CS246 Assignment 5
Spring 2017
Final Report
ChamberCrawler3000 - CC3K

Jingwei Huang
Ruitao Lai
Xiaoxiao Fang

<u>Part 1: Introduction & Overview</u>

- ***Implementation of the Game***
    - After carefully reading the game specification, we found that the idea of Observed Pattern and the classes such as Cell, Grid and TextDisplay from Assignment 4 Question 2 is helpful for us to get start implementing CC3K. We adopted the idea and generated square class corresponding to the cell, floor class corresponding to grid and gameBoard corresponding to the TextDisplay.
- ***Use of design patterns & object-oriented design techniques***
    - We used Observed Design Pattern which is similar to what we used for Assignment 4 Question 2. When the player or enemy moves, they need to notify the gameBoard so that it will display a correct map after each action.
    - We used Visitor Design Pattern between enemy and PC classes. Every subclass of enemy has beAttacked(PC& somePC) method and every subclass of PC has beAttacked(Enemy& someEnemy) method. Besides, when the enemy attacks, it will simply call PC's beAttacked method and when the PC attacks, Enemy's beAttacked method will be called.

<u>Part 2: Updated UML</u>

- The actual UML is not much different from our original UML. We added some more methods and fields for some classes when needed.
- The actual UML contains get/set methods and private fields.
- Please refer to the separated pdf file for the UML diagram.

<u>Part 3: Design & Classes Breakdown</u>

<u>Character class is an abstract class which takes care of the common fields of enemy and PC.</u>

- **Character - Character.h & Character.cc**
    - **Subclasses:** PC.cc/h, enemy.cc/h

<u>PC class has six subclasses representing different races of player characters from which the player of this game can choose one to start the game.</u>

- ***<u>PC - PC.cc/h</u>***
    - **Subclasses:**

        - Shade.cc/h, Drow.cc/h, Vampire.cc/h, Troll.cc/h, Goblin.cc/h, Fairy.cc/h
    - **Highlight:**
        - Troll is different from other subclasses in the way it gains HP. It regains 5 HP every turn.
        - Drow is attacked only once every turn by elves, while other races are attacked twice.
        - Vampire is allergic to dwarves, so it loses 5 HP rather than gain every time it attacks a dwarf.
        - Halfling has 50% chance to miss in the combat.
        - Goblin gets 50% more damage from orcs.

<u>Enemy class has seven subclasses which represents different types of enemy existing on the board.</u>

- ***<u>Enemy - enemy.cc/h</u>***
    - **Subclasses:**
        - human.cc/h, dwarf.cc/h, halfling.cc/h, elf.cc/h, orcs.cc/h, merchant.cc/h, dragon.cc/h
    - **Highlight:**
        - Dragon’s move method was overridden because dragon will not move in our game.
        - Dragon’s attack method was different from other subclasses because Dragon will attack PC when Dragon detects that the PC is within 1 radius of both dragon and dragon hoard place.
        - Merchant has a new field named isHostile. Once the PC attacks him, it will be set to true and from now on Merchant will attack PC when they meet.

<u>Item class has two subclasses representing different types of usable items in this game, including potions and gold.</u>

- **Item - Item.cc/h**
    - **Subclasses:**
        - Potion.cc/h, Gold.cc/h
    - **Highlight:**
        - Gold has an extra field named available, which is used to keep track of whether the gold is available to be picked up. In specific, dragon-hoard is available only when the dragon is killed by the PC.

<u>Square class is the superclass of character class and item class</u>

- **Square - square.cc/h**
    - **Subclasses:**
        - character.cc/h, item.cc/h
    - **Highlight:**

        - Square class is used to represent every square in the given file or in our default file. It can just represent a floor tile, a doorway, a passage, a wall that does not have character or item on it. It can also represent player character, enemy, potion or gold by using inheritance and dynamic dispatch. It is an important component of our floor and the whole program.

<u>Floor class is responsible for generating the game board every time. It supports two modes, one with a given file and one without.</u>

- **Floor - floor.cc/h**
    - **Highlight:**
        - In the initialization function of the game board without the file, we write functions to randomly generate the location of enemies, player character, potions and golds. In order to realize this, we store the available positions in five chambers in a field called theRoom and erase one position from the chamber after generating one character or item.

<u>Gameboard class is a class that acts like textdisplay which make responses once pc or enemy moves.</u>

GameBoard - gameBoard.cc/h

- **Highlight:**
    - Gameboard class is a simple class that is similar to textdisplay class in assignment 4, question 2. It has a field that is type of vector<vector<char>> to redraw the game board every time after a command.

<u>Part 4: Resilience to Change</u>

Our design supports the possibility of various changes to the program specification for the following reasons:

- ***Hard coding avoided as much as possible:***
    - In our code, we tried to assign variable names to numbers that we needed instead of using raw numbers.
    - Besides, we used effective algorithm to solve the problem
- ***Clear documentation and comment:***
    - In Enemy's different subclasses, we put detailed comment about why the specific subclass acts differently.
    - For example, the dragon will be able to attack the PC within both his and hoard's 1 radius range.
- ***Well-designed and Object-oriented program:***

    - We used different observed design pattern to implement the program, which is the main reason for resilience to change.

