How I Use Transfer - Part III - Abstract Objects

Posted At : June 30, 2008 10:39 AM | Posted By : Bob Silverberg
Related Categories: OO Design, How I Use Transfer, ColdFusion, Transfer

In the previous post in the series I discussed the four types of objects that comprise my model. As I was creating instances of those object types I realized that they did a lot of the same things. For example, my UserService needed methods like getUser(), updateUser(), deleteUser() and listUsers(), while my ProductService needed getProduct(), updateProduct(), deleteProduct() and listProducts(). Not only do they need to do similar things, but they pretty much do them the same way. Now that's a lot of code duplication.

So I created an AbstractService, which each of my concrete Service Objects extend. My AbstractService has methods like Get(), Update(), Delete() and GetList(). For the most part, the code required for getUser() is identical to the code required for getProduct(), with the only difference being the business objects with which they interact (User and Product, respectively), so I was able to write a single, parameterized method, which will work for most of my Service Objects.

I call this object abstract because it shouldn't be instantiated on its own. It is used solely as a base object which is extended by most of my concrete Service Objects. The methods in this objects are designed to work inside of the objects which extend it, but it is also totally acceptable to extend (via super) or override these methods in the objects which extend it. So, neither my UserService nor my ProductService contain a Get() method, but my ReviewService does. It contains a Get() method that overrides Get() in the AbstractService.

I'm not going to describe my entire AbstractService in this post, but let's look at the Get() method as an example of what I'm talking about:

```
<cffunction name="Get" access="public" returntype="any" output="false" hint="I return a Business Object, ready to be used.">

 <cfargument name="theId" type="any" required="yes" hint="The Id of the business object" />

 <cfargument name="needsClone" type="any" required="false" default="false" hint="Should a clone be returned?" />

 <cfargument name="args" type="any" required="no" default="" hint="A package of data to load into the object" />

 <cfargument name="TransferClassName" type="any" required="no" default="#getTransferClassName()#" hint="The class name of the business object, as defined to Transfer" />

 <cfargument name="FieldList" type="any" required="no" default="" hint="A list of fields to populate" />

 <cfset var theTO = 0 />

 <cfif arguments.theId EQ 0>

  <cfset theTO = getTransfer().new(arguments.TransferClassName) />

 <cfelse>

  <cfset theTO = getTransfer().get(arguments.TransferClassName,arguments.theId) />

  <cfif arguments.needsClone>

   <cfset theTO = theTO.clone() />

  </cfif>

 </cfif>

 <cfif IsStruct(arguments.args)>

  <cfset theTO.populate(arguments.args,arguments.FieldList) />

 </cfif>

 <cfreturn theTO />

</cffunction>
```

So, what's going on here? This method actually serves three use cases:

1. I simply want to retrieve a Business Object for display purposes
2. I want to retrieve a Business Object because I want to create a new instance of the object
3. I want to retrieve a Business Object because I want to update an existing instance of the object

Let's skip past the arguments and look at the code. First if I want to create a new object, then the Id passed in will be 0 (for the most part), so in that case I simply call getTransfer().new(), passing in the TransferClassName. Otherwise I want to return an existing object from Transfer so I call getTransfer().get(). If I'm working with an existing object, I may want to manipulate that object, in which case I want to create a clone() of it and return that. So now I've got my Transfer Object and I need to decide whether to return it as is, or to fill it with data. If I have passed in an argument called args, which is a struct, then I call the populate() method on my Transfer Object to load that data into the Transfer Object before returning it. That populate() method does a bunch of different things, which I'll cover in a future post. Don't worry about arguments.FieldList for now, most of the time it remains an empty string and isn't used.

So, there's an example of a method in an abstract object. It achieves three different things, and it will work with almost all of my concrete Service Objects. In fact, in my current application I only need to override this method in one of my concrete Service Objects. The rest inherit it directly from the AbstractService object.

The other nice thing about this Get() method is that it allows me to encapsulate my communication with Transfer. This Get() method is kind of like a Business Object factory, in that the logic required to create Business Objects is limited to this one method. If I were ever to stop using Transfer (gasp!) and choose to move to a different ORM, little about my service layer would have to change.

I know I've left a couple of things out in my explanation of the Get() method, but I think this simple introduction to abstract objects has already become less than simple. I'll go into more detail about the implementation of this object in the next post.

Getting back to the four object types mentioned in the previous post, I also have an AbstractGateway which all of my concrete Gateway Objects extend, as well as an AbstractTransferDecorator object, which is used as a base object for all of my concrete Transfer decorators. Because I rely on Transfer to create most of my business objects my AbstractTransferDecorator is really like an abstract Business Object. There are no abstract objects for my Utility Objects as they are all standalone objects that each serve a single purpose.

To recap, my model consists of 4 object types:

- Service Objects, most of which extend AbstractService.cfc
- Gateway Objects, all of which extend AbstractGateway.cfc
- Business Objects, most of which are based on AbstractTransferDecorator.cfc
- Utility Objects, which stand alone

In the next installment I'll take a closer look at the AbstractService Object.