Final Project Design

Overview:

The skeleton of our program involves inheritance, with the realization of applying Observer, Decorator, and Model-View-Controller (MVC) design pattern, which help us reduce duplication of code and demonstrate. The core module of our program is the floor class, which contains a vector of vector of cells. This is our primary “board” of the game. Then in the controller class, we have used a vector of five floor pointers that represents the dynamically changing game. The primary building block of our game is the cell, implemented in Cell class. A cell could contain either a Entity (which is either a Player or Enemy), or Item (which could be a gold, potion or suit). For player characters, we then have the different concrete races inherited from abstract superclass Player. For enemies, we have different concrete enemies (including Merchant and Dragon) inherited from Enemy class. Similarly, different items (gold and potions) are inherited from Item (note that Temporary Potion inherits from both item and the abstract decorator).

Design:

In our DDL 1, we made an ideal design of implementing observer design pattern and Decorator design pattern. Observer design pattern was considered to be used since we are playing a game on a two-dimensional grid, which suggests the possibility of implementing a grid that includes a vector of “Cells” grid just like Reversi. Since the state of each cell are changing for each player action, we need to track and display those changes simultaneously when we are playing the game. Hence, the Observer design pattern could easily handle this problem. We implement two concrete observer class: “textdisplay” & “actiondisplay” to track and display the map and actions.

Decorator design pattern was also adopted for the implementation of temporary potions in our design. The reason is that the attack boost/wound and defense boost/wound effect are temporary. Thus, we can create a decorator that inherits from Player class and Item class so that we can use the static cast method to trace the original Player Akt/Def state.

In the process of implementation of Enemy move, we made Cells to be both subject and observers. However, we have found that it is not only troublesome to do notify observers on specific type of Cells, but also it would cost more time and memory. The purpose of using Observer should be simplifying the process of implementation and make and saving running time. So we reject the idea of setting cells to be both subject and observers but only remain them to be subjects. Therefore, we decided to only implement Observer design pattern on the game displays. It is worth mentioning that we had not added the actionDisplay class until we made that decision, which is also an concrete Observer class and is part of the optimization of our code. In terms of the Decorator design pattern, we kept that part because it has indeed simplified our implementation.

While we have adopted Observer and Decorator design pattern, which greatly helps with implementing text display and adding more temporary features (e.g. Temporary potion) to our program, we have also applied the idea of inheritance throughout our entire implementation. This could demonstrate our solid understanding of Object Orientated Programming, as it is a fundamental concept of OOP. It helps us maximize the reuse of code and make our program more modular and organized. It does not only bring us a relative low coupling code, but also a highly cohesive code since every function of every module is closely to its object.

Notice that the dragon class and the subject class is a template class. In this way, it helps us easily classifies the dragon that protects dragon hoard and the dragon that protects Barrier Suit.

We have also used the Model-View-Controller (MVC) design pattern to manage interaction since we don't want to call the methods of our model directly in our main. For example, when the user inputs a direction, the main function calls Controller's member function playerMove() which invokes the Floor's member function playerMove() that modifies the program's data. In this way, we did not only make a clear separation between model, controller, and view, but also improve the maintainability, testability, and reusability of our code (and it turns out the MVC help greatly when we test the model).

Although we could prompt the user for “Invalid Input” using a big if and else statement, we have chosen to use an Exception class. By using exceptions, we have cleaner code, and the program has a better flow.

Lastly, we are very proud of our application of casting. First, it is freshly taught knowledge. But most importantly, we solved the issue of removing potion effects when the player reaches the next level. This challenge is resolved by writing a getComponent() method in our abstract Decorator class, which is the accessor to the only field of the Decorator class – component. Then in the Controller class, we recursively call this method when the player is on the stairway but has not spawned on the next level until we eventually reach the basic concrete component – A concrete Player object without any temporary potion effect.

Changes:

This section is not outlined in the project guideline. However, these changes are so critical to our final program. Also, these descriptions can help us answer the required questions, especially the discussion of how our chosen design accommodates change. Although we are satisfied with the degree of consistency of our design, couple changes were being made. We have introduced some of them in the previous paragraphs, here is the elaboration:

1. The most significant change we made is that we decided not to make a cell both a Subject and an Observer (like in the Reversi game we implemented in A4). Although adopting this design pattern is doable, it is not as effective as what we have. First, the notify function in Cell would be EXTREMELY hard to implement. This includes the notify functions in its subclasses (Player, Enemies, Items. Note that these classes no longer inherit from Cell, we will elaborate on this). The Observer design pattern intends to help the developer to code in an algorithmically smarter way. However, forced using Observer for Cell increases the code complexity dramatically, and makes our model hard to maintain and

extend. Instead, we take the advantage of the Observer design pattern and use it for map display (Textdisplay Class) and action message display (Actiondisplay Class). Namely, the Textdisplay and Actiondisplay are two observers to every cell in the Floor.

