Document

Final Project – CC3k

Course	CS 246 - Object-Oriented Software Development
Lecturer	Caroline Kierstead
Student name	Hoang Dang
Student number	21053191

Introduction

In this project, Hoang Dang worked on ChamberCrawler3000 (CC3k), a simplified rogue-like game. The major component of this project is C++ using OOP with different concepts like inheritance, polymorphism, and encapsulation. In addition, the following tools were used along the development process:

- Notion: Checklist, Timeline, Questions to ask Instructor, and important Note.
- Lucid Chart: Overall design of the project, what are the classes and their relationship.
- GitHub [private]: Keep track of progress and changes to code.

Overview

In this project, everything starts from main() which handles execution command from the user and based on the provided arguments (map's name, seed, or game mode), it will use a while loop to call appropriate run() function that handles the whole game flow. This includes displaying welcome message, choosing hero, generating Board that contains of Objects (Enemies, Treasures, Potion, and Stair) and continuously processing commands from the user to perform different actions (Race movement, freeze enemies, re-run the game, or quit the game). When the game ends (hero wins or loses) or user decides to quit, the run() function will return an integer of either 0 or 1 indicating different game states. While 0 will break the while loop and end the program, 1 will perform another loop which allows user to re-play again. This approach allows extension in game states in the future by returning other integer numbers.

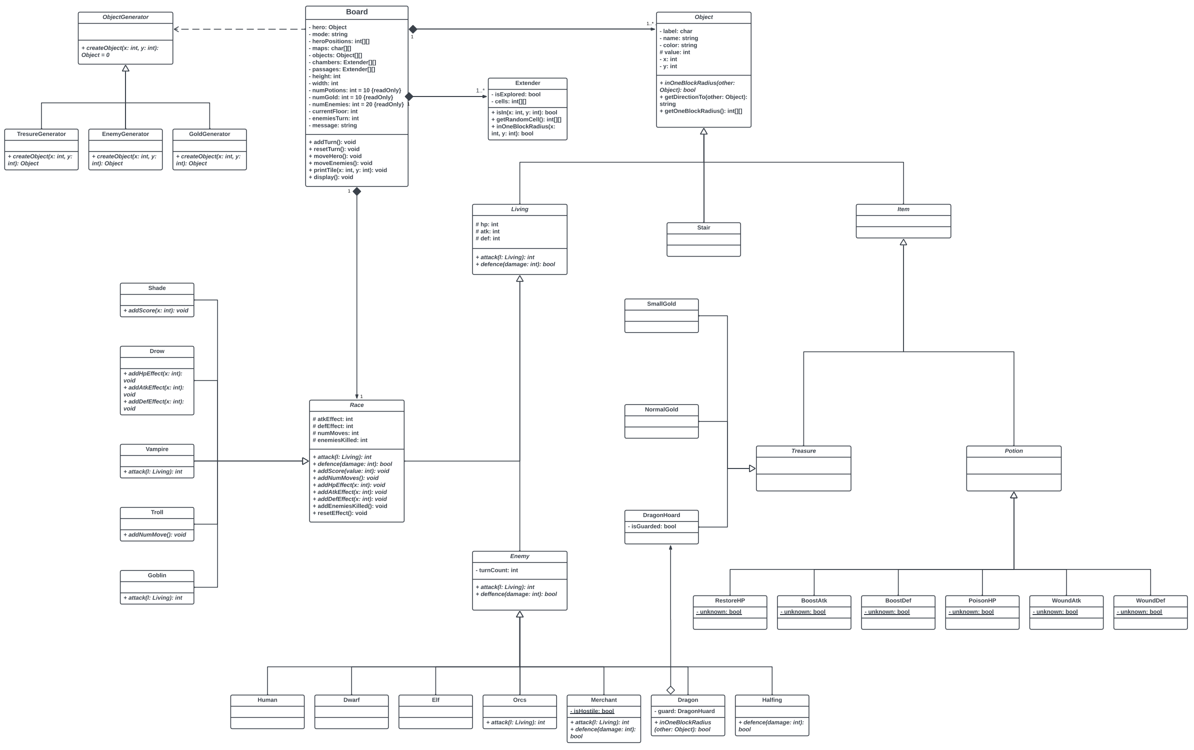
Board is an important component of the whole program, the construction of it goes along with the constructions of all objects and maps in the whole game run. This can be done in three different ways:

- Reading a map with objects specified: builds a map with provided floors and objects layout.
- Reading a file with no objects: If the provided map in upper step doesn't have any objects, the program will perform random objects generation.
- Default: If no command line argument is provided for map, it will use the default one with 5 floors, each floors has 5 chambers, and with random objects generations.

