Single-Table Inheritance with Transfer

Posted At : January 27, 2009 7:53 PM | Posted By : Bob Silverberg
Related Categories: OO Design, ColdFusion, Transfer

I want to talk about object inheritance, which may sound a bit enigmatic, so I'm going to start with an example:

Let's say I have a number of types of **Employees** that I'm trying to model. I have **Developers**, who have attributes such as *favouriteLanguage* and *enjoysDeathMetal* and behaviours such as *code()* and *test()*. I have **Designers**, who have attributes such as *favouriteColour* and *toleratesDevelopers* and behaviours such as *makePretty()* and *makePrettier()*. And I have **Analysts**, who have attributes such as *levelOfAttentionToDetail* and *totalPagesOfRequirementsProduced* and behaviours such as *ask()* and *tell()*.

In addition to those specific attributes and behaviours, all of these employee types also share some common attributes, such as *userName*, *firstName* and *lastName* and also have common behaviours such as *startDay()*, *takeBreak()* and *askForRaise()*.

This is an example of *inheritance*, a form of one-to-one relationship. We can say that a **Developer** *is an* **Employee**, a **Designer** *is an* **Employee** and an **Analyst** *is an* **Employee**. It is not an example of *composition*. We would not say that a **Developer** *has an* **Employee** or that an **Employee** *has a* **Developer**. The terms that are commonly used to describe this relationship between classes are *Supertype* and *Subtype*. **Employee** is a *Supertype*, while **Developer**, **Designer** and **Analyst** are all *Subtypes* of **Employee**.

So, the question is, how do I implement this model using Transfer?

My first inclination was to attempt Multi-Table Inheritance, which is a fancy way of saying that I'd have an **Employee** table that stores all of the common attributes, and I'd have three additional tables, one for each EmployeeType that stores their specific attributes. For example, I'd have a **Developer** table with columns for *favouriteLanguage* and *enjoysDeathMetal*. Unfortunately, try as I might, I could not get that to work with Transfer without using a OneToMany or a ManyToOne. And I really don't want to use a OneToMany or a ManyToOne to represent what's really a OneToOne. As I've already described, this is an example of inheritance, not composition. So instead I opted to try Single-Table Inheritance.

Single-Table Inheritance means, simply, that there's only one table, the **Employee** table. It contains columns for all of the common attributes **and** columns for all of the specific attributes. Each of the columns that is specific to one EmployeeType allows nulls, so a record for a **Developer** would only have data populated into the *favouriteLanguage* and *enjoysDeathMetal* columns, while the *favouriteColour* and *levelOfAttentionToDetail* columns would contain nulls.

Those of you who are fond of normalization, as I am, will cringe a bit, but really, normalization is a bit passé these days. With the cost of disk space dropping all the time and the speed of processors increasing, denormalization has become downright acceptable. But I digress. What I chose to attempt to implement is this Single Table Inheritance scheme using Transfer. Actually that's not entirely accurate, really Paul Marcotte and I chose to attempt this - we worked together on this solution so these ideas are as much his as they are mine.

Anyway, what would the transfer.xml file for this look like? I'm going to start with a simple example, which is not ideal, and then add to it to address its inherent problem:

```
<package name="employee">

 <object name="Developer" table="tblEmployee" decorator="Developer">

  <id name="userName" type="string" />

  <property name="firstName" type="string" />

  <property name="lastName" type="string" />

  <property name="FavouriteLanguage" type="string" />

  <property name="enjoysDeathMetal" type="boolean" />

 </object>

 <object name="Designer" table="tblEmployee" decorator="Designer">

  <id name="userName" type="string" />

  <property name="firstName" type="string" />

  <property name="lastName" type="string" />

  <property name="FavouriteColour" type="string" />

  <property name="toleratesDevelopers" type="boolean" />

 </object>

 <object name="Analyst" table="tblEmployee" decorator="Analyst">

  <id name="userName" type="string" />

  <property name="firstName" type="string" />

  <property name="lastName" type="string" />

  <property name="levelOfAttentionToDetail" type="string" />

  <property name="totalPagesOfRequirementsProduced" type="numeric" />

 </object>

</package>
```

What I've done is define three separate objects to Transfer, each of which points to the same table. For each object I only define those properties that are valid to that object. For example, the **Developer** object has a *FavouriteLanguage* property but not a *FavouriteColour* property. In terms of behaviours, I create an Employee.cfc decorator (the Supertype) that contains all of my common methods, e.g., *askForRaise()*, and I create decorators for each of the employee types (Subtypes) which extend the Supertype's decorator. For example, Developer.cfc extends Employee.cfc and it includes the method *code()*. This all works quite well, but there's a problem.

