Beans and DAOs and Gateways, Oh My!

by Sean Corfield

When you decide to incorporate object-oriented programming and design patterns into your ColdFusion toolbox, the most confusing set of concepts is the whole notion of "beans and DAOs and gateways and services." It seems like so much work just to do something that you used to do with a couple of tags - and everyone seems to have strong, and differing, opinions on how best to implement these concepts.

In this article, I'm going to try to demystify why we might want to introduce a number of layers into our applications and then review the options available to us, along with some pros and cons. My hope is that after reading this article you'll feel less intimidated by the terminology and less worried about making the wrong decision about how to structure your application.

A Four-Layer Cake

The simplest way to write an application is just to mix all the code together on a page-by-page basis. We've all done it and I think we all know that it can lead to duplicated code and, ultimately, to an unmaintainable mess as the application grows.

Few people disagree with the idea that it's a good thing to separate out database code, business logic code and presentation code. The core principles behind Fusebox addressed this by using file naming conventions to emphasize the separation into 'qry' files (for database queries), 'act' files (for "actions" - business logic) and 'dsp' files (for display / presentation code).

This is essentially the same principle as the Model-View-Controller design pattern, although that focuses on separating presentation code, application control code and "everything else" - the model.

As we adopt object-oriented principles, we start to represent concepts in our applications using ColdFusion Components (CFCs). We are told that encapsulation is a good idea, so we write CFCs that have getXyz() and setXyz() methods to provide access to our xyz properties. Following the common Java terminology, we refer to these as "beans". Then we have to deal with getting that data in and out of the database and we need to decide what to do with our application's logic.

What we end up with is a somewhat inevitable series of layers in our applications:

- Presentation
- Control
- Business Logic
- Database Access

Each layer has a fairly clearly defined purpose:

- Presentation - the HTML shown to the user, along with the minimal code necessary to display data and loop over sets of data.
- Control - the logic that routes requests (links and form submissions), interacts with the "model" (business logic) and selects the appropriate "views" (presentation) to display.
- Business Logic - this is (or should be) the core of your application, containing all of your business objects and the operations they can perform or have performed on them.
- Data Access - this contains all of the SQL operations necessary to get your business objects' data in and out of the database.

This approach allows us to change our presentation layer or to refactor and optimize the database without changing the core of our application. It also makes our business logic more testable, since we don't have to deal with either the database or the user interface in our testing and can therefore more easily automate our testing. The layered approach often allows us to reuse more of our code across multiple projects because it leads to code which has fewer dependencies on its environment.

A Review Of Recipes

So how do we go about creating those layers? There's no "One True Way", which some people find very frustrating, and others find very liberating. I'm going to look at four possible recipes for building the bottom three layers: Control, Business

Logic, and Database Access. The four recipes I'm going to review sit on a continuum from big, richly-functional objects to small, specialized objects. In the ColdFusion world, the latter end of the continuum is the one that seems more familiar to developers, but I hope this article will make you take a closer look at other possibilities as well. I'll walk through each recipe first and then look at some pros and cons of each.

We'll start with a familiar recipe. For each table in your database, write a bean that represents a row in the table and then write an object that handles the database operations for that bean. We often call this a Data Access Object or DAO. Next write an object that handles database operations that span multiple rows or multiple tables. We often call this a Gateway Object. Then write an object that contains your business logic for that bean. We often call this a Service Object or a Manager Object. Finally, write a controller that is used to wire the user interface to the service or manager. You'll see this approach in sample applications for frameworks as well as on many blogs and mailing lists.

The second recipe will look, on the surface, like a simple variant of that first recipe. Instead of five objects in the mix, there are only four, with the functions of the DAO and the Gateway blended together. Don't be fooled by that apparent similarity - the real differences run much deeper than that. Whilst there is still a bean for every table in the database, the Gateway objects handle database operations for related groups of tables and thus related groups of beans. The Service Objects are also not tied to individual beans but instead contain business logic that operates across multiple objects. Business logic that operates on a single bean is usually written as part of that bean rather than as part of some Service Object. Finally, the Controller Objects interact with the Service Objects and with the beans, in line with how the business logic shifts between the first recipe and this recipe.