Along the way, it will also uses an algorithm to detect and create appropriate chambers and passages. Three important methods of Board are moveHero(), moveEnemies(), and display(). moveHero() processes direction from user input and perform corresponding actions: normal movement, get to a new floor, pick up Gold, use Potion, or attack an Enemy. moveEnemies() performs enemies' movement in line-by-line fashion. Within one turn, an Enemy will first check if there is a Player Character in one block radius and then decides whether to move or to perform attack. Last but not least, the display() method is used to print the map and objects to the command line, working as a user interface for the game. In addition, an information board is also printed which indicates the PC's name, number of Gold, current floor number, attack and defense values, and an action message.

Lastly, everything occurs on the map other than the layout are considered and belong to the Object class. By using inheritance, Object can either a Living, an Item, or a Stair. Living can be divided into Race (Shade, Drow, Vampire, Troll, and Goblin) and Enemy (Human, Dwarf, Elf, Orcs, Merchant, Dragon, and Halfling); while Item composes of Treasure (SmallGold, NormalGold, and DragonHoard) and Potion (RestoreHP, BoostAtk, BoostDef, PoisonHP, WoundAtk, and WoundDef).

Updated UML class model



What changed:

- Removed Decorator Design Pattern for stacking potion effects
- Added an ObjectGenerator class which uses Factory Method to produces different objects
- Replaced Chamber with Extender class which is used to create chambers and passages
- Added some new methods in some classes

- Based on game design, Merchant Hoard was not needed since Race will go to enemy's location and automatically pick up the gold

Design

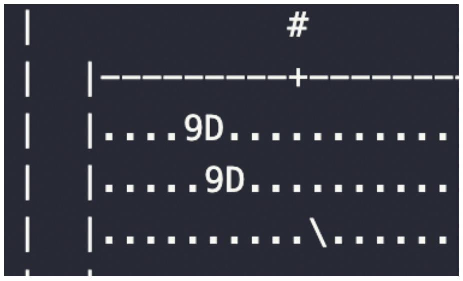
1. Handling invalid input and end of file: Throughout the program, different handlers were provided to handle possible invalid inputs from the users
    - Not existing map: Using is_open() method in <fstream>
    - Seed is not of type integer: Using istringstream in <sstream>
    - Invalid action for PC: Using conditional check
    - Not existing hero: Using conditional check
    - 'q' or Ctrl-D at any time to exit the game: Using .eof() and return statement
    - Etc.

2. Chambers/Passages detection (Extend.h): In order to make the program flexible with different map design, an Extender class was used to detect which floor tiles belong to a particular chamber or passage. This idea is from CS 234 where I use a stack to search for possible path in a graph, in which from one node I can expand to the others. Applying this in CC3k, I keep track of a list of chambers, whenever I find a floor tile with label '.', if it is not in the list of existing chambers, I will create a new one with that particular floor tile. Along the construction of that object, it used depth first search strategy using stack to keep expanding the space as much as possible and eventually achieve the desired chamber.

3. Objects generation:
    - When reading from a file, I used *makeObjFromLabel()* function to create Objects based on their label.
    - Random generation: When no objects are provided, the program will perform random generation using rand() in <cstdlib> and create Object based on their proportion as discussed in project specification. In addition, *getSpawnPlace()* in board.cc is used to return an available random floor cell in a random chamber for the new objects.