2.We have also changed the inheritance relationship between Cell and Entity and Item. In our previous design, we choose to make a Player/Enemy/Item a cell, ie. A Player/Enemy/Item is a Cell. The initial purpose of it is to bring these characters in our Observer Design Pattern. Since we have realized our initial design would take a huge amount of effort to implement and debug, we have chosen to modify this relationship. Now a Cell and a Player/Enemy/Item follows an aggregation relationship, which means a Cell class has private pointers to each of the Entity/Item classes. Intuitively, instead of saying "A Player is a Cell, when a Player moves, the Cell moves to the intended place of the Floor", we now described it as "A Cell contains a player, when a Player moves, it moves to the intended cell of the Floor." In this way, we have decreased the coupling in our program by a significant amount while being highly cohesive. In our current design, a change in Cell class would not impact Player/Enemy/Item, and everything of these classes are still closely related and they all serve a common purpose – define the behaviors of a character.

3.We have used a Controller class, which will manage interaction to modify data. Namely, the main will only ask for user input and invoke Controller's methods. Then the controller will use the functions in the model. Using MVC design pattern, it increases the program's flexibility, code reusability, and maintainability. Also using MVC, we have made our code with low **coupling**. We separate the responsibility of data management (Model), user interface (View), and interaction management (Control). And MVC also promotes **cohesion**, since everything in each of the M, V, C has a common purpose respectively, and they are closely related to each other.

Resilience to Change

We will discuss how our design is resilient to change by our techniques:

1.Decorator design pattern: the Decorator design pattern is resilient to change. If we wish to add more items with temporary effects, the items could be simply added by making them children classes of the abstract

Decorator. More generally, if we add anything that could change Player's stats (either permanently or temporarily), we could add them to the Decorator.

2.Observer design pattern: Adopting this Observer design pattern also improves our resilience to change. This is especially important when we want to change the display, which is essential to our program as this is the only thing the user sees. Instead of changing display by browsing all different classes, or go through each line of the Floor class (if not using Observer Design Pattern), we can now simply go to TextDisplay and make modification. Observer also helps minimize coupling since we separate responsibilities between display and the model. The implementation of them does not impact the other. The above example could demonstrate the low coupling.

3.Template Class: Perhaps we want to create more dragon-protected items, then the template dragon class can improve code reuse and its easier to generate. This makes the dragon generation resilience to change.

4.Model-View-Controller design pattern: The MVC is designed to be resilient to changes, since it separates responsibility of data management, interaction management, and user interface. This lowly coupling design brings flexibility to our program.

5.Exception class: If there are changes, we could add different exceptions so it is easier to maintain, and detect possible error for the changes to our code

Answers to Questions:

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

Ans: We decide to write a Player class that is the abstract superclass of the different concrete race classes (e.g. Human class, Dwarf class, etc.). And the Player class is also a subclass of an abstract class Entity. To generate a race, we could simply call the constructor of that race class. It will not be difficult to add additional classes (races). We could add these classes as subclasses of the abstract Player superclass.

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

Ans: The way of generating different enemies will be similar to generating player character. We will implement an Enemy class that is parallel to Player class (i.e. an abstract subclass of the class Entity, but a superclass of the concrete enemy classes). We then implement concrete enemy classes (e.g. Werewolf class, Merchant class, etc.). To generate an enemy, we could simply call the constructor of that enemy class, which is same as the way of generating a race. The system handles the generating of player character and enemy in a similar way since they are very alike. They both have HP, Atk, and Def. However, player character has gold and moves by the user command, whereas enemies does not "carry" gold and they move conditionally/randomly.

3. *How could you implement special abilities for different enemies. For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?*

Ans: Because we have a controller that can take arguments from the command line, so first we can make an if-control flow, and implement different methods on those enemy classes so that those enemies will be born with special ability when we choose to apply DLC actions.

4. What design pattern could you use to model the effects of temporary potions (Wound/Boost Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?

Ans: A decorator design pattern could be used to model the effects of temporary potions, since we could add temporary effects by creating new subclass objects. For example, if we want to have a dwarf that first uses a WA potion and then a BD potion, then we could create it by Player *dwarf = new BD{… new WA{… new Dwarf{…}}};

5. How could you generate items so that the generation of Treasure, Potions, and major items reuses as much code as possible? That is for example, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code? How *could you reuse the code used to protect both dragon hoards and the Barrier Suit?*

Ans: When initializing the floor in main.cc, we have a vector of all cells that are available for positioning a treasure/potion/major item. By using shuffle function using a certain seed. We can generate a shuffled vector of position. By assigning certain those postions to generate those items, we can ensure that their generation is random and unlapped. Also we can randomly generate the player position. We could write a Dragon template with a parameter corresponding to the type of item protected by the Dragon. It will look like this: template class Dragon { … T protected_item; … }; When generating a dragon that protects Barrier Suit, we could do: Dragon d; // assuming Suit is the class name for Barrier Suit

Answer to Final questions

1. I should not only focus on my own codeworks, but also have a thorough check of the implementation of others and listen carefully; try to understand their opinions of their own designs, so that we can speedily debug as well as properly combine every part of the project together.

2. I would first have a clear and read all the questions carefully from begin to end. That is because our team has found many missed problems in the project description on the half way of the coding or merging, so that we had to consider the whole design again.