Our third recipe typically has fewer ingredients but can be cooked up as a simple variant of either the first or second recipes. The first scenario, a variation on the first recipe, combines beans and DAOs so that the beans themselves know how to get their own data in and out of the database. This is usually referred to as an "active record" and I'll talk more about that in the next section. The second scenario, a variant of the second recipe, combines the Gateway Objects and the beans. These rich business objects know not only how to get their own data in and out of the database but also how to perform database operations that span multiple rows on the table with which they are associated. In both cases, the Service Objects and Controller Objects are unchanged from whichever recipe you used as the basis for Recipe 3.

The fourth and final recipe has the fewest ingredients. In the ideal world of this recipe, there are only business objects. Each business object knows how to interact with the other business objects it needs to get its job done and it knows how to get its data in and out of the database. The intent of this approach is to create a set of collaborating objects, each of which is wholly responsible for an aspect of the application's model. This recipe leads to an application model with fewer, larger objects than any of the other recipes. The design techniques applied in this recipe are often referred to as "Object Think". When you are applying this design technique, you put yourself into the role of each object and ask, "What are my responsibilities?" and "What do I know about?" The first question helps you identify what behavior an object should have (what methods it should implement) and the second question helps you identify what attributes an object has as well as what other objects need to be available to this object (its dependencies).

Even these four basic recipes have a number of variants so that there really is a continuum of approaches to building the basic layers in an application. Let's look at some of the pros and cons of each of these recipes and then we'll take a closer look at some of the ingredients, as well as the "calorie content" of your objects.

As you might expect, each approach has different tradeoffs, and understanding those can help you make more informed decisions about which approach will best suit you - and your application.

The simplest recipe to learn and use is the first one. Starting with a database model, it's an almost mechanical process to build the beans and DAOs, as well as the basic query methods in the Gateway Objects. Indeed, there are a number of code generation tools that can automate this exact process, such as Brian Rinaldi's Illudium PU-36 Code Generator (http://cfcgenerator.riaforge.org) Even though you can't entirely automate the creation of your Service and Controller Objects, the rules for naming each layer are fairly straightforward: a Something bean, SomethingDAO, SomethingGateway, SomethingService and SomethingController. The most obvious downside to this recipe is that it creates a lot of objects and, to the novice, it can look like an awful lot of work to do something that used to be so simple. (It wasn't really so simple but it was something we were used to.) I've started referring to this recipe as the "5:1 Syndrome" because you typically have five objects for every database table. Whilst the automation of code generation can offset the tedium of initially creating all those objects, the repetitive nature of the code can lead to maintenance issues and there are certainly a lot more lines of code in this recipe than in the other three.

Now let's turn our attention to the fourth recipe. For many people this represents a complete shift in thinking because it is

totally object-centric: it's all about responsibilities and behavior, with data taking a back seat. Once you start "thinking in objects" this becomes a very natural way to design objects. In some ways, it's the ideal way to design an object-oriented system but it has a couple of serious downsides. Remember the four layer cake we talked about? One of the key aspects of that layering is separating the database access out of your application logic so that you can refactor and optimize your database more easily. The cake model also makes it easier to test the business logic independent of any database. Unfortunately, this fourth recipe scores badly on both of those measures because the objects are tightly integrated with the database access. It can also be very hard to establish how the dependencies need to be set up so that objects can collaborate without creating separate "manager" objects to help orchestrate
the interactions between otherwise smart business objects.

Next, we'll consider the "active record" recipe and its variants. Ruby on Rails has made this approach very popular and the simplicity of integrating the bean and its database access is very appealing to a number of people. As a variant of the first recipe, this approach has much the same pros and cons: it is simple to learn and it lends itself to automatic generation of beans and their associated database access code, but it still leads to a lot of objects (perhaps the "4:1 Syndrome"). Just as with the first recipe, this can look like a lot of additional work to do something that ought to be fairly straightforward. It also has an additional downside in common with the fourth recipe: blending the database code into the beans makes it harder to refactor and optimize your database without affecting your application code, and it also makes it harder to test your application code in isolation. If the second recipe is used as a basis for "active records", the result is much closer to the fourth recipe so the same concerns apply again: that tight integration with the database can make life harder for us.

