2013-02-28

Today I went deeply into implementing Packages for Fresh. I came across a set of collisions that helped me to see more deeply into the system than I believe I ever have.

At the philosophical root of Fresh are a few key concepts.

* Object
* SmartPtr
* Manifest

Believe it or not, that’s really it.

The first two are determinative. Who owns an object? The truth is that SmartPtrs own everything. SmartPtrs themselves are generally owned by Objects. Therefore, Objects own themselves. Objects and SmartPtrs create a loose, ad hoc, decentralized graph that is at the heart of Fresh’s philosophy.

Seen in this light, a manifest is really just a file on disk that “inspires” the creation and configuration of a set of objects. It isn’t really a container for objects—only a container for “inspirations”—and it certainly doesn’t own them. Only Objects own other objects.

So there is no centralization of ownership in Fresh.

But then, you can simulate centralization of ownership depending on how you arrange the graph. A Stage, or any subtree within a Stage, has centralized ownership in the sense that the death of the ancestor spells the (likely) death of all the descendants. So centralization isn’t *anathema* to Fresh as a use case. But it *is* anathema to the core architecture on its own rights.

The problem with ObjectManager, with Namespace, and with Package is all the same: they all attempt to “own” objects, when objects really can’t be owned in that sense. The idea in prior versions of Fresh that ObjectManager “owned” all objects, such that they *had* to be deleted through ObjectManager::deleteObject(), was simply wrong. Objects might as well have called “delete this”.

So what is the ObjectManager/Namespace/Package concept trying to achieve? It’s trying to achieve exactly two desired features, which *are* important. I thought that Package could bring these two features together, but the problem of coordinating Package names and manifest paths—an apparently minor problem—showed me that there’s a discontinuity at the heart of that strategy.

The desired features are:

1. Be able to search for objects by name (but not all objects—“unmanaged” objects are cheaper).
2. Be able to “group” or “collect” objects in ways that span their pointer graph relationships, but which enable you to treat collected objects as a group.

The use case for the second feature is stage loading and saving. The objects that are “inspired” from a manifest file lump into the object graph just like anything else does. And yet when you go to “save the stage”, you have to know which objects belong with the stage and which (like Assets, Renderer) do not.

So we come to this framework for a solution. Fundamentally, objects are objects. They point to each other. They are created and die in arbitrary ways. They are a big disorganized soup.

But on the other hand, you can search for objects. And you can lump objects together.

We must find a solution that captures these ideas as simply and unobtrusively as possible. As we look for one, what I have to keep foremost in my mind is that nothing really *contains* objects. Objects own themselves. Nothing else can own them. At most, other things can *organize* them.

# Tags and Manifests

In the car today I realized that Package is the exact reverse of what it should be. Instead of a “container/owner” that sits below objects and encompasses them, what we really want is a mere “tag” object that in some sense hovers “above” objects, and simply labels a set of objects as somehow belonging together.

## Tags

This isn’t a finalized solution—I still want to talk about Manifests—but to consider how this would be implemented, in essence a “tag” is simply an object with a list of WeakPtrs. (Each Object might, perhaps, also have a SmartPtr to its tag [or a list of SmartPtrs to all its tags.]) You can search a tag for a given object. You can save all the objects “with” a given tag (or set of tags?) to a manifest. You can tag all of the objects emerging from the loading of a manifest with a tag. A tag doesn’t own its objects—quite the reverse: when no objects are using a tag, it dies.

This solution, on its own, solves both of the two desired features. It allows you to complicate (i.e. “namespace”) object names by requiring a certain tag name:

<pTexture>Texture’alien#system\_tag’</pTexture>

In the same vein, it allows you to search a particular tag for an object. (TODO Can untagged objects be searched at all? What about tags themselves?)

getTag( “editor” )->getObject< Manipulator >( “main\_manipulator” );

And you can save objects, limiting the allowed tags:

saveManifest( “Documents/myStages/stage01.xml”, rootObject, getTag( “user\_stage” ));

Beautiful. And yet. Something even simpler might the same or better. To see this possibility, let’s look more closely at the concept of a Manifest.

## Manifests

The essential idea of a manifest is fundamental. A manifest is an XML file on disk (usually) that contains a set of objects, “inspiring” them to be created and initialized. This idea has proven extremely robust and valuable, and it’s now one of the cornerstones of Fresh.

One of the longest ongoing challenges with Manifest loading, however, is the question of how to retain loaded objects. There are two broad options: (1) return a root node, which contains everything else; and (2) retain everything until the user wants to release them.

Neither option is particularly happy. The notion of a root node implies a structure that is not otherwise required by the manifest concept. Retain all the objects is perhaps more conceptually pure, but creates an ambiguity concerning what the loading of objects means in the context of the rest of the graph. Retaining them until an explicit “release” call makes good sense, and leaves the question of what should actually be retained until after the user code has time to interact with the loaded objects. But it’s not a terribly elegant solution.

A very promising adjustment to this idea came to me today. What if a manifest is itself an object, and in effect it is the “root node” I mentioned previously. It has a vector of SmartPtrs to other objects, and you can fill this vector freely. When you save the manifest, you are in a sense just saving one object—but of course this naturally saves the array of objects that it contains. (You still need something like the “tag” concept in order to either fill or filter this array.) When you load it, it of course retains those objects—to release them you simply kill the Manifest.

Note that Manifests do not reference objects in an exclusive way or pretend to own them except in a momentary sense. You could have more than one Manifest reference the same object. A Manifest is meant to come and go. It’s an Object like any other, except that it has a load() and save() function taking a file path and some search facilities on its members. There is really very little additional functionality here, and virtually no new philosophical concepts.

## Tags vs. Manifests

What I now observe is that manifests and tags are virtually identical, and yet not the same. In essence:

* Both manifests and tags are Objects.
* Both have a container of references to other Objects (“members”).
* Both offer search facilities to find “member” objects.
* Manifests reference objects strongly, whereas tags reference objects weakly.
* Manifests are meant to live for a short time (e.g. just as long as a load or save operation), whereas tags are meant to persist (e.g. bridging a load and save operation).
* Manifests emphasize file serialization, whereas tags don’t care about that.
* You can search for tags by name as a system concept (getTag()), and this functions as the fundamental object search mechanism. Manifests don’t live long enough to make sense of that.

Could it be that tags and manifests are really the same concept, with the one difference being that tags hold objects weakly while manifests hold them strongly? If so, then what if we bring these two concepts into a single concept, where the “Package” (I’ll call it) can reference its members either weakly or strongly, modally. When loading or saving, the package holds its members strongly. Then you can “release” the package, causing it to convert those references to weak.

# Packages

I seem to have come full circle. It does make sense, I think, having divorced the concepts of “tagging” and “save/load manifests”, to now marry them again into a single concept. If we went down the two-class concept, the chief question would be how to feed the members of a tag over into a manifest. Manifests would be irritatingly short-lived, or else would dominate the object lifespan system. Packages have the advantages of both schemes.

The remaining questions:

* How do package names relate to paths? Do they? (A: Probably not at all)
* How are packages retained and destroyed? (A: Like any other object. But they might have a special “I’m empty and unreferenced by anything but my manager so kill me” feature something like Assets.)
* How are packages found? If you search for an object by using a package, how do you search for package objects? Is there a system Package-of-Packages that holds all (other) packages?
* Do Objects need pointers to the package(s) that reference(s) it? (A: I’m hoping not.)
* How does package membership affect an object’s ID? Can’t an object live in more than one package, and if so, how is the ID affected?