19

DECORATOR

This pattern was previously described in GoF95.

DESCRIPTION

The Decorator Pattern is used to extend the functionality of an object *dynamically* without having to change the original class source or using inheritance. This is accomplished by creating an object wrapper referred to as a *Decorator* around the actual object.

CHARACTERISTICS OF A DECORATOR

- The `Decorator` object is designed to have the same interface as the underlying object. This allows a client object to interact with the `Decorator` object in exactly the same manner as it would with the underlying actual object.
- The `Decorator` object contains a reference to the actual object.
- The `Decorator` object receives all requests (calls) from a client. It in turn forwards these calls to the underlying object.
- The `Decorator` object adds some additional functionality before or after forwarding requests to the underlying object. This ensures that the additional functionality can be added to a given object externally at runtime without modifying its structure.

Typically, in object-oriented design, the functionality of a given class is extended using inheritance. Table 19.1 lists the differences between the Decorator pattern and inheritance.

EXAMPLE

Let us revisit the message logging utility we built while discussing the Factory Method and the Singleton patterns earlier. Our design mainly comprised a Logger interface and two of its implementers — `FileLogger` and `ConsoleLogger` — to log messages to a file and to the screen, respectively. In addition, we had the `LoggerFactory` class with a factory method in it.

Table 19.1 Decorator Pattern versus Inheritance

Decorator Pattern	*Inheritance*
Used to extend the functionality of a particular object.	Used to extend the functionality of a class of objects.
Does not require subclassing.	Requires subclassing.
Dynamic.	Static.
Runtime assignment of responsibilities.	Compile time assignment of responsibilities.
Prevents the proliferation of subclasses leading to less complexity and confusion.	Could lead to numerous subclasses, exploding class hierarchy on specific occasions.
More flexible.	Less flexible.
Possible to have different decorator objects for a given object simultaneously. A client can choose what capabilities it wants by sending messages to an appropriate decorator.	Having subclasses for all possible combinations of additional capabilities, which clients expect out of a given class, could lead to a proliferation of subclasses.
Easy to add any combination of capabilities. The same capability can even be added twice.	Difficult.

The `LoggerFactory` is not shown in Figure 19.1. This is because it is not directly related to the current example discussion.

Let us 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. Let us consider the following two small features that clients would like to have:

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

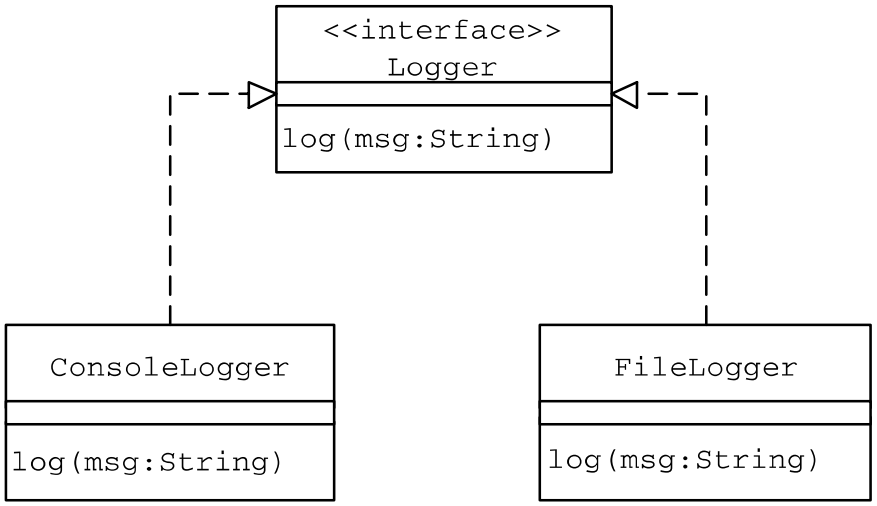

Figure 19.1 Logging Utility Class Hierarchy

Table 19.2 Subclasses of `FileLogger` and `ConsoleLogger`

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.

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 (Table 19.2).

As can be seen from the class diagram in Figure 19.2, a set of four new subclasses are added in order to add the new functionality. 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.