Finally, let's consider the second recipe, which blends aggregate and single object persistence into Gateway Objects and endeavors to create smart objects and workflow-based services. This clearly does not suffer from tight coupling between beans and database access. It also doesn't have the problem seen in the fourth recipe in terms of dependencies because it embraces the need for service objects as a way to handle cross-object workflow. If it doesn't have those downsides, what could possibly be wrong with this approach? The other recipes are all fairly
easy to understand in terms of structure. In the first recipe, the organization of code is almost mechanical. In the fourth recipe the goal is to create richly-functional, self-contained objects with no services or managers being necessary. This recipe sits in the middle ground with no hard and fast rules. That makes it more difficult to teach and, for a lot of people, much more difficult to learn.

My personal preference, of all four approaches, is that second one. It has no artificial structure imposed by dogmatic rules but instead tries to take the most natural "shape" based on the business objects and their workflow, as an organic design. It maintains the separation of the database access code that allows the database to be refactored without changing the business logic. It is amenable to unit testing, partly because of that data layer separation but also because it separates workflow from beans: it is easier to test a self-contained bean than one that has complex dependencies on other beans and it is easier to test workflow in isolation when you can "mock up" beans on which that workflow can operate.

What Are The Ingredients?

I've deliberately glossed over a number of terms in the preceding sections that I will now cover in more depth. So far I've generally used DAO and Gateway to indicate separate concepts and that really dates back to some guidelines that I published in 2003 and updated in 2004, as part of the original Mach-II Development Guidelines[1]. Those guidelines were written when object-oriented programming was still a very new concept to most ColdFusion developers and I was trying to provide simple rules of thumb that would help focus on the object while still providing a "pigeonhole" for CFQUERY style results.

If we look further than the ColdFusion community, we see that although both terms exist, they are actually used almost interchangeably. The term DAO comes primarily from the Core J2EE Design Patterns book[2] and is intended to be an object that provides all forms of data access for a given business object (or sometimes a group of closely-related objects). It's a natural name to give to such an object but in other design pattern literature, this same pattern is called a Data Gateway or Table Data Gateway (Fowler[3]). In this respect, the second recipe above more closely follows the design patterns in common usage outside the ColdFusion community.

Data Access Object is described as an object that encapsulates a "data source" and implements the access mechanism required to work with it. The data source can be any sort of data provider, including a database, LDAP or even business service that might be accessed remotely. The intent is that the Data Access Object provides a simple, consistent interface that completely hides the specific access mechanism. In our common usage, this is usually referred to as CRUD - Create, Read, Update, Delete - and most frequently the data source being encapsulated is a database, so it is the SQL code that is being hidden. In theory, a Data Access Object could encapsulate an LDAP repository or a remote data service and still

provide a CRUD interface. Along with the basic CRUD interface, the Data Access Object typically provides aggregate query operations and sometimes bulk update and delete operations.

Martin Fowler describes a Table Data Gateway as a specific form of the Gateway design pattern that "holds all the SQL for accessing a single table or view: selects, inserts, updates, and deletes." (http://www.martinfowler.com/eaaCatalog/tableDataGateway.html (ibid 3)) A Gateway is a "base pattern" and describes an "object that encapsulates access to an external system or resource." (http://www.martinfowler.com/eaaCatalog/gateway.html (ibid 3)) Fowler goes on to give examples for the Table Data Gateway that look very similar to the Data Access Object mentioned above and, in closing, references the Core J2EE Design Patterns book for further reading. Fowler's Table Data Gateway is a Data Access Object that encapsulates a database table.

As can be seen from the above, "DAO" and "Gateway" is a somewhat artificial distinction to be making in ColdFusion if we want to speak the same language as other computing communities. It will be interesting to see whether this distinction goes away over time - in other words, whether we shift from favoring recipe #1 to recipe #2.

Active Record is another design pattern that is explained in depth in Fowler's book. He describes it as an "object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data." Again, this is focused on encapsulating database access rather than some arbitrary external source of data. Fowler makes the observation that an Active Record is "a Domain Model in which the classes match very closely the record structure of an underlying database." In other words, while the Active Record encapsulates the database access, hiding the SQL, it does not hide the structure of the database.