4. Dragon and Dragon Hoard: When reading objects from file, one challenge is to correctly link Dragon with its corresponding Dragon Hoard. Possible edge case may occur when

there are two Dragons next to each other and we incorrectly connect two middle one together first



```
|            #
|  |--------+-------
|  |....9D...........
|  |.....9D..........
|  |..........\......
|  |
```

My approach for this is to generate all possible combinations of Dragon and Dragon Hoard in the current floor, and as soon as there is a combination satisfies the condition (Every Dragon has its own Hoard), the function will stop. While for random Objects generation, it is easier since as soon as there is a Dragon Hoard, I will also create a Dragon to guard it. A Dragon will have a pointer to its Dragon Hoard, and Dragon Hoard will have a Boolean field isGuarded. When a Dragon is killed, it will uses that pointer to update the status of Dragon Hoard so Race can take it.

5. Managing maps and objects layouts (board.h): In my implementation, floor layout was stored in a vectors of char, while objects layout was stored a vector of Object pointer.
    - This approach allows me to perform enemies' movement in line-by-line fashion by using a nested for loops. In addition, to avoid enemies from moving more than once in one turn, I used an additional field to keep track of number of movement.
    - Whenever an Object is removed (Gold is picked up, Potion is used, or Enemy is killed), I will use the x and y fields stored in that Object to update the corresponding position in Object vector to a nullptr and thus delete that particular object.
    - In terms of Enemy's movement, I will use these two vectors to generate a list of possible blocks to go (Enemy can walk on and no Object is already on that tile), and then randomly choose one between them. If there is no possible move (all blocks are occupied), enemy will stay still.

6. Unknown Potion and Hostile Merchant: since a potion is not known until it is used for the first time and all merchants will be hostile as soon as one of them being attacked, I used a static field to keep track of their status in order to perform reasonable actions. However, since we can re-play the game again, it is important that we reset those fields back to the

initial status before the current game end so they will be correctly used in the next run, in destruction of Board, I create a temporary objects and correctly reset those fields.

7. Attack & Defense: *Attack()* and *Defense()* functions are virtually implemented in the Living class. The attack function takes an Object as the parameter, then performs damage calculation before calling the target's defence() with input as the damage. By using Polymorphism, some Living who have special abilities in attack and defence (deal more damage with Goblin, 50% of dodging attacks, etc.) can override Attack() and Defence() functions in its superclass.

Resilience to Change

1. By using inheritance, it will be easy for me to add a new Race, a new Enemy, or a new Item just by creating a new concrete subclass.
2. Polymorphism is also used throughout the project with different virtual functions to apply different abilities of Race and Enemy:
    - addScore() will double the score for Shade
    - defence() will have 50/50 chance of dodge attacks for Halfling
    - attack() will check if target is killed and add 5 golds to Goblin
    - addTurn() besides from adding 1 turn will also add 5 HP for Troll
    - attack() will add 5 HP to Vampire every time called
    - etc.

   If there are chances in any of the abilities, I can just override the function of the subclasses.

3. By using Factory Method design pattern, I can create different concrete subclasses of the abstract factory for different generation based on game levels.
4. By using encapsulation, most of the fields are set in private and can only be accessed via provided functions, which lessen the proportion of unspecified behaviors.
    - Outers cannot set the HP for Living directly but have to use setHp() method which was implemented to ensure Hp is not > Max Hp.
    - Adding potion effects also used the same strategy to ensure Attack and Defence of PC is not less than 0.
    - Board doesn't know anything about how a Living attacks other

   Low coupling ensures that when there are changes, I don't have to change much in other places.

5. The program also ensures high cohesion in different classes
    - Race class has attack(), defence(), addScore(), etc. that work closely to each other.
    - Object class has inOneBlockRadius(), getDirectionTo(), and getOneBlockRadius() which are all related to direction.

Answer 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 races?

Answer: By using inheritance, I have a base class Race that has common fields and behaviors shared by all heroes in the game. For each of the hero, I will create a derived class that inherits from Race, and within each class, contains race-specific attributes and behaviors. When choosing race for the game, based on the command from the user, the program will create corresponding hero and use a Race pointer to pass the object in different locations. In addition, this approach allows me to easily add different heroes just by making additional derived classes.