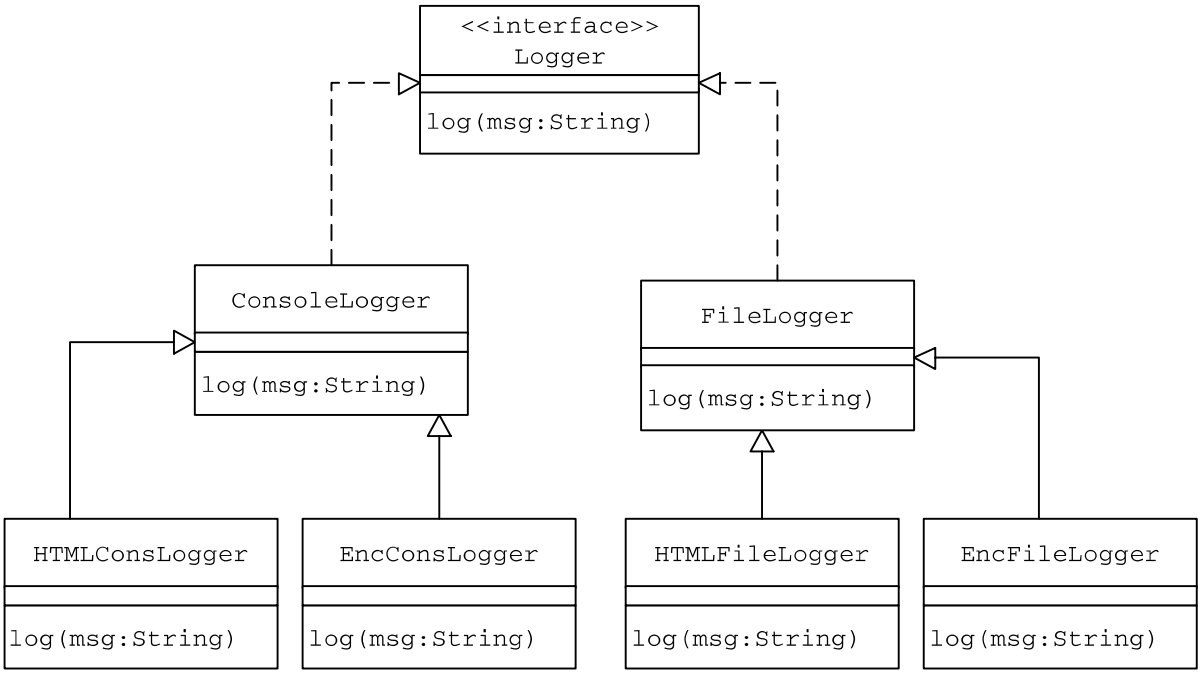

Figure 19.2 The Resulting Class Hierarchy after Applying Inheritance to Add the New Functionality

Listing 19.1 `LoggerDecorator` Class

```
public class LoggerDecorator implements Logger {
  Logger logger;
  public LoggerDecorator(Logger inp_logger) {
    logger = inp_logger;
  }
  public void log(String DataLine) {
    /*
      Default implementation
      to be overriden by subclasses.
    */
    logger.log(DataLine);
  }
}//end of class
```

The Decorator pattern comes to our rescue in situations like this. The Decorator pattern recommends having a wrapper around an object to extend its functionality by object composition rather than by inheritance.

Applying the Decorator pattern, let us define a default root decorator `LoggerDecorator` (Listing 19.1) for the message logging utility with the following characteristics:

- The `LoggerDecorator` contains a reference to a `Logger` instance. This reference points to a `Logger` object it wraps.
- The `LoggerDecorator` implements the `Logger` interface and provides the basic default implementation for the `log` method, where it simply forwards an incoming call to the `Logger` object it wraps. Every subclass of the `LoggerDecorator` is hence guaranteed to have the `log` method defined in it.

It is important for every logger decorator to have the `log` method because a decorator object *must* provide the same interface as the object it wraps. When clients create an instance of the decorator, they interact with the decorator in exactly the same manner as they would with the original object using the same interface.

Let us define two subclasses, `HTMLLogger` and `EncryptLogger`, of the default `LoggerDecorator` as shown in Figure 19.3.

CONCRETE LOGGER DECORATORS

HTMLLogger

The `HTMLLogger` (Listing 19.2) 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.

