Enhydra Design Patterns for ASP's

Currently Enhydra is receiving lots of attention for it's capabilities as a server for wireless applications but, Application Service Providers (ASP's) can use Enhydra to dramatically improve their ability to provide custom branding while at the same time reducing development costs. In this article I will outline some the design patters that we used with Enhydra to create a set of applications that support thousands of very different looks from a single code base. Even more impressive is the potential for evolution, instead of using file based development, like we did back in the days of C programing, we can begin to apply object oriented concepts, like classification and abstraction, to HTML development.

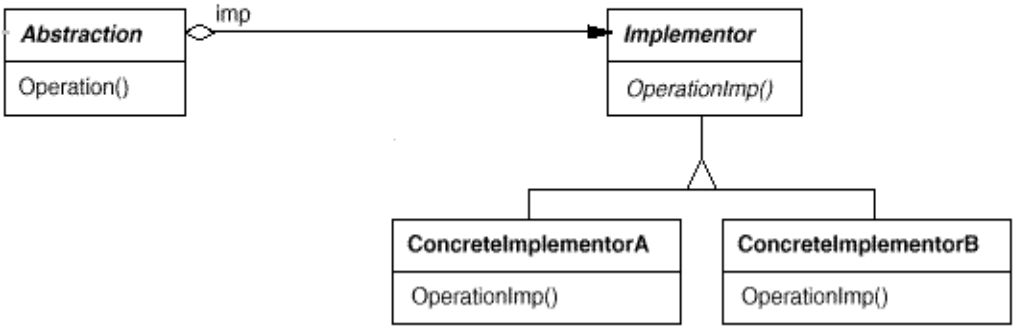
The Requirement

Our client is a ASP for a group of about thirty commercial financial institutions. Each of these commercial institutions uses the products internally and resells them to retail financial institutions. Each of these financial institutions has a unique look and feel that is an important part of their brand. Brand recognition is a very valuable commodity to a financial institution and having a look and feel that consistently matches their branding is an absolute requirement. Developing applications that can support thousands of different looks based on a user's login is a difficult task and before Enhydra we were limited to the items that are available in Cascading Style Sheets (i.e., colors and fonts) and perhaps a different logo in the corner.

The Bridge Pattern

The first pattern we used is the Bridge Pattern from the "Gang of Four" book[1]. This pattern allow the interface of an object to bridge to several different implementations. The pattern looks like this:

1Design Patterns, Gamma et. al., Addison-Wesley, 1995



Enhydra's XMLC compiler has built-in support for the Bridge Pattern. To use it requires both the "-generate both" and "-implements" options on the XMLC compiler.

To use this pattern effectively requires a little up front analysis. What we need to look for is classifications or classes of pages. For example, we may find that our application has several HTML forms. Yes, each instance of the HTML form is different, it may have different input elements appearing in a unique order and positioned. But closer examination revels that all the HTML forms are built from the same finite set of elements, this is a sure sign of a correct classification. In our experience, you will find a very small number of classes (form, detail display, summary display, etc...) in all your applications.

When a classification is discovered, the next step is to create some code that defines the class (the HTML Form class). This class definition is used as a template to create all the instances of the class that the application requires. In our case the class is defined in HTML. The class HTML file will need to contain one example of each of the elements that that will be needed to create the instances. For example, our form class will need to have a sample submit button, a sample reset button and several input element samples. Each of the sample elements should have a unique id attribute so that the XMLC compiler will generate connivence methods that make it easy to get a hold of these elements when we start programing.

With the class defined, we are ready to generate the abstract interface and reference implementation Java classes. Your XMLC command will need to use the "-generate both" option. This will cause XMLC to generate an abstract interface class the we will use in all of our servlets. The implementation generated at this stage can be used as a reference implementation so that you can start coding your servlets right away. This is great because the developers can start building the application while the art department is still working their magic on the look and feel.

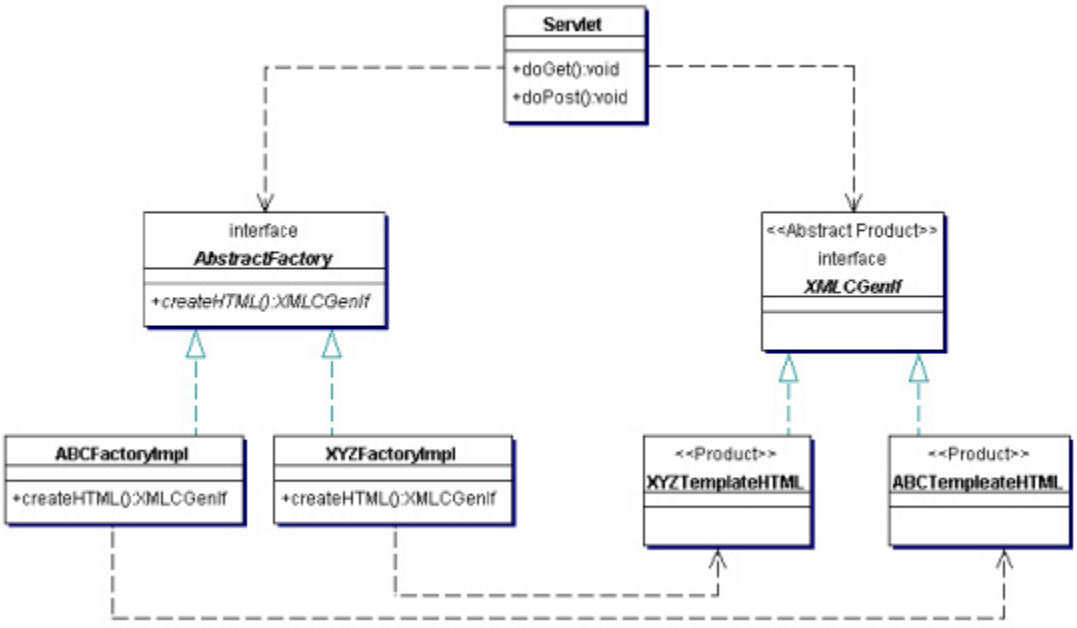
After this, adding new looks to your application is done by creating a new class definition HTML file that has the same elements as original class definition. The elements themselves can be very different but each and every id attribute that was in the original

