CS 246 A5 Group Project: CC3K Villain

Group Members:
Austin Hardy
Andy Rock
Pratik Choudhary

Breakdown of the Project:

The Character Class:

The Player and Enemy classes branch off from the Character class. This is an abstract class that we have created to hold many of the common fields and functions of the characters in the game. It has to hold the basic attributes such as HP, Atk, Def, Location, display character as well as several methods which includes accessors and mutators for the above mentioned attributes. It also holds several virtual and pure virtual methods such as: accessors for base HP, base Atk, and base Def values. It also holds a move function since almost all subclass objects of the Character class need to be able to move, we implemented the move function in the Character class. Similarly, we made two attackedBy functions, taking two different parameters, one taking a Character pointer and the other taking a Vampire pointer. These functions are common in most of its subclasses, so it was also implemented in this class. It also has a friend function with the operator overloaded function for "<<". This function simply prints out the display character of the character objects.

In summary, this class holds all the generic features of any character to allow maximum code reuse. However, some of these functions are virtual because they have different implementations for special cases with special characters. In which case, these functions are overridden in the subclasses.

The Player Class:

The Player class contains of specific contribute not common with the Enemy class. This attribute is the gold value. So, the player class holds attribute for the gold value as well as the accessors and mutators for it. It also has virtual methods for drink potions, attackedBy Elf, attackedBy Orc, and killed Enemy. The drink function is partially virtual because all potions effect each player race in a similar manner except for the Drow class, where it is magnified by 1.5 times. Similarly, the damage caused by Elf and Orc are similar to all characters except Drow and Goblin respectively. Also, since when an Enemy is killed, based on the Player's character, it has a certain behaviour, so it is a virtual method for most classes and overriden in the Goblin class because, goblin gets 5 gold extra for every slain enemy. Hence the method is written in the player class and then overridden respectively in the Drow and Goblin classes to match with the appropriate behaviour.

The Enemy Class:

The enemies are broken down in a slightly more complicated manner because of the larger range of types of behaviour of the enemies. We have divided the Enemy class into two subclasses: GivesGold and DropsGold. These are also abstract classes, like the Enemy class, but classifies the enemies based on how they behave when they die. So, GivesGold has

subclass for Dwarf, Orc, Elf and Halfing. When they die, the gold is automatically added to the player's value. So, to add that feature into the code, the GivesGold class has a function called KilledBy Player and KilledBy Goblin. We have two because goblin gain an additional 5 coins per slain enemy. Similarly, DropsGold, has subclasses for Human, Merchant and Dragon, also has the two KilledBy functions. However, instead of transferring all the values of the gold to the player's value, it drops the coin objects in the cell.

The Enemy Class also holds virtual functions for hostile. Since all enemies are hostile in nature except for Merchant and Dragon, they have overriding functions for them.

The Item Class:

This is a generic class that has subclasses Gold and Potion classes. It has the attributes value and collectible. It also has pure virtual methods affect for Enemy and affect for Drow because Gold and Potions affect the players differently and Potions also have magnified effects on Drow.

The Gold Class:

This class overrides functions for affect and the "<<" overload operator is also a friend function of this class. It has 4 subclasses, small, normal, merchantHoard and dragonHoard. These subclasses have constants attributes in them to store the specific values of each gold type.

The Potion Class:

This class is an abstract class with 3 subclasses: AtkMod, HpMod and DefMod. These classes implement the affect functions for Player and Drow to make them concrete classes. They mutate the value of the the attributes of the character class for the player and drow objects.

The Floor Class:

This class represents an instance of one of the five levels in the Game. When created, the init function randomly places the Player, and randomly spawns all characters and items onto itself. Floor serves as a layer through which Game can interact with the current state of the level. It provides functionality for checking certain characteristics of the level, such as whether the player is on the stairs, and which cell is where. Additionally, it is the interface that game uses to execute actions, through functions like *use_potion*, *enemy_move* and *attack*.

The Cell Class:

The is an abstract class which represents a single cell on the game board. It stores information such as its location, and what items are currently on top of it. It consists primarily of virtual methods which query or update info about the cell (i.e. its location, type of cell, what is on it, etc). It is subclassed by all the types of cells (Floortile (.), Wall (-, |), Stairway (/), Passage (#), Door (#), Empty ( ). These subclasses override the virtual methods in cell and implement their own specific functionality. Cell overrides the << operator to allow for the easy printing of cells, which in turn calls the print function overridden by each of the subclasses.

Design Patterns:

Visitor Design Pattern: This design pattern is the most used one in this project. We have been using this throughout the project. We have used it mostly to cover up special cases and edge cases in the game. Firstly, we use it when we deal with attacks between the Player and Enemies. Both classes have the attackedBy functions that take in a Enemy pointer or a Player pointer appropriately. However, since there are edge cases where different enemies have different impacts on various player races, there are visitor patterns for it to address those cases. Killed and KilledBy functions work the same way. Collecting Potions and its effect is also done a similar way, where we have a special case for Drow.