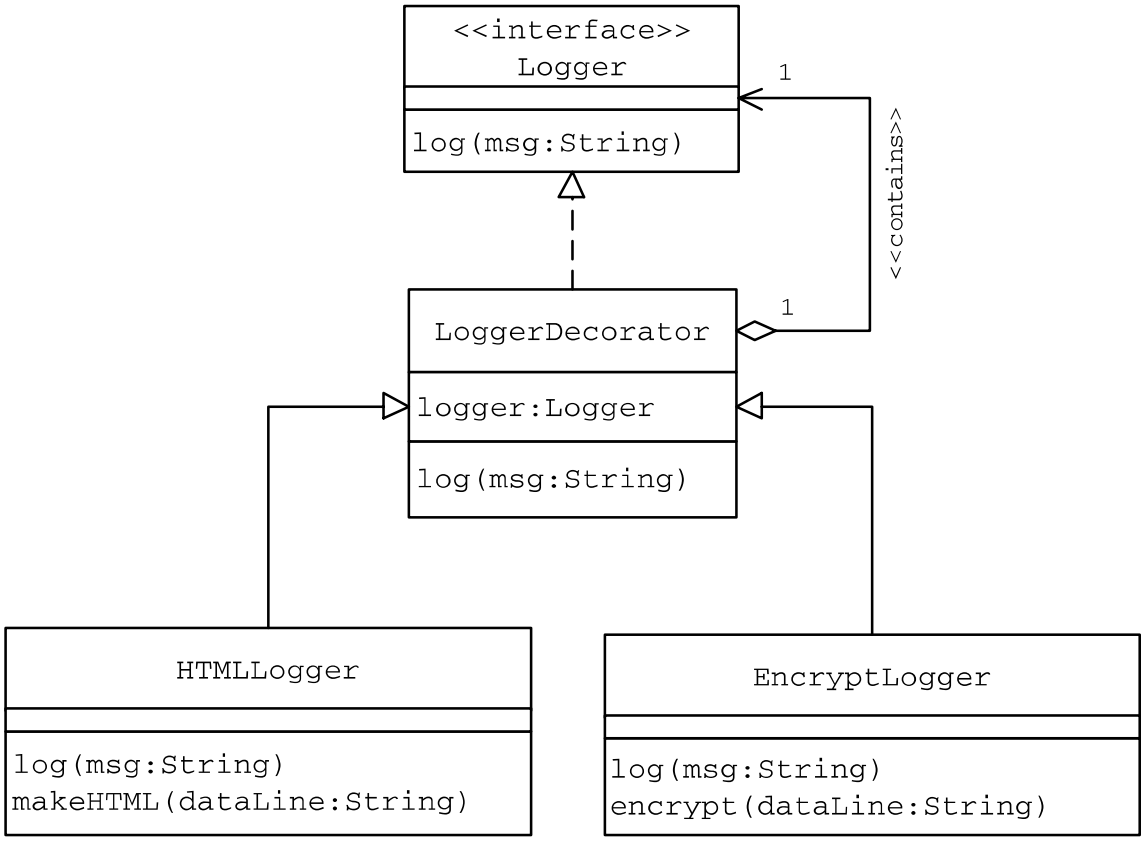

Figure 19.3 The Decorator Class Structure for the Logging Utility to Add the New Functionality

EncryptLogger

Similar to the `HTMLLogger`, the `EncryptLogger` (Listing 19.3) overrides the `log` method. Inside the `log` method, the `EncryptLogger` implements simple encryption logic by shifting characters to the right by one position and sends it to the `Logger` instance it contains for logging.

The class diagram in Figure 19.4 shows how different classes are arranged while applying the Decorator pattern.

In order to log messages using the newly designed decorators a client object (Listing 19.4) 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 Step 1 as an argument to its constructor.
- Invoke methods on the `LoggerDecorator` instance as it would on the `Logger` instance.

Figure 19.5 shows the message flow when a client object uses the `HTMLLogger` object to log messages.

Listing 19.2 `HTMLLogger` Class

```
public class HTMLLogger extends LoggerDecorator {
  public HTMLLogger(Logger inp_logger) {
    super(inp_logger);
  }
  public void log(String DataLine) {
    /*
      Added functionality
    */
    DataLine = makeHTML(DataLine);
    /*
       Now forward the encrypted text to the FileLogger
      for storage
    */
    logger.log(DataLine);
  }
  public String makeHTML(String DataLine) {
    /*
      Make it into an HTML document.
    */
    DataLine = "<HTML><BODY>" + "<b>" + DataLine +
      "</b>" + "</BODY></HTML>";
    return DataLine;
  }
}//end of class
```

ADDING A NEW MESSAGE LOGGER

In case of the message logging utility, applying the Decorator pattern does *not* lead to a large number of subclasses with a fast growing class hierarchy as it would if we apply inheritance. Let us say that we have another Logger type, say a `DBLogger`, that logs messages to a database. In order to apply the HTML transformation or to apply the encryption before logging to the database, all that a client object needs to do is to follow the list of steps mentioned earlier. Because the `DBLogger` would be of the `Logger` type, it can be sent to any of the `HTMLLogger` or the `EncryptLogger` classes as an argument while invoking their constructors.