Fowler goes on to say that Active Record suits domain logic that "isn't too complex" but if your "business logic is complex, you'll want to use your object's direct relationships, collections, inheritance, and so forth," and notes that this "will lead you to use Data Mapper instead." The Data Mapper is a "layer of software that separates the in-memory objects from the database," and is responsible for transferring data between the two while keeping them isolated from each other. This is the core idea behind an Object-Relational Mapping (ORM) tool such as Reactor or Transfer. Fowler recommends using a Data Mapper to allow "the database schema and the object model to evolve independently."

After reading this section, you should have a better understanding of how the different patterns (recipes) relate to each other and some of their strengths and weaknesses. Now we'll take a look at a common pitfall with all these patterns.

Eating Well or Poor Diet?

A term you may have heard, often in disparaging tones, is anemic Domain Model. This is considered an "anti-pattern" - something you should try to avoid in your design. In order to avoid it, you need to know what it is. Many consider it good practice in object-oriented design to create richly-functional business objects that contain data and all of the associated logic for that data. This is your domain model. An "anemic" domain model's business objects really only contain data (and getters and setters) and the business logic is instead in other objects, typically Service or Manager Objects. A symptom of this is code that calls getters on an object to retrieve data, performs some operation on that data, and then calls setters on the same object to update the data. It's likely that the operation should be part of the object that already contains the data, and the code should just ask the object to perform that operation directly on its own data. If your code exhibits the symptoms described above, your beans are probably just acting as data containers and all your business logic is in your Service or Manager Objects. In other words, your domain model is anemic. You need to feed your beans some more logic and turn them into proper business domain objects!

The anemic domain model anti-pattern is easy to fall into for two reasons. The first is simply that most of us in the ColdFusion community have a procedural coding background where our data and our code have historically been separate. If we wrap our data in an object and we wrap our code in another object, we may be taking the first baby steps on the path to 00 but in reality we have an anemic domain model. The second reason can be born out of our choice of recipe. The "5:1 Syndrome" in the first recipe can lead to an anemic domain model, particularly if you auto-generate your code. When you start out with beans that are auto-generated as data containers, it's often easier to just add logic to your service objects and leave the generated beans alone. The same can be true of using an ORM framework. When the framework automatically manages your beans and their basic database access, it's often easier to add your business logic elsewhere. The two most popular ORM frameworks - Reactor and Transfer - both provide ways for you to add your business logic to your beans, where it should be. Reactor auto-generates some empty components that extend the Active Record beans that it also generates. You can add your business logic to these components and Reactor will not overwrite them. Transfer uses the Decorator design pattern to allow you to add your business logic to the beans that it

generates. The main difference is that Reactor expects your business logic components to have particular names and be in the same directory where it generates code, whereas Transfer lets you use any name and location for your Decorator code. Either way, it's still a change of habit for most
people so it requires effort to avoid the anemic domain model.

Real-World Web Applications

Despite all the best intentions regarding design patterns and the goal of cultivating well-fed business objects, the reality of most of the applications we build is that we are often building simple data management applications. These applications contain pages that list record sets in summary form, pages that display the details of a single record and pages that let us edit a single record. There simply isn't much business logic in these applications so our domain model is often going to look somewhat anemic; we have no business logic to feed to our beans!

Even in such applications, we will probably have validation logic and that may be a good candidate to add to our beans in an effort to create the habit of writing business domain objects instead of dumb data containers.

Web applications are evolving all the time and more sophisticated systems are appearing on the web, especially with the advent of richer user interfaces that make it possible to build more complex applications that are still highly usable. This provides us with both the opportunity and the challenge of designing more complicated domain models, comprised of smart objects that collaborate to get their work done.

I hope this article has provided some illumination and guidance on how to meet that challenge and take advantage of the opportunity before you.

References

[1] Web Technology Group Mach-II Development Guidelines - Sean Corfield
http://livedocs.adobe.com/wtg/public/machiidevguide/
[2] Core J2EE Design Patterns - Deepak Alur, John Crupi, Dan Malks
http://java.sun.com/blueprints/corej2eepatterns/
http://www.amazon.com/exec/obidos/ASIN/0130648841
[3] Patterns of Enterprise Application Architecture - Martin Fowler
http://martinfowler.com/eaaCatalog/
http://www.amazon.com/exec/obidos/ASIN/0321127420