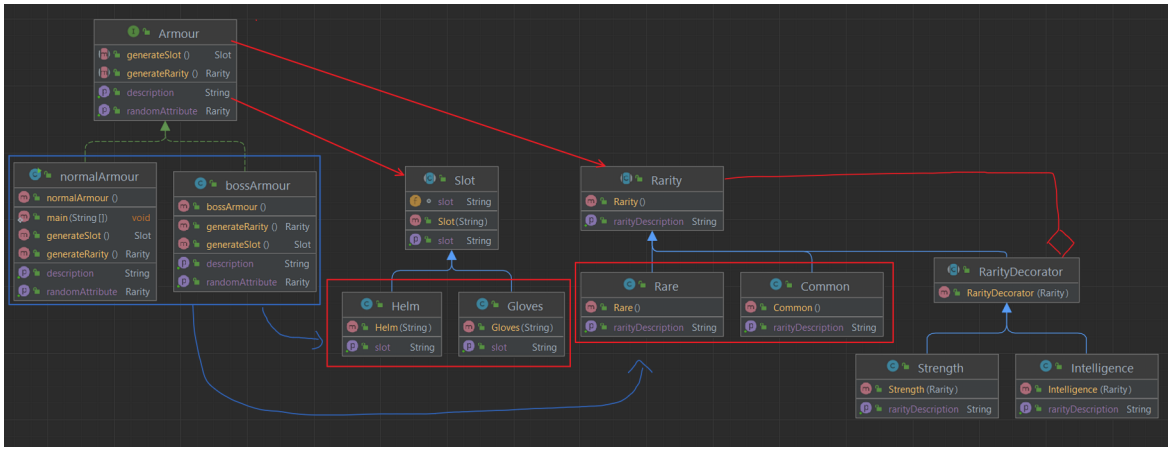
Partner: Housam Alamour, alamourh, 400317089
Omar Muhammad, muhammao, 400325041
SFWRENG 2AA4
April 9, 2022

Assignment 3

Question 1. (Armour)

1. A UML class diagram of your refactored code



2. A list of design patterns you applied and why you applied them

   The design pattern our program follows are the factory pattern and the decorator pattern. A Factory allows you to make related products using classes or interfaces without making a concrete class. In out code, we have *Armour* as our main factory, which produces concrete Armour classes (eg. normalArmour, bossArmour). **The reason for this change** was to allow for the control of how armour is generated. For example, if you want the armour dropped from a boss to be of higher quality, you can simply edit the bossArmour class or if you want to add a different type of enemy, you can make a differentArmour class that implements *Armour*.

   Then we have 2 products, being slot and *rarity*. These two products are abstracts classes and they produce concrete products being $typesofslots$ (eg. Helm, gloves), and $typesofrarity$ (eg. Rare, Common). These products have methods that are then used by the concrete armours to give a value to the slot and rarity of a particular concrete armour instance. **The reason for this change** is so that if new bonuses or new armour types were to be added such as resistance or health, they can easily be added by extending the slot or rarity classes.

   Finally, we implemented the decorator method for the *Rarity* class. After this we made the modifications/attributes extend the rarity decorator, and thus they extend the *Rarity* class. **The reason for this change** is this allows us easily add more modifications later on in the development of this code by simply extending the rarity decorator. This also allows us to give new behaviour or mods to the armour at runtime, so when generating the armour we can give it different mods. The decorator patter also has the benefit of making it very easy to print all the modifications a piece of armour has. The rarity is "wrapped up like a burrito" with mods and so when we run the get call the get description methods, it recursively gets all the mods and finally finishes with the rarity.

3. A list of design principles you feel your design enforces, and a justification as to why it does

Single responsibility principle: our program follows this design principle as each of our classes is specialized in only one of our topics, for example in our modification folder there is 'intelligence' and 'strength' classes, each of these classes has the sole purpose to give the armour a modification of intelligence or strength. In addition the only reason why we would ever want to change these classes are if something with the intelligence or strength modification changed (in game update for example) this design principle is seen all throughout our code and is why our code implements it (each class has a sole purpose/a single responsibility). This makes our code easier to understand and maintain later on down the lines, as if you want to make a simple change to some class, you can be rest assured that you only have to make that change in that class and the code will still work without having to find where else to make that change.

Open Close Principle: Our program uses a factory to generate the armour which is a direct connection to this principle, a factory is incredibly easy to extend by adding more classes that implement them, however to actually modify the factory this cannot be done by the client at runtime, only if the code is reworked. Therefore by definition of a factory and the open-closed principle our program follows this (easy to extend factory but closed to modify the factory itself). This makes our code open for extension and closed for modification. If new slots, rarities, or mods want to be added they an easily be added without breaking the related abstract classes.

