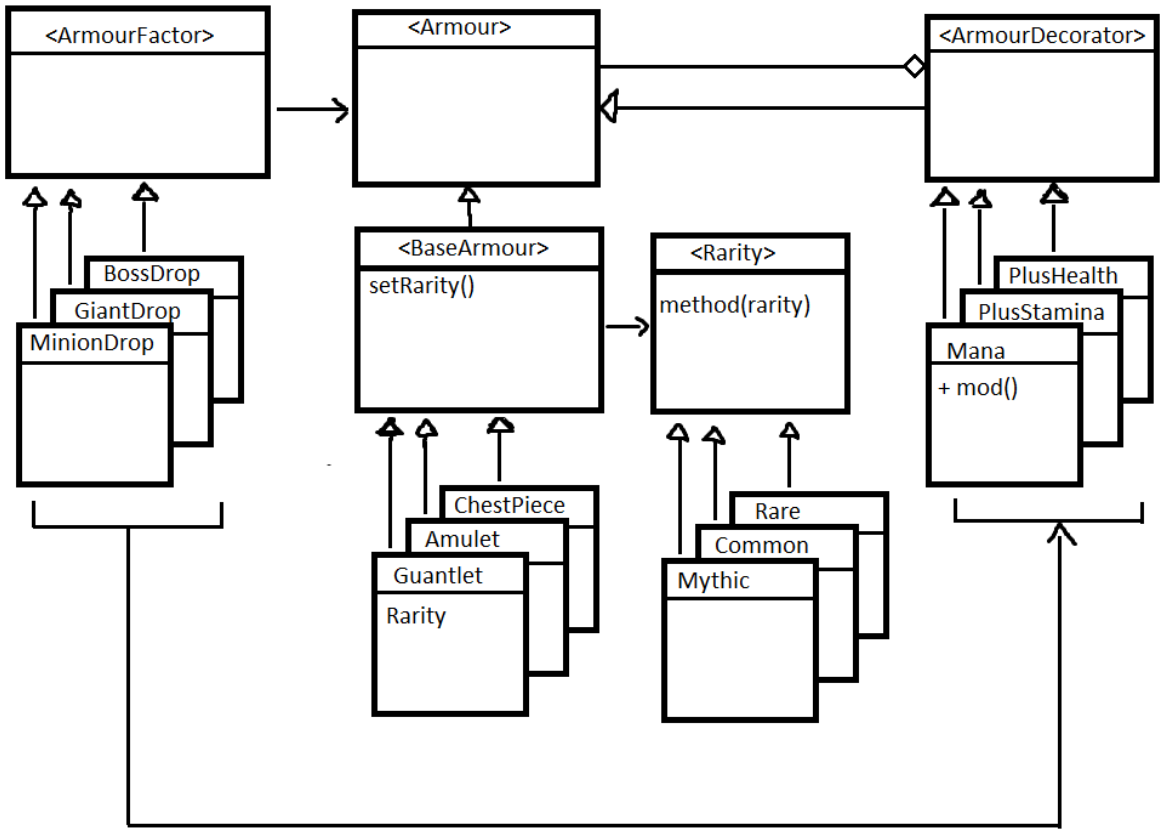
2AA4 Assignment 3

Kai Arseneau 400319194 arseneak

Travis Wing 400308161 wingt

Question 1

UML Diagram



Design Patterns

Factory

We used the factory pattern for the armour generation aspect of the program. We thought that the quality of the dropped armour would stem from the level of the boss that was killed. Therefore, we created an armour factory interface and then created 3 different levels of enemy

drops (minion, elite, boss). For example, a minion can only drop a helm, ring or amulet and the rarity of these armour pieces can only be common, uncommon or rare. These factories also use the modifiers since the level of rarity determines how many modifiers can be added to any piece of armour. So overall the different factories produce a piece of armour with a rarity and a set number of modifications that can be added to it all based on the level of factory.

