Description

"Attach additional responsibilities to an object dynamically."[GoF] In other words, this pattern allows you to add logic to another class without the other class or the application itself being aware of the new logic. The two key concepts of the decorator pattern are adding functionality dynamically and doing it in a seamless manner. This pattern is similar in concept to inheritance but without many of its downsides (ie a rigid structure) and can be dynamically added or removed at runtime. A decorator class will implement all the methods of the class it will be decorating. Each method in the decorating class then has the opportunity to impose its own logic before, after or in place of the decorated class's method(s). A great, real-world example of using this pattern is to decorate a class with logging information - which I will discuss in more detail below.   
  
When and when not

The Decorator pattern doesn't seem to be commonly used, whether it be in applications or TopCoder components. The only time I've ever seen the pattern employed (beyond applications where I've used it) was to add logging. Because of that, I'd like to explore the logging example in depth to show you some important benefits of the decorator pattern.   
  
Let's say you are working on an application and have been experiencing some bugs in the code. You want to add some logging to 'see' what interactions are happening between objects. Most people will typically think of doing two options:

1. Using your favorite logging component and manually insert logging code into your classes;
2. Using something like Aspect Oriented Programming (AOP) to create logging aspects for those classes and/or methods.

Both of these methods have their downsides. Manually inserting logging code means you have access to the source code and are able to recompile the application. This, unfortunately, means that you are creating a dependency between the code and the logging component. Additionally, it means that the application code will always execute your logging code (one of the reasons logging components are lean and have 'shortcuts' methods to prevent logging in certain cases). AOP can usually get around having the source code (since it can interweave byte code), as it requires recompiling the application and therefore will always execute the logging code as well (Check this to make sure the intent is maintained). While these methods usually are fine in an application environment, they both have significant difficulties when used with components where the source code is unavailable. Likewise, neither method is really effective in dynamically adding logging to an existing, running system or effective when you want to decorate specific instances of the class rather than all instances of the class.   
  
An alternative would have been to use the Decorator pattern to decorate the class and/or methods you want to log. This overcomes the biggest disadvantages of the prior methods because the system can dynamically, at runtime, decorate all or even a single instance of class with the logging functionality. Like AOP, you also have a clear separation between the application code and the logging code base (i.e. the application code doesn't even know that logging is happening). Unlike the other methods, the logging code is only executed when you have decorated an object instance - all other instances will run at full speed.   
  
If you are designing a component and have implemented a strategy pattern, (see Implementation Details below for the reasoning), you should consider whether there is optional functionality (like logging) that can be used dynamically at runtime. If there is, you have a very good candidate for the decorator pattern.   
  
The only time I have really seen when a decorator pattern was used incorrectly was a case where objects were decorated with security measures. While this seems to be a good idea in concept, (keeping a clear separation between the logic and the security), it's generally a very bad approach. When you decorate an object, that other object has been created somewhere and is a live, separate entity that can be 'leaked' somehow (either through the API, reflection or even more advanced methods). If the underlying object is 'leaked' and you used a security decorator, that security can be fully bypassed because it's not part of that class. Decorator patterns should only be used where the functionality of the decorator is fully optional and nothing in the application requires it.   
  
Considerations

The biggest consideration happens when the application needs to know what class or type it's dealing with. When this situation occurs, the widespread practice is to simply do 'instance' checks. This practice falls apart when you use the decoration pattern because the decorating class does not belong to the same class hierarchy that the application is checking for. A common practice in using a decorator pattern is to provide an accessor function to retrieve the underlying object (which can then be checked). While this accomplishes the goal, it forces the application into recognizing the decorating class directly:

if(someClass instanceof DecoratorClass) {

someClass = ((ACMEWidget) someClass.getDecorated();

}

if (someClass instanceof ACMEWidget) {

... Do something ...

}

The above becomes even more complex when multiple decorators are used. A better approach would be to define a method in the decorated class call 'getID' or 'getType' or something similar that would identify the class being used. Any decorating class can then simply delegate this call to the underlying decorated class (and so forth if decorators decorate other decorators) and simplify the above call to:

If (someClass.getID() == ACMEWidget.ID) {

... Do something ...

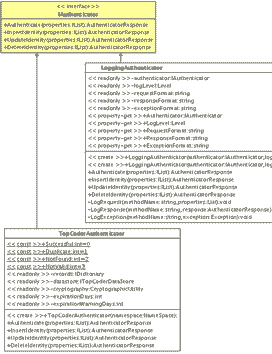
}

Decorating classes can then be seamlessly integrated into the application without any changes.

Implementation details  
A decorator pattern generally requires a strategy pattern to be implemented because you don't want the application to be aware of the decorating class itself. Without the strategy pattern, the application would need to know about the decorating class in order to be able to use it and that would break one of the key concepts of the pattern.   
  
To implement the decorator pattern, you need to implement three things:

1. An interface defining the API
2. An implementation of the interface
3. An implementation of the interface that has another implementation that it decorates.

Let's take a look at the .NET Authentication component.

[](http://www.topcoder.com/i/education/decorator1_lg.gif)  
\* Please note that I have rearranged the classes in this diagram.

This .NET Authentication component can be used by the application to authenticate some type of user. The designer chose to implement the decorator pattern to logging of authenticator calls - possibly as a debugging aid or possibly as an auditing aid to the end user component. This design has the three parts defined above:

1. IAuthenticator is an interface that defines the API
2. TopCoderAuthenticator is an implementation of the IAuthenticator API
3. LoggingAuthenticator is the decorator implementation.

If we look at the LoggingAuthenticator class, you will see that all constructors take an implementation of IAuthenticator as an argument. This is a required feature of the decorator class because it needs to decorate another class. The decorating instance then stores this decorated instance as an instance level variable.   
  
The LoggingAuthenticator implements each of the IAuthenticator methods. Each of those methods will then:

1. First call the helper method LogRequest to log the request;
2. Call the same named method in the authenticator instance that was saved from the constructor;
3. Lastly call the helper method LogResponse/LogException with the results of the call.

Please note that this logic is specific to the LoggingAuthenticator. A decorator can really do anything within the method - including calling the decorated instance first then performing logic, performing logic first then calling the decorated instance, or simply perform some logic and ignore the decorated instance (although this is less useful).   
  
The end using application may have code like:

// Get an authenticator instance

IAuthenticator auth = authenticatorFactory["topcoder"];

// Call the authenticate with our principle

auth.Authenticate(principle);

The Authenticator Factory is then free to return (using its own logic which may be based on dynamic factors) the implementation or a decorated implementation - either way, the application has no direct knowledge of that. If it returned an instance of "new TopCoderAuthenticator(…)", the application would simply be authenticating directly with the implementation. If it returned an instance of "new LoggingAuthenticator(new TopCoderAuthenticator(…), …)", the application would be authenticating and writing out logging messages at the same time. The point is, both the application and implementation are unaware of the logging functionality that was added between it.