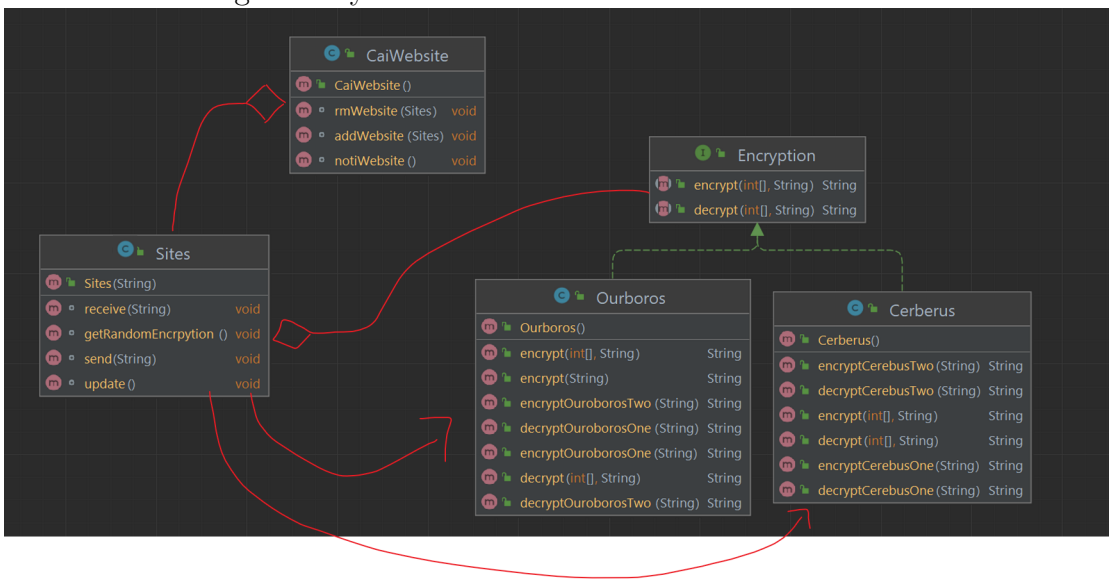
Interface Segregation Principle: I know this because a key concept of this design is that all the interfaces are used and all the methods will be used for any instance of it. To break this down further we can examine our code where we will see our interface has four main methods generateSlot, generateRarity, getRandomAttribute and getDescription, all of these methods are essential for our application to run properly and all are used anytime an armour is made (cannot have an armour without any of these methods) and since this is our only interface, this implies that our program abides by this principle. This makes our code more compact and easier to maintain and understand to make it easier for programmers to work on this later on down the line.

4. An explanation of how your design is intended to be used in the overall application.

An example of the code running can be seen in the *Runner* class. To use this code, you simply create an instance of *Armour* and then assign it to be one of the armour types (eg. normalArmour). Inside the normalArmour class, the constructor calls methods to give the armour a slot and rarity. Then according to the assigned rarity, it will give the armour either one or two random attributes. Thus with the simple creation of the armour, all the other actions such as assigning slots are dealt with, and now the armour is ready to be used by the getDescription method. This is much better than the old code which had to do all the heavy lifting to assign attributes was done in the armour class, making it hard to maintain, modify, extend and understand. To add more modifications, you simply extend the rarity decorator class with what you want. Then this mod can be applied to armour by modifying the getRandomAttribute method in normalArmour to allow this new attribute to be applied. new slots and rarites can be added similarly by extending their corresponding classes and generateSlot and generateRarity methods. Finally, new Armour types that are dropped by different enemies can be added by implementing the *Armour* class and assigning the appropriate RNG drop values for all the methods.

Question 2. (CAI/EncryptionProtocols)

1. A UML class diagram of your refactored code



2. A list of design patterns you applied and why you applied them

   In this program we applied the observer design pattern and the strategy pattern. The strategy pattern is a pattern that allows us to define a family of algorithms, in this project we used it to define the encrypt and de-crypt methods for any of the classes needed (more could be added), the family that this would fall under would be the Encryption family where you can freely encrypt and de-crpyt using this interface. The benefits of using the strategy pattern are that it makes it easy to define the a class that does something specific and lets us do it in a lot of different ways, essentially by using this pattern we made the program much easier to extend (more encryption methods could be added fairly easily) and is why we decided to implement this pattern. The observer pattern basically allows for the code to control a bunch of objects in this case our ”sites” were being controlled by this pattern, in our code we had the sites being controlled and stored into an array-list in a separate class, this allowed us to perform operations such as update() fairly easily since all of our code was being ’watched’ by the observer pattern. The benefits of the pattern are as stated before, it allows for our program to maintain multiple objects that were created and makes it easier to update, remove, add or perform operations of the object. These two design patterns allowed us to create a well polished program.
3. A list of design principles you feel your design enforces, and a justification as to why it does

   We believe our program abides by the Interface Segregation principle and the open close principle, the Interface Segregation principle is obviously followed since in our interface for encryption the two methods can always be used for the program, there is no need to create a separate interface since our only methods are used for the application to run as intended, this implies that our program abides by this principle since this is our only interface and all the methods of the interface can be used for each instance.

   For the open closed principle, our program also abides by this principle because it is easy to extend upon while also being harder to modify, the code is easily extendable because it would be easy to add another encryption class that will implement the interface, this new class will have its own algorithm to use and with minor changes to the code the program will run smoothly, the interface and classes itself are not as easy to modify as there are many dependencies and numbers that may cause an error (type error, index out of bounds, etc) that will cause a large rework of the application in order to successfully implement a change to the class. So this implies that our code follows the

open closed principle since it is easy to extend (add more encryption methods) but not as easy to change the actual classes.

4. An explanation of how your design is intended to be used in the overall application.

   This application is intended to be used for websites or places to send and receive codes that could be then understood by receiving the message and translating it, so to actually use our code the intention is to create an instance of the class CaiWebsite which contains an array-list to keep track of all the sites that may be created, this array-list has a class of its own that also includes functions that will be used in conjunction with the interface for encryption, so anytime an instance is created a 'Site' is added to the array-list, this site can then be used to send or receive a message by inputting a word of type String into the parameters, the sites can also all by updated with new encryption numbers by using the update function, this function with the help of the notiWebsite will iterate through the array-list to update each of the protocols of the sites, this is extremely useful to keep things random which is needed for such a program, in addition to the previous features, there is also a remove site function which can remove the site from the array-list which means the CAI will no longer be keeping track of the site, the site will no longer be updated. Lastly, to wrap all this together the program is extendable due to the design patterns we used which means that if needed more encryption protocols and types can be made that will implement our interface, these can then be used by the sites to offer more ways to encrypt a message.