Listing 19.3 `EncryptLogger` Class

```
public class EncryptLogger extends LoggerDecorator {
  public EncryptLogger(Logger inp_logger) {
    super(inp_logger);
  }
  public void log(String DataLine) {
    /*
      Added functionality
    */
    DataLine = encrypt(DataLine);
    /*
      Now forward the encrypted text to the FileLogger
      for storage
    */
    logger.log(DataLine);
  }
  public String encrypt(String DataLine) {
    /*
      Apply simple encryption by Transposition…
      Shift all characters by one position.
    */
    DataLine = DataLine.substring(DataLine.length() - 1) +
               DataLine.substring(0, DataLine.length() - 1);
    return DataLine;
  }
}//end of class
```

ADDING A NEW DECORATOR

From the example it can be observed that a `LoggerDecorator` instance contains a reference to an object of type `Logger`. It forwards requests to this `Logger` object before or after adding the new functionality. Since the base `LoggerDecorator` class implements the `Logger` interface, an instance of `LoggerDecorator` or any of its subclasses can be treated as of the `Logger` type. Hence a `LoggerDecorator` can contain an instance of any of its subclasses and forward calls to it. In general, a decorator object can contain another decorator object and can forward calls to it. In this way, new decorators, and hence new functionality, can be built by wrapping an existing decorator object.

