DECORATOR PATTERN

Enginyeria del Software I
UdG
Darío Madrid Alonso

Description

- The Decorator Pattern is used to extend the functionality of an object dynamically
  - No inheritance
  - No changes in the class source
  - How? Object Composition
- Accomplished creating an object wrapper around the actual object

Characteristics

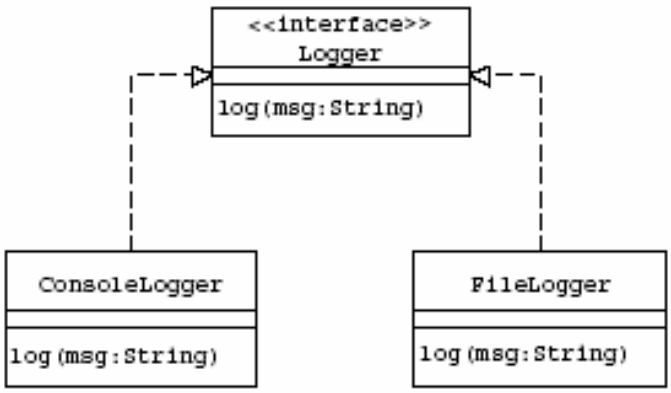
- The Decorator Object:
  - Has the same interface as the underlying object
    - Clients interact in the same way with the new object
  - Contains a reference to the actual object
  - Receive and forward calls from a client to the underlying object
  - Adds additional functionality before or after forwarding requests

Decorator Pattern vs. 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.

EXAMPLE: Message Logging Utility

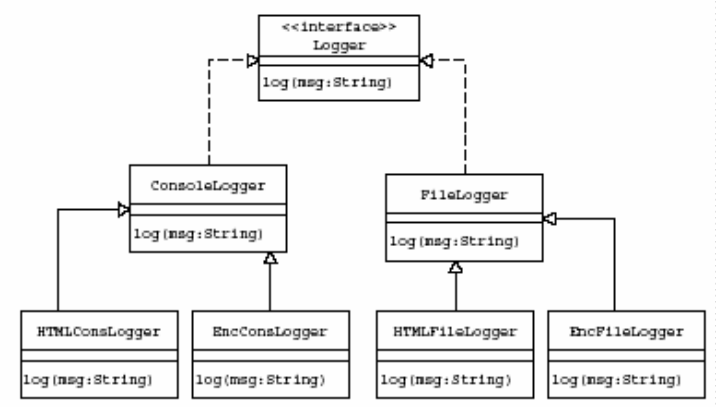
- Initially:



NOTE: LoggerFactory not shown in this figure

EXAMPLE (Inheritance):

- Suppose new functionalities:
  - Transform an incoming message to an HTML document
  - Apply simple encryption on an incoming message



Class Explosion!!!

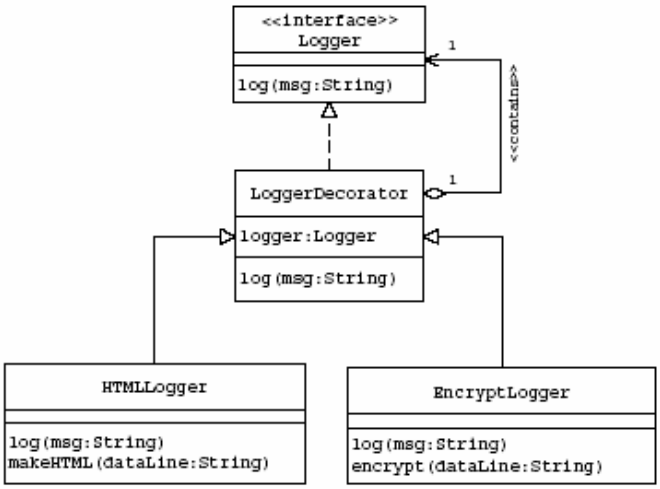
EXAMPLE (Decorator Pattern):

- Define a Default root decorator “LoggerDecorator”:
  - Contains a reference to a Logger instance that points to a Logger object it wraps.
  - Implements the Logger interface and provides the basic default implementation for the log method, forwarding calls to the Logger object it wraps.
- Define two subclasses: HTMLLogger and EncryptLogger

EXAMPLE:

- 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
```



EXAMPLE:

- Concrete Logger Decorators:
  - Both HTMLLogger and EncryptLogger:
    - Overrides Default Log method
    - Implement new functionalities
- Client needs to:
  - Create an appropriate Logger instance (FileLogger or ConsoleLogger using factory method)
  - Creatre an appropiate LoggerDecorator instance by passing the Logger instance created in Step 1 to its constructor
  - Invoke methods on the LoggerDecorator instance (Step 2) as it would on the Logger instance (Step 1)

EXAMPLE (Client Code):

```
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
```

EXAMPLE: HTML Decorator 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
```

```
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
```

EXAMPLE:

- What if we add a new Message Logger (DB Logger)?
  - A client only needs to do the steps explained on the last slide
  - DB Logger would be of Logger type and it can be sent to any of the HTMLLogger or EncryptLogger as an argument while invoking their constructors)
- And adding new Decorators?
  - Decorators inside Decorators!

EXAMPLE:

- Association between Different Logger Classes and Logger Decorator