The problem is how to protect the integrity of the Subtypes. What do I mean by that? Here's an example:

Let's say there's a **Developer** named *Bob* in the system. I could say:

```
objDeveloper = Transfer.get("employee.Developer","Bob");
```

And I'd get back a **Developer** object for Bob. I could then call *getFavouriteLanguage()* and *askForRaise()* and *code()* on that object. Fine. But what if I did this:

```
objDesigner = Transfer.get("employee.Designer","Bob");
```

Hmm, now I have a **Designer** object, but it's not valid because really the employee I'm dealing with is a developer. I can no longer call *getFavouriteLanguage()* nor *code()* on the object, but I can call *getFavouriteColour()* and *makePretty()* on the object, which would probably end up badly, considering that Bob is really a developer, not a designer. So this is a problem. But there's a solution. Enter **EmployeeType**.

First we create an object for **EmployeeType**, and then we create a ManyToOne between each of our Subtypes and EmployeeType. For example, **Developer** has a ManyToOne pointing to **EmployeeType**. Finally, we change the Subtypes to use a *composite key* instead of a single *id*. So the key to **Developer**, for example, is no longer just *userName*. It's now *userName* **plus** *EmployeeType*. Now, whenever we want to ask for an instance of an employee, we will specify both the *userName* and the *EmployeeType*. This "specifying the EmployeeType" will happen automatically in the Gateway - we won't have to do it manually. To take a look at this implementation let's start with the transfer.xml:

```
<package name="employee">
 <object name="Developer" table="tblEmployee" decorator="Developer">
  <compositeid>
   <property name="userName" />
   <manytoone name="EmployeeType" />
  </compositeid>
  <property name="userName" type="string" />
  <property name="firstName" type="string" />
  <property name="lastName" type="string" />
  <property name="FavouriteLanguage" type="string" />
  <property name="enjoysDeathMetal" type="boolean" />
  <manytoone name="EmployeeType">
   <link to="employee.EmployeeType" column="employeeTypeId"/>
  </manytoone>
 </object>
 <object name="Designer" table="tblEmployee" decorator="Designer">
  <compositeid>
   <property name="userName" />
   <manytoone name="EmployeeType" />
  </compositeid>
  <property name="userName" type="string" />
  <property name="firstName" type="string" />
  <property name="lastName" type="string" />
  <property name="FavouriteColour" type="string" />
  <property name="toleratesDevelopers" type="boolean" />
  <manytoone name="EmployeeType">
   <link to="employee.EmployeeType" column="employeeTypeId"/>
  </manytoone>
 </object>
 <object name="Analyst" table="tblEmployee" decorator="Analyst">
  <compositeid>
   <property name="userName" />
   <manytoone name="EmployeeType" />
  </compositeid>
  <property name="userName" type="string" />
  <property name="firstName" type="string" />
  <property name="lastName" type="string" />
  <property name="levelOfAttentionToDetail" type="string" />
  <property name="totalPagesOfRequirementsProduced" type="numeric" />
  <manytoone name="EmployeeType">
   <link to="employee.EmployeeType" column="employeeTypeId"/>
  </manytoone>
 </object>
 <object name="EmployeeType" table="tblEmployeeType">
  <id name="employeeTypeId" type="numeric" />
  <property name="name" type="string" />
  <property name="description" type="string" />
 </object>
</package>
```

This transfer.xml just implements everything discussed in the previous paragraph. It should be fairly straightforward. So, how do we use it? Well, all of our calls to Transfer.get() are centralized in a *Gateway*, thereby encapsulating database access. So all we have to do is something like this, in our Gateway code:

```
<cffunction name="get" access="public" returntype="any">
 <cfargument name="userName" type="any" required="true">
 <cfset var theKey = StructNew() />
 <cfset theKey.userName = arguments.userName />
 <cfset theKey.employeeType = getEmployeeTypeId("Developer") />
 <cfreturn getTransfer().get("employee.Developer",theKey) />
</cffunction>

<cffunction name="getEmployeeTypeId" access="private" returntype="any">
 <cfargument name="EmployeeType" type="any" required="true">
 <cfreturn getTransfer()
  .readByProperty("employee.EmployeeType","Name",arguments.EmployeeType)
  .getEmployeeTypeId() />
</cffunction>
```

What you see above is a concrete example of how it actually works, (that code would reside in DeveloperGateway.cfc) but my actual code is quite different from the above because it's based on abstract classes. I don't want to complicate things here by getting into the details, but basically I have an**EmployeeGateway** which contains parameterized code which is then inherited by the **DeveloperGateway**, **DesignerGateway** and **AnalystGateway**, none of which have any code in them at all. So all that hardcoded stuff that points to "Developer" is nonexistent in my code.

The bottom line is that I can call a *getDeveloper()* method or a *getDesigner()* method from my Service, passing only the *userName*, and be assured that I'll always get a valid object back.

This seems like a pretty neat solution to the problem, but so far it's only been used in theoretical situations. Can anyone see any problems with it that we haven't thought of?