Factory Design Pattern: This design pattern was used for constructing Players and Cells. Both of these can be constructed from single characters. Players are constructed from their representation on the board (i.e. 's' -> Shade). Using the factory pattern, we abstracted this logic into the player class by making a static function Player::Create(char), which consumes a char and returns an instance of the subclass of Player that the char refers to. I.e. Player::Create('s') yields a new Shade. We did precisely the same thing for the Cell class. I.e. Cell::Create('#') yields a new Passage.

How does the program accommodate to changes:

There are several steps taken to make our program easy to accommodate changes:

1. We avoid hardcoding values. We have created a file called constants.h. This file contains all the constants that we use throughout the project and we can simply change the values there and it will change throughout the program. This is incredibly helpful since there are a lot of files in the project and changing constants by going one by one through all the files can be time consuming and tedious.
2. The structure of the Character class and its subclasses are in such a way that we can easily create a new class for Player or Enemy without a lot of complicated changes in the code. There will be mostly code addition in terms of adding special visitor design pattern methods based on the requirements of the character race.
3. We can make this game a multiplayer game easily because of the way we treat each player object. Since we are dealing with pointer to Players and have developed good object oriented style, we can add another player to the gameplay without changing any behaviour in any class except the game class to simply instantiate and spawn the new sets of players.

Questions and Answers:

Question 1:

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

Answer:

There is a Game class that holds a Player Character pointer. The Player character is organized such a way where there is a Player Character abstract class that has 5 concrete subclasses, each representing a race of the player character. Hence, when the user chooses the race using one of the commands: **s, d, v, g,** or **t,** the player character pointer will be initialized with the appropriate subclass player object, to be stored in the heap. Once the player is initialized, the location of the player is then generated based on the location of stairway such that they are generated in different chambers. Otherwise, the player is spawned in a random cell in the floor which however, has to be within the chamber and not a wall or passage.

When creating an additional race, we need to create a new subclass to the abstract Player Character class and implement the virtual functions appropriately to make sure the subclass is concrete. After that, all we need to do is add another command to specify the race for the input interface and fill out some of the visitor design pattern methods based on the special features of the player.

Question 2:

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

Answer:

Within the Enemy class, we have a static function called create. This function randomly chooses the enemy race to be spawned. It creates the enemy object in the heap and then returns its address accordingly at a given location that is passed into the function and then the constructor. Since the specification of the game asks for no more than 20 enemies being generated (not counting dragons). Dragons are generated differently where they are affected by whether or not a dragonhoard has been spawned in the floor.

It is not different to creating a new Player object in terms that the they call the Character constructor as well. However they are different in a way where in one case, the user chooses which player object to be created whereas in the other case, its chosen in random order.

Question 3:

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:

We use similar techniques as for the player character races. Functions such as attack, attackedBy, and killedBy. These functions are either pure or partial virtual functions which are implemented within the subclass methods GivesGold, DropsGold or the races classes. Since

there are multiple cases where a particular enemy race and player character race have special behaviours depending on each other's types, we will use the Visitor Design Pattern. For example, when an orc attacks a goblin vs when it attacks any other enemy, we make separate functions, one for specially goblin type and the other for all the other types of Enemies.

Question 4:
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.
Answer:
Advantages of the Decorator pattern is to wrapper type design pattern where a basic object can be wrapped around another class, where the two object classes share the common base class. This allows to alters and modifies the original object and do appropriate tasks in run time. Disadvantage of this pattern in this case is that potion objects do not need to be wrapped around more potion objects. This is because potions make a one time change in the HP, Atk or Def values of the player. So, the decorator pattern would not be necessary because every time a potion effects the PC, the potion's effect function just needs to be called to alter the fields of the player appropriately.
Advantages of the Strategy pattern is that implements a simple encapsulation. This is most advantageous for this problem for potions because each potion has a different behaviour. So, when a player consumes any unknown or known potion, the potion base class having a pure virtual behaviour and the concrete subclasses having implemented the methods, each potion can perform its appropriate behaviour to the player class individually.

The decorator pattern would have been useful if a potion could alter the behaviour of another potion, but since the potions of independent behaviours, the strategy pattern is a better option.

In the program, we have done something similar to Strategy Pattern however changed it a little bit to use the Visitor Design Pattern to take account of the special abilities of the Drow class because it has 1.5 times effect from a potion as compared to any other player.

Question 5:
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:
Potions and Gold piles are generated based on the probabilities given. Each of these classes have a create function that creates the appropriate type of potion or gold pile and then spawns them in a random cell in a random chamber in a floor. They are subclasses of the Item class and so they share some of the attributes and methods. However, due to difference in the nature between the two types of items, they override functions such as affect within the respective classes.