INSTRUCTION:
Read the description below and do the necessary.

## Case Study 1 (Part 4 - Decorator)

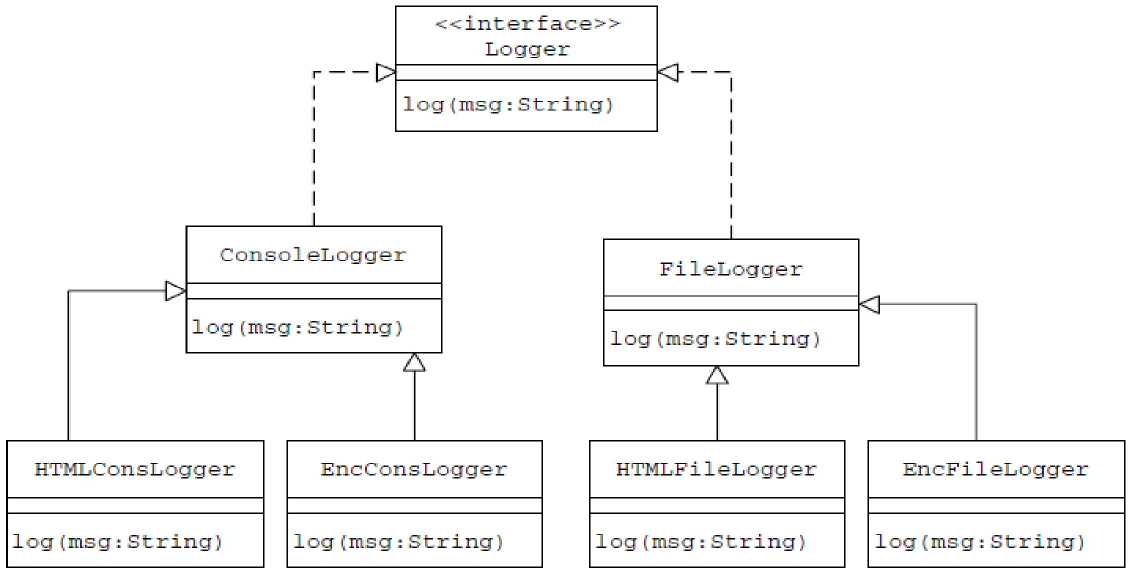
Suppose that some of the clients are now in need of logging messages in new ways beyond what is offered by the message logging utility. The clients would like to have these two small features:

- Transform an incoming message to an HTML document.
- Apply a simple encryption by transposition logic on an incoming message.

Typically, in object-oriented design, without changing the code of an existing class, new functionality can be added by applying inheritance, i.e., by subclassing an existing class and overriding its methods to add the required new functionality. Applying inheritance, we would subclass both the *FileLogger* and the *ConsoleLogger* classes to add the new functionality with the following set of new subclasses:

| *Subclass* | *Parent Class* | *Functionality* |
|---|---|---|
| `HTMLFileLogger` | `FileLogger` | Transform an incoming message to an HTML document and store it in a log file. |
| `HTMLConsLogger` | `ConsoleLogger` | Transform an incoming message to an HTML document and display it on the screen. |
| `EncFileLogger` | `FileLogger` | Apply encryption on an incoming message and store it in a log file. |
| `EncConsLogger` | `ConsoleLogger` | Apply encryption on an incoming message and display it on the screen. |

The resulting class hierarchy after applying inheritance to add the new functionality:



1. If we had additional *Logger* types (for example a *DBLogger* to log messages to a database), it would lead to more subclasses. With every new feature that needs to be added, there will be a multiplicative growth in the number of subclasses and soon we will have an exploding class hierarchy.

   Rescue the situation above by using the *Decorator* pattern. The *Decorator* pattern recommends having a wrapper around an object to extend its functionality by object composition rather than by inheritance. Draw a UML class diagram to how you apply *Decorator* pattern to add the new functionalities to the message logging utility.

2. The *HTMLLogger* overrides the default implementation of the *log* method. Inside the *log* method, this decorator transforms an incoming message to an HTML document and then sends it to the *Logger* instance it contains for logging. The *EncryptLogger* does the same thing but implements a simple encryption logic by shifting characters to the right by one position. Implement *LoggerDecorator*, *HTMLLogger and EncryptLogger.*

   Hints:
   For *HTMLLogger,* you can add the following HTML tags to the message you want to log.

   <HTML><BODY><b>yourMessage</b></BODY></HTML>

3. In order to log messages using the newly designed decorators a client object (*LoggerTest* in this case) needs to:
   - Create an appropriate *Logger* instance (*FileLogger/ConsoleLogger*) using the *LoggerFactory* factory method.
   - Create an appropriate *LoggerDecorator* instance by passing the *Logger* instance created in the previous step as an argument to its constructor.
   - Invoke methods on the *LoggerDecorator* instance as it would on the *Logger* instance.

   Add the statements in *LoggerTest* class to apply HTML transformation to the message before logging it to *log.txt* or console. Add the statements in *LoggerTest* class to apply encryption to the message before logging it. The output of logging the message “hello, how are you?” together with the current timestamp should be as below:

```
<HTML><BODY><b>2018-09-22 10:43:09.525: hello, how are you?</b></BODY></HTML>
?2018-09-22 10:43:09.525: hello, how are you
```

4. Write the statements in *LoggerTest* class to produce the following output for the message “Good Bye”.

```
<HTML><BODY><b>eGood By</b></BODY></HTML>
```

*Note: In this case study, the decorator* adds some additional functionality before forwarding requests to the underlying object.