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

Answer: Once again, I used inheritance to have another base class Enemy that contains common fields and behaviors between different enemies and a set of derived classes for each of the specific kind. However, when it comes to enemies generating, it would be different from how I generate the player character. While the PC is chosen by the user at the beginning of the game, the enemies are generated using Factory Method design pattern and based on the randomization proportions as discussed in the project specification. For example, Human will correspond to 0 to 3, Dwarf corresponds to 4 to 6, ..., and finally Merchant corresponds to 16 and 17. I will then pick a random number between 0 to 17 for 20 times to generate 20 enemies. After that, all of the enemies' pointers will be stored in a vector for later use (move, attack, die, etc.).

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: For the various abilities of enemies, I tried to implement them directly in the enemies' concrete subclasses as much as possible to ensure encapsulation and polymorphism. This allows me to customize different behaviors of enemies like attack or defense. For example, when Orcs attacks, it will check whether the Player is Goblin to increase to increase 50% of damage; or when Halfling defenses, it will have a 50/50 chance of dodging the attack of PC. However, there are some abilities that I have to implement differently like for Merchant, I will have a static field to keep track of whether it is hostile to PC. The abilities of player characters were also implemented in the same way.

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, discussing the advantages/disadvantages of the two patterns.

Answer: In my opinion, both Decorator and Strategy patterns can be used to apply the effect of potions. While Decorator pattern allows me to stack the effects of potions on top of each other. The Strategy pattern allows me to only need one Strategy object which keep track of changes in Attack and Defense value. **However**, I think it would be simpler to has two additional fields in the Race class: attackEffect and defenceEffect that keep track of changes in Attack and Defense; and whenever the player goes to a new floor, the program will just need to reset those fields.

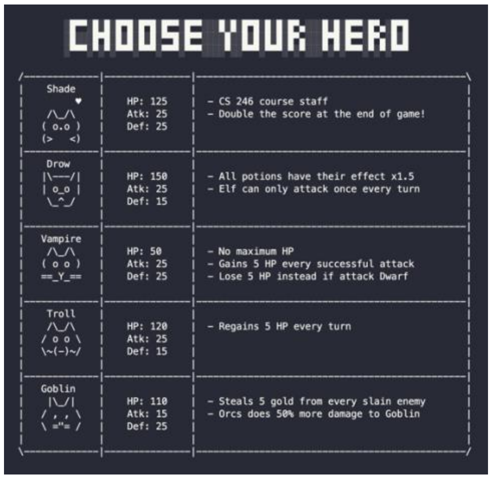
How could you generate items so that the generation of Treasure and Potions reuses as much code as possible? That is, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code?

Answer: In order to make the generation of Treasure and Potions reuse as much code as possible, I used Factory Method design pattern. This allows me to have an abstract factory that has an objects generation strategy and its concrete subclasses which can be used to produce different objects of different types that are from a same base class. In addition, this approach allows me to easily add more objects in the future without changing the code of the base factory class.

Extra Credit Feature

8. Throughout the project, only smart pointers and vectors were used for memory management and there is no delete statement in the whole program.