Decorator

We used the decorator pattern for the modification aspect of the program. The idea behind this part was that we would have a piece of armour with a rarity on it, then we wanted to be able to decorate it with different modifications. This like the example from class where we had the burrito where we wanted to be able to add toppings and then also be able to add additional toppings to our topping list. The use of decorator allows us to apply modifications and then also add to our modification list without touching the rest of the program.

Strategy

We used the strategy pattern to be able to add a rarity to a piece of armour. Since a piece of armour can only have one rarity type, we used the strategy piece to avoid having an armour get two rarities. We applied this pattern to basically make sure that pieces of armour don't have access to it more then once.

Design Principles

Our design uses the single responsibility principle in all different ways. Basically, every class in our program has one job and then passes it to the next class. For example, we have a class for picking an armour piece and a class for picking rarity, then we have a class adding modifiers. We do not make classes perform more then one function except for our factories since they must do a quite a few different things.

Our design uses the open for extension but closed for modification design principle. This is implemented in the modification aspect of the design. You can add as many modifications as you want without effecting any other part of the code. Since the modifiers are decorators for the armour they take in an existing armour and modify it, therefore since it doesn't deal with anything else you can add as many as you want without having to change other classes.

We use Liskov's substitution principle in our program. A good example of how we used this is that we have a class called Base armour which is implemented by many different types of armour. If you replace this class with one specific armour piece, helmet for example, the program will function the exact same way except the armour piece will always be a helmet. This fulfills this design principle that the child class of a parent can replace the parent and not cause problems.

