enemy, however, the dragon has an extra field "item_ptr" that points to the treasure it protects, and a method that can accessed its item_ptr field.
- Using the template method pattern, we also set the notify() method to be virtual so enemies can customize their own notify() method.

- **Item** class has two fields:
    - "position", which stores the item's position on the map.
    - "pickable", which stores whether the item is available for the player to pick up.

  Item has three subclasses: Potion, BarrierSuit, Gold.
    - **Potion** only stores a string, indicating the type of the potion. The effects of using the potion is depended on the player's race. The default "pickable" field for potion is always true.
    - **BarrierSuit** does not have extra fields. It's "pickable" is false at default. This indicates that the dragon that guard this suit has not been

slain. When the dragon is dead, it will set the barrier suit's "pickable" as true, then the player can pick it up.
    - **Gold** has an extra field: "amount", which indicates the amount of gold in this pile. If "amount" is not 6, which means the gold pile is not a dragon hoard, then the field "pickable" is true. If it is a dragon hoard, the amount would be 6 and the field "pickable" is false. Similarly, only when the dragon is slain, the hoard can be pickable.
    - Using template method pattern, we let potion class override the getter method for the field "pickable", since the potion is always pickable, while keeping the Gold and BarrierSuit class's getter method returning the actual field's value.

- Observer Design Pattern: enemy_ptr->notify(player_ptr, string &)
    - We used observer design pattern in this program. Whenever the player moves or acts, every enemy in the enemy lists will be notified. We also have certain enemies notify() method overridden, which allows these enemies react differently to the player's actions. For example, vampire can steal health, goblin can steal gold, and troll can restore health when notify function is called.

- Model-View-Controller Design Pattern
    - We used the MVC design pattern by separating the model (**Floor in floor.h**), view (**main.cc**) and controller (**Game in game.h**). First, the main function takes user inputs, and the controller controls the workflow, then the model modifies the current state of the Player and Floor, which returns the updated state to the view to display to the user.
    - By doing so, we follow the single responsibility principle such that each class only has the responsibility of a single part of the game's functionality. Each class only has one reason to change. It makes our program more manageable and easy to explain and understand.

- Coupling and Cohesion
    - Our design tries to achieve low coupling and high cohesion as much as possible. Each class has its own responsibility; for example, each of the Character/Item class and its subclasses is only responsible for updating or returning the corresponding character/item's state or information; the Floor class is only responsible for updating or returning a floor's state; the Game class is only responsible for controlling the overall game flow; the main function is only responsible for taking user inputs and producing output displays.
    - However, we could not avoid some level of coupling since there are some interactions needed between the Player and the Floor. More specifically,

Player with Enemy (attack) and Player with Item (use potion/pick gold). In these cases, it is necessary to take objects from other classes as parameters which increase the degree of coupling. Another example is the Dragon class, it has high coupling with the Gold/BarrierSuit class. We decide to sacrifice coupling in this case because it now enables us to trace the relationship between the dragon and the item.

- Difference between the original and final design
    - In the final design, we added the Game class to be the main control of the game to adopt a MVC design. We believe doing so could help us achieve a better and clearer design.
    - We remove the Chamber class from the original design. We included the class when we first design the implementation because we want it to store the chamber related information, but we quickly realized all the information is already stored in the Floor class, and there is no need to have an extra Chamber class, otherwise the program is not efficient enough.
    - We also added an Item class to act as a base class of Potion, Gold, and BarrierSuit. In our first design, we did not believe there are enough common features among these two items to create an Item class, but we later found out the use of Item class enables us to reuse the code, and makes it easier to manage the structure of our design.

Resilience to Change

- There are a couple of examples to demonstrate the high ability to make changes in our design:
    - The inheritance structure we used for the Character and Item class enables us to easily implement a newly created “enemy” or “item”. For example, if we want to add another type of enemy “Zombie”, we could simply create a Zombie class which inherits the Enemy class and implements its special features by overriding some virtual methods. We could also simply remove a type of enemy from the list.
    - Similarly, the inheritance structure in the Item class also benefits us in the aspect of making changes. For example, if we want an implementation such that a Dragon can hold a potion, we can easily achieve it by passing a Potion object to the Dragon constructor and set the “pickable” field to False without making any other changes, because the Dragon takes an Item object, and potion/gold/barrier suit are all Items.
    - If we want to change the number of enemies, potions or golds to be generated on a floor, we only need to change the parameters passed to the Floor’s generation methods in Game class; we can also easily change the total number of levels in a game by changing the default floorLevel value in Game.

Answers to Questions

- How could your design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes?
    - We had a parent class called Player with each race as its children class. We store each race's default setting in their constructors so that when calling their constructors, we only need to pass in the player's initial position and the constructor will generate the rest of the fields automatically.
    - Using such design, adding additional classes is easy since other classes' methods interact with Player class only, instead of overloading the method four times with each of them interact with one race.

- How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?
    - For a single enemy, the generation is almost the same as a player, since in our design, they are both in the same Character field due to their common features (HP, Atk, Def, Position, etc.).
    - However, there are multiple enemies need to be generated in a game, and each of them has different probabilities.

- How could you implement special abilities for different enemies. For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?
    - We override the notify method for enemies with special abilities. Our notify method is called by an enemy with the player as parameter (enemy->notify(player)). Goblins can access player's gold and steal it through member function, vampire can access player's health and steal it, etc.

- 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?
    - We could use decorator design pattern to model such effects as explained in the plan of attack; however, we didn't use any design pattern for this feature in the actual design pattern. We just simply add a resetPlayer function so when the player moves to the next floor, the player's Atk and Def are reset to default value. We believe it is easier to understand while not breaking any design rules.

- 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 hoards and the Barrier Suit?

- We have a base class Item and subclasses Potion, Gold, and BarrierSuit. We constructed an Item class that has a pair<int, int> field "position". Thus, we just need to pass in the initial position for making a list of such items (and an additional numGold field for gold).
- We constructed an Item class that has a boolean field "pickable". Barrier Suit and Dragon Hoard's "pickable" field are set to be false by default. Only when this field is changed to true, the hoard or the suit can be picked up.
- Therefore, we are able to reuse the code for treasures, and especially dragon hoards and barrier suit.

Extra Credit Features

- No new and delete: we use unique_ptr and shared_ptr so that we don't need to manage memory ourselves.
- Add special abilities to some enemies:
  - Goblin has a 50% chance to steal gold from the player
  - Troll has a 50% chance to restore 10 HP
  - Vampire has a 33.3% chance to steal 5 HP from the player

Final Questions

- 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?
  - Discuss as a group first: When developing software in teams, the team needs to discuss and agree on the overall structure of the program first. We found out that in this way, we saved a lot of effort in making sure that each individual's codes are compatible with the whole team.
  - Documentation your work well: Make the function and variable name as descriptive as possible, and comment all the cases. It will be much easier for other team members to debug your code.
  - Well communication: ask whenever you have questions about understanding of the project, design, etc.
  - Do the team project together: Whenever possible, the team should do the project together and talk face to face. This could save much efforts on communication.

- What would you have done differently if you had the chance to start over?
  - We would use visitor design pattern if we had the chance to start over. After finished the project, we found out that it is much easier to add additional bonus features, such as weapons and armor, if using visitor design pattern. We

also won’t need additional fields indicating whether the enemy is merchant, dragon, or other enemy.