PLAN OF ATTACK

ChamberCrawler3000

Tianli Zhan (t4zhan)

Angad Singh (a289sing)

Lizhong Bi (l4bi)

After reaching a consensus, we think that there will be two main components inside of the entire game, the object, and the floor. The object consists of different characters, which includes player character and also enemy characters, and items, which includes potions and treasures. The floor consists of the basic overview and the placement of each objects within the game. The floor acts as an observer, observing and change its states when the subject, in this case the object, changes. We have also made player character and enemy both observer and subject. When one attacks the other, the attacker will act as a subject and the defendant will act as an observer and vise versa. We think it is good design for us to use the decorator design pattern for a character with the basic player character and its decorators being the different kinds of potions that the player is able to pick up. This structure that we are trying to implement our program in should provide us the most optimal interaction between the different classes within our game.

Division of Work

We have decided to work on the implementation of floor class, textdisplay class and also the main function together, since we think that it is too much for one person to do and it is definitely the hardest component within the implementation. After we get the framework of the game working, we will split up the implementation of the rest of the game as follows:

1. Angad will work on creating the items class including the potions and gold subclasses.
2. Tianli will work on creating the Enemy class including the different types of enemies subclasses.
3. Lizhong will work on creating the Player class including the different types of player character subclasses.

Breakdown & Completion Dates

1. We will first work on main, TextDisplay class and Floor class, with the general character class, implementing the observer pattern. We will work on all the character generation location generation methods. We do this as a group because then we can all have a working demo. (Wednesday)
2. We will work on our own to add all the interaction methods such as motion and attack so the game works with general characters and enemies (Thursday).
3. We will implement specific character, enemies, and potions so that the game works (Friday).
4. Rigorous testing and fix bugs, and have bugless working code (Sunday).

Question: 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?

We have decided to use inheritance to create different races since all the races, be it player characters or enemies have the basic fundamental structures and behaviors. All of the distinct subclasses inherits the pure virtual methods and all the private fields from the Object class. Each subclass contains the statistics and also the effects for its specific race. Having this sort of template allows us to add new races freely and easily. When a new race needs to be added, all we have to do is to make sure to include the new race inside the structure that we have, which is to make the new race class extend from its superclass Enemy if it's an enemy race and Player if it's a possible character player race. And because of dynamic dispatch, the overridden methods inside the race class will be used, thus the different effects of the race will be applied when attacking/being attacked.

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

For the player character, it is generated based on the liking of the client that plays the game. Thereby saying that player characters are generated based on the command given by the player if it is specified by command, or else Shade is automatically generated. However, enemies are generated differently. As given in the CC3K rules PDF, there are specific chances given when generating the enemies. So when we are generating the enemy, we will have to apply the given possibilities into the random generation of our enemy. As to randomly generating the enemy. We will probably use a set of numbers to reflect on each enemy and the possibilities of each enemy based on the probability given. And we will select a random number within the set, and whichever enemy the specific number reflects to will be generated at a random possible location on the level the player is currently in.

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.

Since we have a super class that provides the virtual attack method that all the subclasses, distinct monsters and player character, will override, we now have the power to add different features to each monster's attack based on the description given in the game manual provided. This works because of dynamic dispatch, when we are calling the method, we are calling from an object, which are one of the distinct monsters, and when the program runs, it will first check the type of the object that calls the method and because of dynamic dispatch, the program will find the appropriate method to call in the subclasses. And without loss of generality, we are dealing with the player character races the same way as well.

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.

We chose Decorator pattern over Strategy pattern because we wanted to add the extra points (attack/defense) as a new feature of the character for that particular floor rather than changing the player character itself. We are aiming to add to the existing attack and defense capabilities instead of changing it's implementation at runtime. This is also useful since we have to reverse the effects of the potion once we have progressed to a new level (except for RH and PH potions).

Decorator Pattern will let us add existing functionality to our basic player character and we would be able to remove these effects when we choose to do so. This wouldn't have been the case if we were using Strategy Pattern since it would've changed the implementation of using potions instead of *decorating* our basic player character.

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?

Take note that we have created an item superclass which the gold and potion classes extend from. This provides us with the same basic structure for both kinds of objects. We will have spawnPotion and spawnTreasure methods in the Floor class, and the Floor class will also contain a private helper function "spawn" that both of these calls. Spawn will handle where the object’s location will be, so since both spawnPotions and spawnTreasures randomly spawn the location, both of which will just call the spawn method from within to avoid duplicate code, and also to reuse as much similar code as possible.