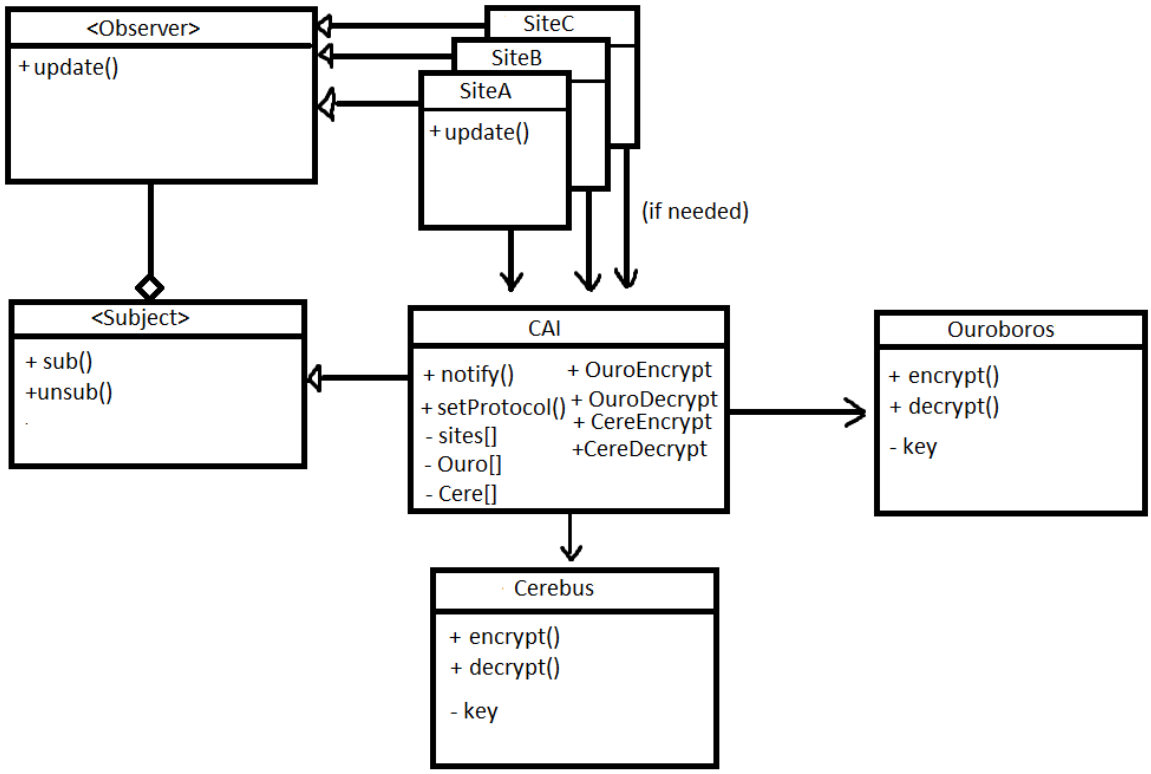
Design Explanation

The way our design will function with the overall application is that different factories (enemy types) can be designed based on the drop rates and rarities you'd like, which can then generate a piece of armour with a rarity (unique) and multiple modifiers based on the rarity (can have duplicate modifiers).

There are several different enemies drop factories (Minion, Elite, Boss) and they each have unique drop rates. For example, a minion can only drop a helm, ring or amulet and the rarity of these can only be common, uncommon, or rare. Then the way that the modifiers work is that depending on the rarity of the armour piece and where the armour was dropped it’ll add those corresponding modifiers. For example, an uncommon piece of armour dropped from a minion can have one modifier and it’ll either be intelligence or dexterity. Therefore, the different level of enemy that you kill drops better loot with higher rarity and more modifiers.

Question 2

UML Diagram



Design Patterns

Observer

In our design we used the observer design pattern. We used this because the question wanted us to have the CAI be able to tell the websites to update all at once and this is a classic use of the observer pattern. We created an observer that has sites observing the observer, these sites get updates from the CAI telling them what encryption type and protocol key to use. The CAI also has the option to add or remove sites from the list.

That is the only real design pattern that we implemented, the Cerberus and Ouroboros classes are only outside of the CAI for simplicity reasons. We figured that it would be much more complicated if both of those were inside the CAI.

Design Principles

Our design uses the Single Responsibility principle in two different cases. We separated the Cerberus and Ouroboros classes to be separate from the CAI because we wanted the simplify the CAI and give these classes one singular responsibility which is to encrypt and decrypt the messages using their respective algorithms.

Our design uses the open for extension but closed for modification principle in two different ways. Firstly, we can add and remove sites to the CAI's site list without having to change the functionality of the CAI or the sites themselves. Basically, we subscribe or unsubscribe the sites from the CAI's messages. Secondly, we can add as many new protocol keys as we want without changing anything else. All you do is implement a new set of numbers for Ouroboros or a new number for Cerberus and that doesn't involve the changing of anything else, it just adds a new protocol key that CAI can randomly select from.

Our design uses the interface segregation principle. We use this principle by having a subject interface and an observer interface. By separating these interfaces away from each other we simplify both. Instead of having one interface that can add or remove sites and be able to notify them we have an interface that can add and remove and another interface that can only update.

Design Explanation

The way our design functions with the overall application is that the CAI handles the decision-making process for the keys of which the sites use for security and the sites would have access to encrypting and decrypting messages. The CAI will decide what kind of encryption type to use and which protocol key to send with that type, this will get sent to the websites, so they know how to update their security. Then the websites have access to the CAI's send and receive message functions so they can follow they can make implement the new security that has been given to them (this is outside the scope of the assignment, but we are including it in the description since the sites could use those functions).

As well the CAI can add or remove sites from the site list, and it has access to a whole roster of either Cerberus or Ouroboros protocol keys that it randomly selects from to send out to the sites depending on the encryption type chosen. In addition, the CAI can also add new protocol keys for Cerberus and Ouroboros and add them to the roster list. You could have the protocol

keys be stored on the website and just have the CAI tell them which one to use but this would be less secure. Therefore, we decided that the CAI will handle the updating storing of the protocol key and then all it must do is send out an individual one each time.