CS 246 Assignment 5: Due Date 1

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?

In our system, we have a derived class, Character, that has two derived classes: Hero and Enemy. The Hero derived class has five derived classes: Shade, Drow, Vampire, Troll, and Goblin. The Enemy derived class has seven derived classes: Human, Dwarf, Elf, Orc, Merchant, Dragon, and Halfling. This design yields a simple implementation of each race generation. Since each race is a derived class of the derived class, Character, these classes inherit the methods from Character. Adding additional races would require implementations of (pure) virtual methods that are in the other race classes. That said, this helps with abstraction so that we do not have to have to check the type of race when using a method. For example, in our strikes method, we alter the Hero/Enemy health points, however, the implementation of these pure virtual methods may be different depending on the combatants.

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

Previously mentioned above, our Character class has two derived classes. The Enemy derived class has seven derived classes: Human, Dwarf, Elf, Orcs, Merchant, Dragon, and Halfling. At the moment, two virtual methods that need to be overridden are our move and strikes methods. The move method will move the enemy accordingly i.e. a dragon will not move whereas other enemies randomly move within their chamber. The strikes method will strike the hero according to specifications i.e. a dwarf causes a vampire to lose 5 HP. Although there are also similarities when generating a Hero or Enemy including the fields health points (HP), attack (Atk), and defense (Def).

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

Our current plan of attack is to have a virtual method, strikes, in the Enemy class and each of its derived classes that will take a Hero as an argument. If the Enemy is of type Human and the Human is demised in combat, the Hero object's gold field will increase by two. If the Enemy is of type Dwarf and the Hero is of type Vampire, the Vampire's HP will decrease by five. If the Hero object is of type Elf, the strikes method will be called twice. If the Enemy is of type Orc and the Hero is of type Goblin, the damage done to Goblin will be 50% greater. If the Enemy is of type Merchant and the isMerchantHostile field is true then the Merchant will strike the Hero otherwise nothing. If the Enemy is of type Dragon it will strike as per the formula, but strikes will also be called if the Hero encounters gold. Lastly, if the enemy is of type Halfling then it has a 50% chance to cause the hero to miss in combat. Essentially, these various abilities are considered dependent on the combatants i.e. in conditionals.

The Decorator and Strategy 8 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.

The Strategy 8 pattern would be a wise choice considering we want to define a class, Potions, that will have one or two behaviours - in our case positive or negative effects that are like other behaviours in a list. These behaviours could be permanent effects and temporary effects which could be used dynamically. For example, when the Drow uses a potion, the effects are multiplied by a factor of 1.5. For bonus implementation, we could have this happen to other Heros. Implementing the effects of potions using the Strategy pattern would help with encapsulation and decoupling and reduce a long list of conditionals needed for the different combatants being considered. Further, changes to one class would not need to be implemented in other classes thus saving time. On the other hand, there are five derived classes which would require an increased number of objects / classes. If we decide on the Decorator pattern, modifying an object dynamically would be very easy and manageable. Furthermore, since we have five derived Potion classes using the Decorator pattern would be more flexible than inheritance. The Decorator is also a good choice since we need the capabilities of inheritance with the subclasses, but at the same time might need to add functionality at run time. Two important factors we also must take into consideration is the fact that the Decorator pattern simplifies our code because the implementations are done using many simple classes so the structure can be copied and pasted. One last important factor is that to include new functionality we won't have to rewrite old code but can extend with new code. Carefully considering the pros and cons of each pattern, we believe that implementing the Potions class with the Decorator pattern will be more wise. This is mostly due to the fact we want to build off the basic functionality of the potions and build new features.

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?

In our system, we have a derived class, Item, that has two derived classes: Gold and Potion. Potion has six derived classes: RH, BA, BD, PH, WA, WD. In the Gold class is an integer field, worth, corresponding to the value of hoard. To implement the spawn method, we will use virtual methods since both Gold and Potions need to be randomly placed on the floor.

In addition, your plan of attack must include a breakdown of the project, indicating what you plan to do first, what will come next, and so on. Include estimated completion dates, and which partner will be responsible for which parts of the project. You should try to stick to your plan, but you will not be graded by the degree to which you stick to it. Your initial plan should be realistic, and you will be expected to explain why you had to deviate from your plan (if you did).

To complete a problem with many working parts it is necessary that each team member has a concrete understanding of the problem; consequently, each team member will need to know the proper implementation of certain concepts. That said as a group, we decided to have a base class, Square, representing every possible character a square can take. Further, Square will have three derived classes: Character, Item, and Cell. The Character class will have two derived classes: Hero and Enemy. Similarly, the Item class will have two derived classes: Gold and Potion. Lastly, the Cell class will have five derived classes: Wall, Floor, Doorway, Stairway, Passage. To get the project started, Patrick will set up a gitlab repository and as a group will discuss implementations of classes and methods. Workload will be delegated accordingly: Patrick, will take care of the main file and controller class implementation and aid Nicholas with the combat method; Nicholas, will oversee the Character class and its derived classes; and Vuk, will implement the Item class and the Cell class and their respective derived classes. Of course, some aspects will require more attention such as the combat method. To combat this, no pun intended, we hope to accomplish this via pair programming. We believe this not only helps each team member be involved in the problem solving and implementation stages, but so each team member can learn the mindset of their team member.