Assignment 4-1

1 Analysis of Assignment

In this assignment, we need to implement the *Decorator Pattern*, which conceptually resembling a series of wrappers of a object, is in fact a kind of loose coupling method that could be used without breaking the original design of the wrapped object. This pattern also provides a potential of the runtime alteration of attributes.

2 Overall Design

The *Decorator Pattern* requires a base object and a series of decorators, which a shared method of returning the current state (*e.g* cost) should be included, and a decorator should be able to wrap another decorator or the base. In the case of Java programming language, to meet the two requirements, we extract the both to make it an interface (see Text.java), as it provides a standard getter and the accessiblity of referencing objects across classes.

Therefore, `Text` were specified as a public interface, should be inherited by a base class and an abstract decorator class (see PlainText.java and CypheringDecorator.java). The `CypheringDecorator` should cover the methods of all decorators as are back referencing and current state returning. There will be three classes in decorator, inheriting from the decorator, each of which simply makes a little change based on previous object. Upon calling of the getter, one should trace from the top back to the base.

3 Key Components and Detailed Design

Considering this ciphering task, from top to the bottom, there includes a root interface `Text`, a descendent original text class `PlainText`, followed by an abstract text-modifying decorator and several real-world ciphering function (see `ShuffleCypher`, `ReverseCypher` and `ReShuffleCypher`).

At runtime, we create an object of the plain text with “HelloWorld!” in as an attribute, (see Test.java), then declare three decorator references, create them seperately while nesting the constructor with previous objects. At the end of the `main` function, it should print some text with 3 levels of decoratings, some characters resembling “ˆ_!̂dlroWolleHˆ_̂”.

Implementing the decorative methods is actually implementing a recursive function, which will be a continuous nested function calling when it is expanded. This back tracing chain is actually a fundmental mechanism as a trigger of being able to call from the start. There are totally three layers in this recursive calling (if `PlainText` is exclued), each of which only cares about what they will add to the string referenced from their down side of the calling stack, and return a new string to it's upper function. Advantageous as the *Decorator Pattern* could be, one can add a new decorator or copy a previous decorator with ease, and it can be done at runtime, enhancing the ability of modification.

Actually, decorating at returning time leads to a problem. Whenever it answer to a calling of a getter, it will always trace from the bottom up to the calling point, which can be a burden when a middle layer decorated object is frequently requested from the user. One can prevent this with a constructing-time decorating, concreting the results in each layer, more like an iterative version of this pattern (I don't know if it's correct). Another side effects of *Decorator Pattern* is an increasing number of small classes from the abstract class (`CypheringDecorator` here).

4 Conclusion

To sum up, *Decorator Pattern* is a loose coupling methods for altering attributes of objects with ease in the realm of OOP. Implementing dynamically, it influences nothing in the wrapped part inside the calling. Any couples in the chain, if the distance is large than one in between both, they are transparent from each other.

In this report, we analysised the basic concept of *Decorator Design Pattern*, then discussed the paradigm of implementing this pattern in Java, and the actual detail in my project. Considering the positive side, we talked about the calling process of the chain, and we also mentioned some negative sides.