Attack of Plan For Project ChamberCrawler3000

Partner 1: Stanley Yip(WatID: syipmanl)
Partner 2: Xingyu Lai(WatID: x7lai)

Implementation Priority Order:

1. Interface

    a. "main.cc"(Deal with arguments)
        i. Can enter default Mode
        ii. Can load default map
        iii. Can enter Test Mode
        iv. Can enter onus Mode
    b. "gamesystem.h", "gamesystem.cc"
        - Interface of the program
        - take command from standard input
        - Output on standard output on screen (and open xwindow for bonus)

2. Floor

    - "floor.cc", "floor.h"
        - Contains cells
        - interact with other classes

3. Character

a) Character class(abstract super class)
b) Player_Character class & Enemy class(subclass of Character class)
c) Classes for each race in player character and enemy(e.g. class Shade, Dragon)
d) Items
    1) Potion class and each kind of potions
    2) Treasure class

4. Interaction

a) Moving
- Let characters move validly
b) Attack System
- Basic attack and defense system
- Handle special abilities of characters

II. Estimated Completion Dates(haven't decided yet)

1. Task 1: July 18th
2. Task 2: July 19th
3. Task 3: July 20th
4. Task 4: July 21st

III. Work Assignment For Each Student

For the important structure and big class we work together, for subclasses like Shade, Dragon we may do them separately.

Question and answers:

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 plan on have a super class called player character which captures(encapsulates?) all fields that are similar amongst the races, this includes hit points, attack, defence, gold count, and some common methods attacking. Each race will be a subclass of character, and will only be requiring a constructor that constructs the information pertaining their stats. In other words, we will be taking advantage of polymorphism of how each race is a character. In terms of uniqueness, relating to their special abilities, certain functions will be overloaded accordingly. For example, vampire will have its attack function overloaded as it gives +5HP for a successful attack.

A solution to make adding additional races will not be difficult, as mentioned above, all it needs is to be a new subclass of player character, with the necessary information of its fields, and the necessary overloaded function.

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

We will handle generating different enemies in a sequence of steps. First, using the cstdlib library provided, we will generate a random number called temp between 1-18. Temp will let us know what enemy will be generated:

Temp within range: Generate this enemy
1 – 4: Human
5 – 7: Dwarfs
8 – 9: Elf
10 – 11: Orcs
12 – 13: Merchant
14 – 18: Halfling

Once the enemy type is decided upon, next using a similar algorithm, we place this enemy on a random floor tile on the dungeon.

This will be different from generating the player character in a couple of ways. One is that enemies will be generated randomly, leaving the player no control over what is spawned. In contrast, player character would be generated with full control by the player as it is handpicked

at the beginning of the game. Another difference includes a small restriction on which a player character can only be spawned in a chamber that doesn't not have a stair way, however enemy has this freedom to be generated so. Other than these points, each floor tile in the dungeon will have an equal chance of being spawned on by both player character and enemy, and neither cannot be spawned on anything except for a floor tile.

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

There are two sets of abilities, ones that doesn't not require the restriction on which can only be activated between the interaction of two unique characters, shade's ability is an example. This will be implemented with overridden subclass methods as we would for all of player character abilities mentioned before.

On the other hand, ones that do require two unique characters, for example vampire attacks dwarf is the only case when dwarf ability is triggered. This will be implemented using the visitor pattern, there will be a double dynamic dispatch going from a super class to a subclass to calling a method of another class. For example, as mentioned before with the vampire example, a super class character will dynamically dispatch to vampire's attack method, and this will dynamically dispatch to dwarf's method that makes vampire -5HP instead of +5HP.

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.

Both patterns are similar in a sense that in many cases, they accomplish the same goal. However, a key difference to consider here is how they are implemented. For decorator pattern, we are adding to existing functionality with extra functionality on top of it. An example of this is the scrolling on menus. On the other hand, strategy pattern relies heavily on allowing the program to actually change the functionality or implementation of a behaviour at run time.

Some advantages of the strategy pattern is to actually allow separation of algorithms into classes that can be used during run time(dynamically) multiple times, this improves readability and reduces duplicate code. To put another way, the families of algorithms created by the strategy pattern provides a new and improved way of sub classing. As this is chosen dynamically, the client must be aware of the available and different strategies, otherwise bugs may occur.

Like a scroll button on a menu, decorator pattern offers many advantages. One is that it allows that it allows certain functionalities to be used only when it needs to be used. For example a scroll button will only be needed if the page becomes too long. This saves a lot of unnecessary work compared to if you always had a scroll bar. A disadvantage could be that since all decorator subclasses are so similar, sometimes it can become hard to distinguish one to another since they only vary in so little ways. A common disadvantage between the two of these is the need to increase the number of objects, which could be troublesome.

After careful analysis from above, we believe that decorator pattern would work slightly better in this case, one reason being that the disadvantage of there being too many similar decorator subclasses will not be a problem here, since we will only be detailing with two subclasses of treasure and potions. Another reason is that we think the implementation of potions and treasure does not need much rigours algorithm to be implemented, therefore we do not see much of a need to create an entire family of algorithms to accommodate for this. Instead we will just treat this as simple ‘extra’ functionalities for the character class.