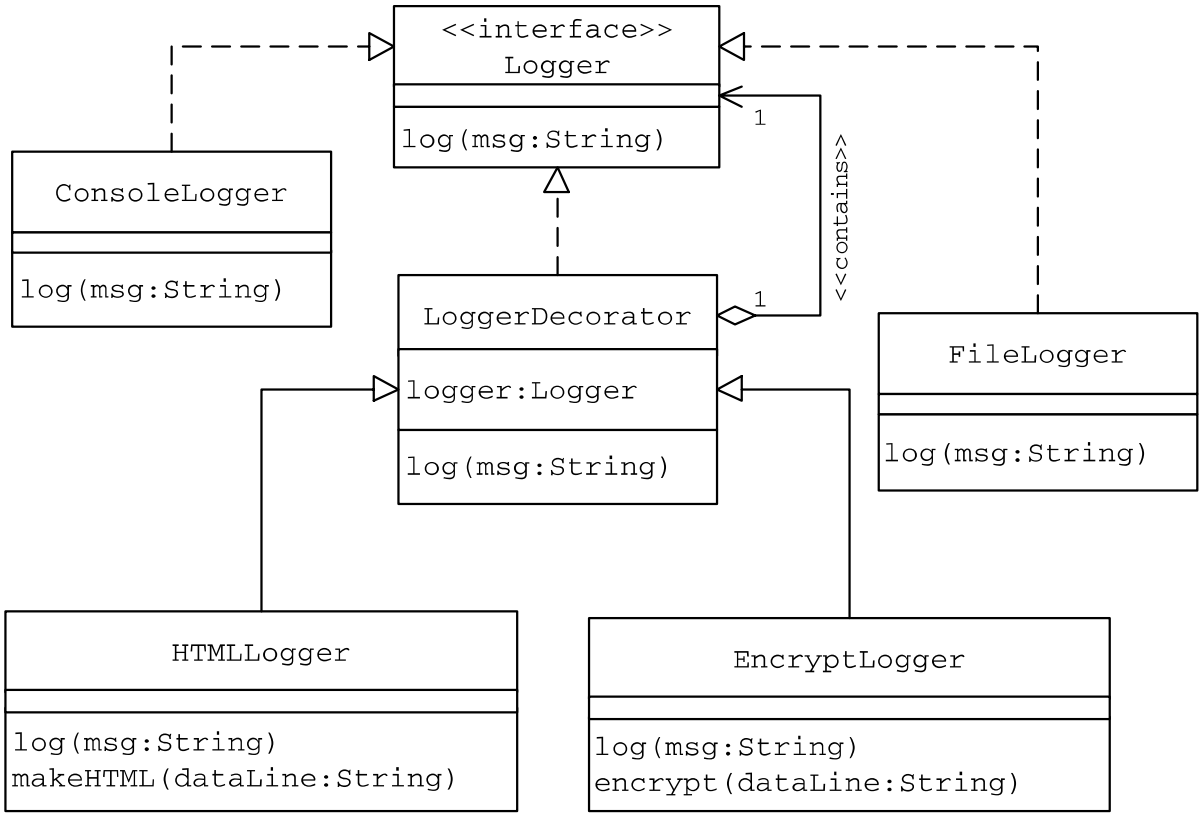

Figure 19.4 Association between Different Logger Classes and Logger Decorators

Listing 19.4 Client `DecoratorClient` Class

```
class DecoratorClient {
  public static void main(String[] args) {
    LoggerFactory factory = new LoggerFactory();
    Logger logger = factory.getLogger();
    HTMLLogger hLogger = new HTMLLogger(logger);
    //the decorator object provides the same interface.
    hLogger.log("A Message to Log");
    EncryptLogger eLogger = new EncryptLogger(logger);
    eLogger.log("A Message to Log");
  }
}//End of class
```

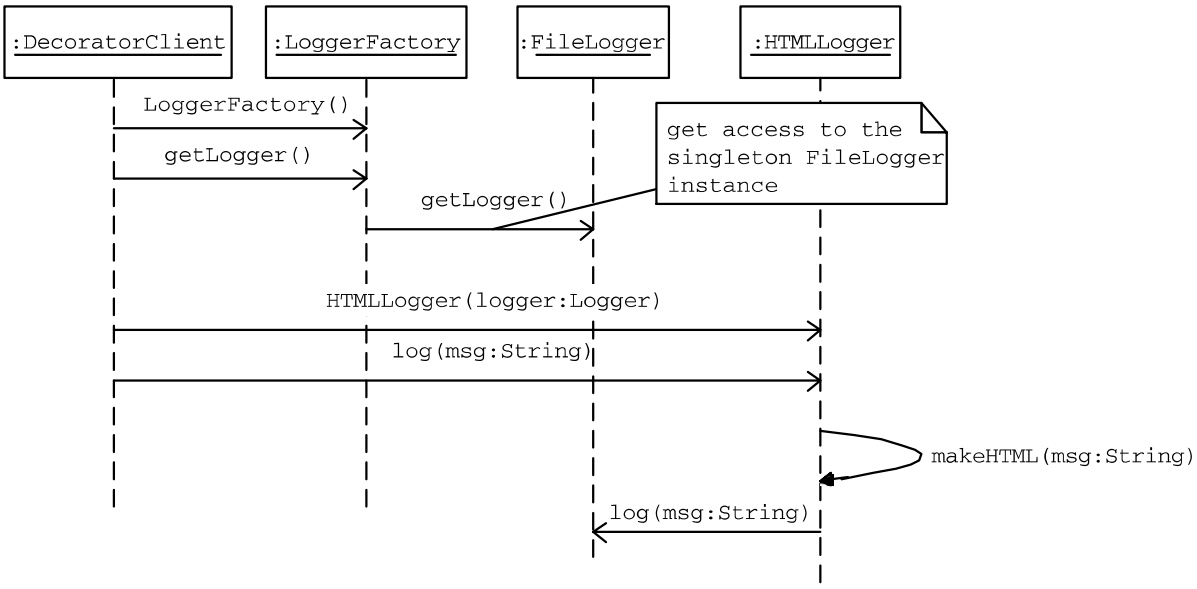

Figure 19.5 Message Flow When a Client Uses the `HTMLLogger` (Decorator) to Log a Message

PRACTICE QUESTIONS

1. Create a `FileReader` utility class with a method to read lines from a file.
2. The `EncryptLogger` in the example application encrypts a given text by shifting characters to the right by one position. Create a Decorator `DecryptFileReader` for the `FileReader` to add the decryption functionality, after reading data from a file.
3. Enhance `DecoratorClient` class to do the following:
   - Write a message to a file using the `EncryptLogger`.
   - Read using the `DecryptFileReader` decorator to display the message in an unencrypted form.