Introduction (if needed)

Overview (describe the overall structure of your project)

Updated UML

Design (describe the specific techniques you used to solve the various design challenges in the project)

Resilience to Change (describe how your design supports the possiblity of various changes to the program specification)

Answers to Questions (the ones in your project specification)

Final Questions (the last two questions in this document).

Conclusion (if needed)

Overview of the structure and explanation of classes

1. We use observer pattern as the overall structure of our project, and adopted decorator pattern for the convenience of displaying and item using.
2. First, there are two base classes called Subject and Observer. The Subject class stores two structures: State and Info. State contains information of action and direction. Info contains the symbol of the Subject, information of location, battle information like atk and def, and the gold carried by this Subject.
3. The Cell class is the base class for most of the objects in our code, and it inherits from both Subject class and Observer class. Apart from Cell, Floor is also an Observer, which contains a Display object to print out the game.
4. We write two subclasses of Cell called BasicCell and CellDecorator. The CellDecorator class represents the map elements that will not move or disappear like the walls and doors, while the BasicCell class represents the elements that may move or interact with other elements like the player or dragons. The CellDecorator contains a pointer to Cell.
5. There are two subclasses of BasicCell, one is Pickup and the other is Animals, where the Pickup stands for all item objects and Animals represents all attackable objects.
6. The Pickup stores a Mod structure, which contains four int values as modifiers of atk, def, hp and gold to represent the effect of items applied at player. Pickup has four subclasses: Potions, Barrier, Treasure and Compass. They differ in their notify() method.
7. The Animal class contains four subclasses: Dragon, Player, Merchant, and Enemy.
8. The subclasses of Player represents different races, and the subclasses of Enemy represents different monsters.

UUUUUUUUUUMMMMMMMMMMMMMMMMMLLLLLLLLLLLLL

Design:

1. We adopt observer pattern for this project because there are a lot of reactions of other objects when the user is operating one single object. This design pattern lowers coupling since it may requires more interactions between modules otherwise.
2. To make it possible for implementing more advanced items(like traps or hostile cannon towers), we make Info structure contain battle information instead of containing only location and symbol and leave the battle part to Animal class.
3. For State.action and State.direction, we add one more action called ‘s’ and one more direction called “P”, where ‘s’ stands for “the object is not moving” and “P” stands for “the object is not acting or its action involves all nearby units”. This could help us reduce bugs and makes the State structure easier to understand. A public method called reset\_state() is implemented to set the object’s state back to ‘s’ and “P”. At first we wanted to combine notifyobservers() with reset\_state() to enhance cohesion, but then realized that not all objects should be reset to normal state (e.g. a dead enemy, a dragon pursuing the thief who tried to steal its treasure). A public method called set\_state() is provided for the Floor to execute commands.
4. We use Decorator pattern for two main reasons: display and potion effect. With Decorator pattern, we can always use the symbol for the innermost cell when displaying, and make sure the original cell at that position is not changed after new objects occupy and leave, which means that we don’t need to track what’s under a Player’s(or other objects) feet in Player object or Floor object. This significantly lowers coupling.
5. To apply the potion effects, we let the floor set the state.action of Player as ‘u’ and give it a direction, then call its notifyobservers(). When a potion is notified by such a Player, it will set its own state.action as ‘u’(using set\_state) , reversing direction and call notifyobservers(). When Player is notified by a Potion in right direction and ‘u’ action, it passes the Potion’s Mod structure to Player.get\_index method, which automatically converts the potion effect in to the real effect onto the Player and returns the real effect as a Mod. After that, the Player stores the atk and def effects into its own Mod structure called potions and applies the hp and gold effect directly onto itself. When a Player calls get\_info(), the returned Info would be a combination of Player’s own Info and the effects stored in Mod potions. When all these procedures are done, the floor object will remove the Potion from its position and fill the position with the original BasicCell contained by Potion and exposed by a method called cell\_detach().

When a new floor is entered, the Floor only needs to call Player.upstairs() to reset the temporary effects of potions by reseting the Mod structure inside the Player object.

This helps to reduce coupling since, in this process of applying and removing potion effects, only several public methods with no parameters are called.

1. To make the dragon guard its treasure(or barrier suit), whenever a guarded item is notified, it will call notifyobservers to notify the dragon. Then the dragon attacks in all directions(IT IS A DRRAAAGOOON), but only the Player will get involved by its attack. In this way, we avoid storing a treasure pointer in a Dragon object, which lowers coupling.
2. Same as enemies like vampires, merchants will also automatically attack player when player is beside them, but only those players who once killed a merchant will really get attacked. A Boolean value called wanted\_by\_merchants is stored in Player object as a private member.
3. The notify() method for Player is implemented within Player class instead of in different race classes to minimize coupling and reuse code. We provided a method called special() for race classes if the user wants to add more features for different races. Special() will be called at the beginning of notify() and will return a bool to determine whether or not will the original notify() still be executed. This helps separate race features with Player basics and makes the code much more easy to modify. Similar approach is also used in Enemy class.

To add a new race, the user only need to give the basic information in constructor, and write its own get\_index() and special() if necessary.

To add a new enemy, the user only need to give the basic information in constructor, and write its own enemy\_feature()(similar with special()) if necessary.

Resilience to Change

1. To change the ability of a race, we only need to modify the parameter its constructor passes to Player’s constructor for basic values like starting hp, and modify the feature() method to implement special abilities. Then modify the get\_index() to change its interaction with potion effects. All of these three modifications are in one single .cc file and nothing else is needed.
2. To change the effect of a potion, we only need to change the Mod parameter passed to potion’s constructor. The program already supports all kinds of positive/negative atk/def/hp/gold potions.
3. To add more basic map elements like grass, floor or sky, user only need to pass different char values into BasicCell constructor. (i.e. simply draw it out in the map file without changing code)
4. To implement vision for player, we only need to add one more subclass of CellDecorator called WarFog, which covers all cells and contains a private member called Bool visible. Bool visible is true only when WarFog is notified by the Player, and only when Bool visible is true, get\_info method can return the symbol of the cell beneath it.
5. Since notify() methods are written in Player and Enemy class instead of their derived classes, it would be much simpler to change the interactions between objects.

Answer to Questions

1. //
2. Write the functionalities into enemy\_feature() and use returned Boolean to terminate the notify() if necessary..
3. Decorator pattern.
4. //

Answer to Final Questions

1. It is really important to keep a good version control and keep track of all commits and merges using comments.

The amount of work required by different parts should be determined first so that everyone would finish at roughly the same time, which allows us to start testing earlier.

1. We would read the assignment document more carefully and take notes for those important information so that we won’t need to rewrite some part of our code again and again because of neglecting some instructions.

The first version of design would be built with more caution, and we should make sure it works before starting coding. Several designs should be proposed first, then the best will be picked.

Before coding, everyone should get a table of the methods he or she is allowed to use and a table of methods he or she should provide.