OO Principles: Composition (part 2)

Posted At : September 1, 2009 10:45 AM | Posted By : Steve
Related Categories: ColdFusion

I don't "do OO" development in ColdFusion. I'm starting with that statement not to spark another debate about whether to use OO in ColdFusion, but rather to clarify that while this post is about a principle of object oriented development, you don't need to "Do OO" in order to learn, use, and benefit from composition.

In the last "**OO Principles**" entry, I **introduced composition**. The examples that I used were barely complicated enough to show some of the benefits of composition. Much more complicated than that, however, and you can also run into some challenges. While these do not (in my opinion) overcome the advantages of composition, they are still worth considering.

The two main challenges are large dependency chains and circular dependencies.

Large Dependency Chains

The most complicated example that I used in my previous entry had three components with the first being passed to the second and the second being passed to the third. Even using just a service based model (instead of the complexity of a full object oriented model) the number of components in use on a real world application can grow far beyond this, as can the number of dependencies that you need to keep track of.

For this, a dependency injection engine (or DI Engine) can be really helpful. In ColdFusion, the two most popular options here are **ColdSpring** and **LightWire**. **Brian Rinaldi** recently wrote and excellent **Beginner's Guide to the ColdSpring Framework for ColdFusion**. It is so good, in fact, that my covering any more about the problems and solutions of dependency injection just seems redundant.

I would only add that, while ColdSpring is far and away the most popular option, keep in mind that it is not the only choice. The concepts that Brian covers would apply to most DI Engines.

Circular Dependencies

If you discover that your Component A needs to know about Component B, but Component B also needs to know about Component A then you have a circular dependency. You cannot have each pass the other into the init method as one will have to exist before the other.

I have found that most circular dependencies can be avoided (by have a third component compose both, for example). Those that can't will have to be dealt with outside of simply passing the components into the init method.

Some developers compose components by way of "setter' methods:

```
<cffunction name="init" access="public" returntype="any" output="no">
<cfargument name="DataMgr" type="any" required="yes">
<cfargument name="Organizations" type="any" required="yes">

<cfreturn This>
</cffunction>

<cffunction name="setOrganizations" access="public" returntype="void" output="no">
<cfargument name="Organizations" type="any" required="yes">

<cfset variables.Organizations = arguments.Organizations>

</cffunction>
```

```
<cfset Application.DataMgr = CreateObject("component","DataMgr").init(request.dsn)>
<cfset Application.Organizations = CreateObject("component","Organizations").init(Application.DataMgr)>
<cfset Application.Users = CreateObject("component","Users").init(Application.DataMgr,Application.Organizations)>
```

Note that most ColdSpring (and probably other DI Engines as well) support both constructor injection (via the "init" method) or setter injection (as shown above).

Using composition in this way can open up new challenges (such as circular dependencies) as well as new opportunities for organization and automation (using a Dependency Injection Engine, for example). If you are using components, however, I would highly recommend using composition over breaking encapsulation.