Part 5: Answers to Questions

1.Player Character

Question:

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

We will solve this problem by using *public inheritance*, i.e., creating a player character base class, with each race as a subclass, including shade, drow, vampire, troll and goblin. With such a solution, we can add additional races by simply adding additional subclasses and overriding virtual methods.
We used the same method for due date 2 program.

2.Enemies

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

Our system handles generating different enemies using *public inheritance*. It's the same approach as generating the player character because the generation of the player character and enemy follows similar pattern: different races share similar properties, yet there are subtle differences between those properties; for example, races classes all have HP, Atk, Def fields, but they differ in default values. So does enemy class. Thus, we can define a base class to contain those similar properties, and then specify for a particular race in corresponding subclass.
We used the same method for due date 2 program.

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.

We could use the *Visitor Design Pattern*, which is used to implement double dispatch. Also, we do use the same techniques as for the player character races.
We plan to design two pure virtual methods for player character and enemy, attack(Defender) and be_attacked(Attacker), and override these methods in subclasses to support their various abilities. Each time an enemy attacks, we use the type of the enemy (the Attacker) to determine which specific attack(Defender) gets called, and the type of the player character (the Defender) to determine which be_attacked(Attacker) gets called. The same thing happens when the player

character attacks. That is, we need to know the types of both the attacker and the defender at runtime to select the correct methods.
We used the same method for due date 2 program.

3.Items
3.1 Potions
Question. 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 weighing the advantages/disadvantages of the two patterns.

Design Patterns	**Advantages**	**Disadvantages**
Decorator Pattern	● Allow the system to determine the specific type of the potion at runtime. ● No need to explicitly track which potions the player character has consumed. ● A good solution to permutation problems because of the ability to wrap a player character with any number of potions	● Creating and updating the component(player character/ player character + potions) can be complicated because you need to wrap the component in many decorators(potions)
Strategy Pattern	● Instantiating component comparatively easier because only switching among different strategies is needed ● Allow the system to determine the specific type of the potion at runtime. ● No need to explicitly track which potions the player character has consumed.	● A huge permutation issue may be raised when there are many possible states with different behavior - you will have to write strategy (i.e. subclasses with different algorithms) for every single possibility.

Since there are six different types of potions, we cannot handle all possible combinations of known and unknown potions by simply writing a small number of strategies. Thus, implementing using strategy pattern can be complicated. In conclusion, the decorator pattern would work better.

For our game, we didn't use decorator pattern to implement item class. Instead, when the player character uses a potion, his fields will be changed to response different effects.

3.2 Treasure

<u>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?</u>

In order to reuse as much code as possible and prevent duplicate code, we plan to put Treasure and Potion under the same superclass called Item.
We used the same method for due date 2 program.

<u>Part 6: Extra Credit Features</u>

- ***ASCII Arts***
    - Our program has a welcome page
    - When player character loses, a unique ASCII Art page will show up.
    - When player character wins, a unique ASCII Art page will also show up.
        - The challenging part of the feature is that we need to carefully consider where to put the art page and using ifstream to read in file properly.
- ***Merchant has the ability to sell different types of potions.***
    - Merchant will only sell three different types of boost potion: RH(restore health by 10), BA(boost attack by 5), BD(boost defence by 5)
    - When the Player Character buys a potion, it need to tell the merchant the direction and a potion type.
        - The challenging part of the feature is that we need to check if the health will exceed the maxhp or not while attack and defence value have no upper bound.
- ***New Player Character Race - Fairy***
    - She has more defence and less attack default value.
        - The challenging part of the feature is that creating a new PC subclass is needed and we also need to include different heading files.

<u>Part 7: Final Questions & Conclusion</u>

<u>Question 1. What lessons did this project teach you about developing software in teams?</u>

Lessons that we learnt from developing software in a team:

- ***Sharing ideas with your team member is important.***
  - Each member has their own thoughts based on their knowledge. Sharing the idea might lead up to bring up a solution for a specific question.
  - For example, while going through the game specification, Xiaoxiao mentioned that the idea of the game is really similar to Assignment 4 Question 2, which helped all of us as a momentum to have a basic knowledge of the game.
- ***Distribute task correctly can maximize group productivity.***
  - Different team member was assigned different task which fits their ability and interest.
- ***Communication is the key of success.***
  - Since the Enemy class and PC class are closely related to each other, decide which how virtual method was arranged is important before starting coding.
  - After our discussion, we have decided to implement beAttacked method for specific enemy/PC.

<u>Question 2. What would you have done differently if you had the chance to start over?</u>

Here are three things that we would have done differently if we had the chance to start over:

- ***Planning before coding***
  - When we are writing our code, we found that a few parts of our original design cannot work as we expected. We should have put more time to deeply discuss different class method and how they are related to each other.
- ***Compiling right after producing a small piece of code***
  - We realized that we spent a lot of time while compiling our program. Many mistakes were simple so that if we compile as we go, the problem can be easily avoided.