How I Use Transfer - Part II - Model Architecture

Posted At : June 24, 2008 6:36 AM | Posted By : Bob Silverberg
Related Categories: OO Design, How I Use Transfer, ColdFusion, Transfer

I think that the best place to start is with the high level architecture of my model. I have four types of objects in my model:

- Service Objects
- Gateway Objects
- Business Objects
- Utility Objects

Note that these names are not necessarily the generally accepted or "proper" names for these types of objects in the Object Oriented community. These are terms that I have chosen to use, and I will define what I mean by them.

Service Objects

These act as the API for my model so collectively I refer to them as a Service Layer. A controller always talks to a Service Object to interact with the model. If a form needs to be processed the Controller will pass the data to a Service, and if the Controller needs data for a view it will ask a Service for the data. Often a controller will need data from a query, in which case it will ask the Service for the data, and the Service in turn will ask a Gateway for the data.

Gateway Objects

Gateway Objects are used to retrieve multiple records from the database. It's as simple as that. So Gateways consist of methods which execute and return SQL queries. These are sometimes calls to stored procedures, sometimes hand-coded SQL, and oftentimes performed by Transfer either by List() methods or via TQL.

Business Objects

Business Objects represent the entities with which my application is concerned. These are the "nouns" of the business, such as User, Order, Product, etc. For the most part, Business Objects interact with a single record in a database table. Most of my Business Objects are generated for me by Transfer, which means that each Business Object corresponds to a single database table. Although that is true, I make extensive use of Transfer Decorators, which actually allow my Business Objects to interact with more than one database table.

For example, I often create a method in a decorator which accesses a Service Object, which then interacts with other Business Objects or Gateways. Also, I create methods in decorators which access either the Parent or Children of the current object, so once again I'm not limited to dealing with just one database table within a Business Object.

Utility Objects

Utility Objects are not concerned with database access. They perform utility functions, such as email, XML conversion, file handling, etc. I include objects which help manage the model, such as a Bean Injector, a Transient Factory and Observers in this category as well. It's basically a catch-all for any objects that do not fit into one of the three previous categories.

Some Strategies

I try to employ the following strategies when designing my app:

Try to push the logic as deep into the model as possible.

One of the first mistakes I made when just starting out, which I believe is quite common, was to put too much logic in my controllers. I then learned about the concept of "dumb controllers", meaning controllers that do little more than facilitate communication between the view and the model. I started moving my logic out of my controllers, into my service layer and found that it worked well for me.

The next step was to try to make my service layer as dumb as possible by moving logic into the Business Objects. I do this by moving the logic from my Service Objects into my decorators. The logic that remains in my Service Objects is generally about flow control and managing multiple Business Objects.

This process, of moving logic from one layer to another, is part of a process that is commonly known as **refactoring**.

Create only as many objects as necessary.

I do not create one Service Object per Business Object. Generally I'll create one Service Object per subject area (e.g., User, Order, Product, etc.), and that Service Object will be responsible for managing interactions with a number of Business Objects (e.g., User, UserGroup, Address, etc.). From a Transfer standpoint, these subject areas often correspond to the packages that I define in my transfer.xml file. I generally end up with one Gateway Object per Service Object.

Try to write as little code as possible (DRY).

I make extensive use of abstract classes to help me with that third strategy. In fact, that is the topic of my next post.

But before that, one last comment:

There are many reasons for employing the above strategies, some of which are based on core OO principles. I have found that it is very helpful to familiarize yourself with these principles, but I have also found that as you work through this process you start to get a "feel" for what works and what doesn't work. There have been moments when I've moved some logic from one component to another and then stopped and marvelled at how much "better" the design seems. I have actually found that aspect to be one of the most satisfying things about attempting an OO design.