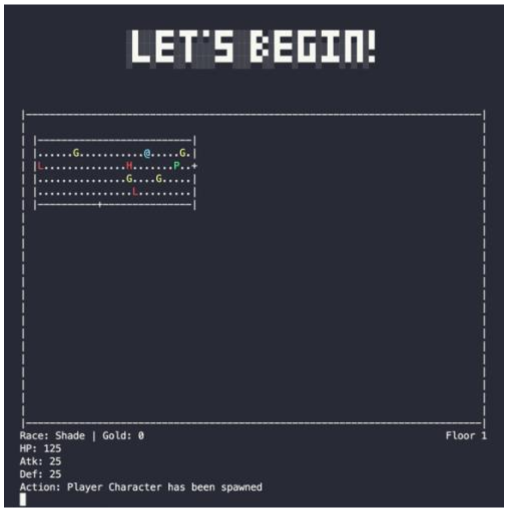
9. UI design: Game beginning, hero choosing, and game ending were supported with ASCII design to improve user experience.



```
CHOOSE YOUR HERO

/--------------|--------------|----------------------------------------\
|    Shade     |              |                                        |
|          ♥   |   HP: 125    | - CS 246 course staff                  |
|    /\_/\     |   Atk: 25    | - Double the score at the end of game! |
|   ( o.o )    |   Def: 25    |                                        |
|   (>   <)    |              |                                        |
|--------------|--------------|----------------------------------------|
|    Drow      |              |                                        |
|   |\---/|    |   HP: 150    | - All potions have their effect x1.5   |
|   | o_o |    |   Atk: 25    | - Elf can only attack once every turn  |
|    \_^_/     |   Def: 15    |                                        |
|              |              |                                        |
|--------------|--------------|----------------------------------------|
|   Vampire    |              |                                        |
|    /\_/\     |   HP: 50     | - No maximum HP                        |
|   ( o o )    |   Atk: 25    | - Gains 5 HP every successful attack   |
|   ==_Y_==    |   Def: 25    | - Lose 5 HP instead if attack Dwarf    |
|              |              |                                        |
|--------------|--------------|----------------------------------------|
|    Troll     |              |                                        |
|    /\_/\     |   HP: 120    | - Regains 5 HP every turn              |
|   / o o \    |   Atk: 25    |                                        |
|   \~(-)~/    |   Def: 15    |                                        |
|              |              |                                        |
|--------------|--------------|----------------------------------------|
|   Goblin     |              |                                        |
|    |\_/|     |   HP: 110    | - Steals 5 gold from every slain enemy |
|   / , , \    |   Atk: 15    | - Orcs does 50% more damage to Goblin  |
|   \ ="= /    |   Def: 25    |                                        |
|              |              |                                        |
\--------------|--------------|----------------------------------------/
```

10. Game stat: Displaying statistic after the game.
11. Users can create their own maps (can be tested by running `./cc3k big.txt`)
    a. Not restricted to 5 chambers.
    b. Not restricted to provided chambers/passages design.
    c. Not restricted to height of 25 and width of 79.
    d. Not restricted to 5 floors in total.
12. Using seeds (`./cc3k -s [seed]`): A seed can be provided when running the game to ensure randomizations to be the same. In addition, if no seed is provided, a random seed will be generated and stored in data/seed.txt if user want to later play the same game again.
13. Hard mode (`./cc3k -m hard`): Player only knows the layout of chambers and passages if he/she has been there before, otherwise they are all blank!



```
LET'S BEGIN!

|-----------------------------------------------------------------------------|
|                                                                             |
| |--------------------------|                                                |
| |......G...........@.....G.|                                                |
| |L..............H.......P..+                                                |
| |...............G....G.....|                                                |
| |................L.........|                                                |
| |-----------+--------------|                                                |
|                                                                             |
|-----------------------------------------------------------------------------|
Race: Shade | Gold: 0                                                    Floor 1
HP: 125
Atk: 25
Def: 25
Action: Player Character has been spawned
```

Note: Different commands arguments can be provided in any order when running the program, here are a few examples:

14. `./cc3k`: default map layout with 5 floors, each floor has 5 chambers, and random objects.
15. `./cc3k cc3k-5floorsWithItems.txt`: generate map that follows the design of the provided file (if the map is empty, the program will randomly generate new objects).
16. `./cc3k -s 1000 -m hard`: run the program with the seed of 1000 and play in hard mode.
17. `./cc3k big.txt -m hard -s 123`: using the layout of big.txt, play in hard mode with the seed of 123.

Final Question

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: In my opinion, planning is the most important stage in this project. I have spent nearly a week since the distribution of CC3k Specification just to read and understand the game. All important details were highlighted, all unclear parts were noted down to later ask the instructor, and lastly all ideas about what I was going to do were kept track in Notion. After that, the coding part was really fast since throughout the process, I knew what I was doing and what I needed to do next.

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

Answer: If I had a chance to start over, I would try to focus more on my work cause sometimes I got distracted by other small stuff (have a chess match, watch some YouTube videos) and thus slow down the development process and had to stay up late many nights. Doing so also affects my work quality since I sometimes forgot what I was doing, and thus had to go through many things again.

Conclusion

I have been learning CS for nearly two years, but CC3k was actually my first ever project. I love it and will definitely do a lot more in the near future!