class must be in the subsequent classes. For example, the reference implementation HTML file might have a plain button element for a submit button, later implementations can define fancy rollover buttons and as long as the id attribute matches the original class it will work just fine.

The question that always comes up is, "what if I miss one of the attributes"? Well XMLC's Bridge Pattern support can help. Subsequent implementations are processed with the "-implements <interface>" option. If the abstract interface is used with this option the generated Java implementation will implement the abstract interface and if any of the id attributes don't match or are missing you'll get a compile error. Because you have compile time checking you can safely let other developers produce HTML class definitions.

Abstract Factory Pattern

This pattern also comes from the "Gang of Four" book. The Abstract Factory creates instances of an object without the client actually know what the implementation class is. The pattern looks like this:



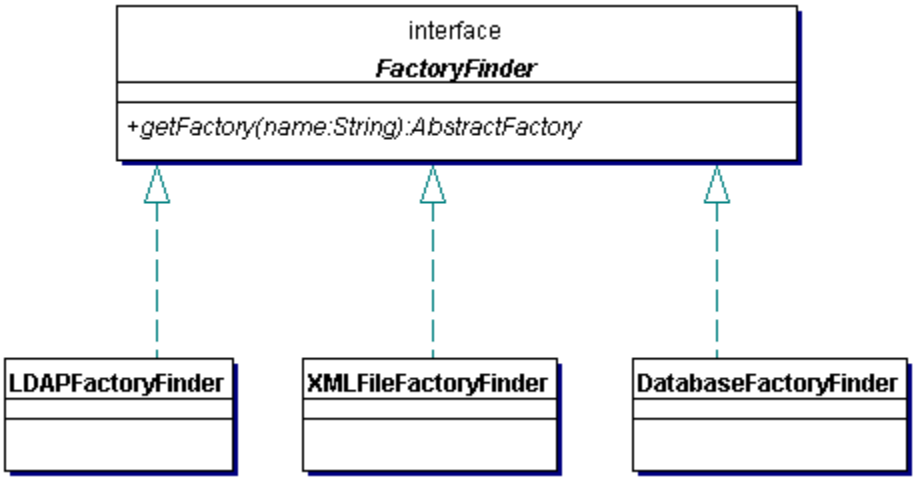
We are using the pattern because we don't want our servlets to know which implementation from the Bridge Pattern that we are using. That way the servlet is only dependent on the interface not any actual implementation.

To use this pattern we need to have our HTML classes defined because we will need a create method for each of our interface classes on the Abstract Factory interface. Then we'll need to create a concrete factory implementation for each related group of classes. In our case, each look and feel needs it's own factory implementation. Coding all these factories was tedious, to say the least, but Java's introspection capabilities and XML can help. We created a factory implementation that reads in an XML configuration file. The

XML file defines which HTML class the factory implementation should instantiate. This means that we only need one concrete factory implementation that changes it's behavior based on the XML file that it loads. Well, you might be asking how do we make sure that we load the right XML file and aren't there scaleability limits to this approach? Of course the answer to both is yes or I wouldn't have brought them up.

Factory Finder Pattern

This pattern is deviation on the Abstract Factory pattern, in this case the objects created by the Abstract Factory are Abstract Factories. A good example of this pattern in the CORBA Life Cycle Service. In our case we'll use a Factory Finder to get the correct factory that will create the HTML instances. The Factory Finder has a single method that creates the correct object based on a supplying a string name. We started out with a XML file configured factory finder but soon found that it broke down after several thousand XML files were loaded. So, we switched to an implementation of the Factory Finder that was loaded from a relational database. This also proved to have scaleability limits so, we switched to an implementation that loaded data from an LDAP directory. The LDAP directory has better read performance than a relational database.



Putting It All Together

OK, we've got a Factory Finder the returns a Factory class that can create objects that can create objects that will give us a HTML class. The HTML classes have examples of all the elements that are required to make the screens that our application requires. To build an application you need to take all the above add a servlet or two and stir. Each servlet is passed the name of the look and feel to use. The servlet, in turn, uses the name to lookup the correct Factory in the Factory Finder. The Factory Finder found the Factory either it's own cache or the LDAP directory. The servlet the uses the Factory to get an instance of the HTML class. The servlet then makes copies of the example elements in the HTML class to create the HTML page, the example elements are then removed from the HTML class and the class return HTML to the browser. The servlet only knows the class interface and has no direct dependencies on anything to do with the look and feel. We

have found that using the above techniques we can deploy a new look and feel, that is consistent across all of our applications, in about 20 or 30 minutes.

About the author:

Nick Xidis is a Senior Principal Architect with Iconixx in Overland Park, Kansas. He can be reached at nxidis@acm.org.

You can learn more about Iconixx at http://www.iconixx.com/