Massive Refactor – Questions

# Package Object Ownership

Packages have an ambiguous relationship with objects.

In one sense a package owns an object. Only packages create objects, and only packages delete them. An object’s package keeps the one unique\_ptr to the object. *That’s* ownership.

And yet this is far from the whole story. The SmartPtr system, being layered over the Package system, really has a stronger sense of ownership than the Package system does. That is, they both own the Object in some sense, with the Package ownership being “lower level” than SmartPtr.

This is really not such a huge surprise. unique\_ptr is really just my latest implementation of what was previously raw new and deletes in ObjectManager.

The real conundrum comes when you consider that Packages are actually *allowed to die when their objects still exist*, at least theoretically. Actually the relationship is more complicated than that: a package lives just so long as it holds any living objects *or* (non exclusive) there is a pointer to the package.

This means that although Packages hold objects, objects actually hold Packages too—not really in a cyclical relationship, but in a conceptually difficult one.

The sharp end of this particular stick (or one of them) comes where you consider how to load a package and then access the objects it contains. If the package doesn’t really hold the objects (in the sense of “keep them alive always”), then the objects are liable to die before the loading function is even done. (In the manifest system the solution was to pass back a sole “root object” that was kept alive by the user (hopefully) and kept any non-transient objects alive.)

Should the package, then, keep its objects alive? I mean, in an eternal, ownership kind of way? If so, this would mean that objects would only die when their package was closed—no sooner, no later.

Surely that’s not what we want. The “assets” package, if it were one package, would have to keep all the assets alive forever. Once you saw an asset, it would be in memory for the rest of the game. So what would you do? Segment assets into smaller packages that can be loaded and dismissed as a set of objects? But this removes all automation: now you have to know precisely what objects should be alive or dead at any point. Basically you’ve eliminated SmartPtr in favor of a “block of objects” approach to allocation and deallocation. That sounds like the worst of all worlds.

So no, Packages shouldn’t keep objects alive.

But this raises the opposite question: why should packages hold onto anything? Why not just let SmartPtrs delete objects directly, much like shared\_ptr does? Then a package keeps WeakPtrs to its objects, merely organizing them for searching purposes.

And I suppose that’s the point. Packages organize objects, helping them to load together from a common location and with common fixup, or save together to a common location, or to be searchable. They’re not really a memory management feature. If packages take care of memory, it’s really just an accident of convenience.

So okay, I’ve returned to a degree of comfort with the current unique\_ptr solution.

But what about the outstanding issue? How do you load a package such that the loaded objects (some or all) are retained by the caller?

I see two broad strategies in terms of volume. Either the load function returns a root that is responsible for the rest of the objects. Or it returns a vector of all the objects in the package.

Another axis of questioning is whether the return value is really the right place for “release” of the objects. We could have a separate function that releases them. But no, that really seems arbitrary and error-prone.

So a return value. And there’s the question: is it a single root object or a vector?

A root object solution isn’t as limiting as it sounds. You can always create an object that amounts to a vector of pointers to other objects. But is there some disadvantage to just passing back a vector?

It certainly involves more responsibility on the user side. The user has to search it, for example. But that shouldn’t be hard considering that the Package is there to support the search. Hm.

I’m not a big fan of returning 200 object pointers in a vector, especially if the user’s typically pattern will be to retain one.

Okay, but if I don’t like the vector, the root has problems too. What determines the root? Does the package itself know the root? Does the load process request a root (as in the old system)? Does the save process indicate a root which is written to the manifest and then recalled during load (as in the newer old system)?

No, I’m thinking the vector approach is worth experimenting with.

Alas, questions like this so often come down to limitations on the number of return values from a function and such. Current the syntax, quite sensibly, is:

pPackage = openPackage( fullPath );

So what do you do?

pPackage = openPackage( fullPath, /\* out \*/ loadedObjects );

// OR

loadedObjects = loadPackage( fullPath ); // With the package pointer implied?

// OR

pPackage = openPackage( fullPath );

root = pPackage->findObject( rootId );

pPackage->releaseUnreferencedObjects();

This last one is actually starting to grow on me. I think I’d better sleep on this question, though.

Another paradigm is actually a bit scary. Maybe we don’t have openPackage() at all. Maybe instead the idea is:

pObject = loadObjectWithPackage( fullPath );

And loadObjectWithPackage() just “happens” to load a package, though the focus is on the object fetched from the package. That’s interesting. It definitely puts an emphasis on the root object concept. The object isn’t just a root anymore: it’s the whole point of the load. And that does make some sense.

# How Do Packages Relate to Paths?

On the one hand we need packages to have short, general names, such as “level1” in the following:

Sprite’level1.gypsy’

That’s as opposed to:

Sprite’/User/billy/Documents/ElectricToy/CTC3/Assets/level1.gypsy’

Right? This is important both for readability and for generality—allowing files to be moved and not to invade the object space of the game too much.

But on the other hand, packages can of course be loaded from an absolute path and saved there too.

How do package names relate to package paths?

Note particularly the tension involving a package that the user created, which is untitled. If we save it, and in saving it, we rename it, then won’t other objects in the system get confused about where those objects are? Maybe not? I’m not sure. Anyway I need to decide how package paths and names correspond. Maybe they don’t? I need stronger use cases.

# How to Hide Construction/Deletion from all but those who should do it

This a recurring theme for various parts of the engine: objects, classes, factories, default objects, and packages. There is a particular path by which these various things are meant to be created. How do I mark their constructors and destructors as private and friend other things such that only the correct things can create or destroy what is supposed to be created and destroyed?

# “Loose”, Temp, and Untitled Packages

Here we have three concepts that are connected but, I think, distinct.

An untitled package is one that that user has created; we don’t know what the name is nor what file it will end up saving into; but we intend, probably, to save it eventually, at which point it will gain a real name.

A temp package is one that may or may not have a name, but it is never intended to be loaded or saved. It’s an in-memory-only package.

The “loose” (or “default”—though I tremble at the word) package is the one that is used by createObject<> functions that don’t specify a package. Should this even exist? Is it a mere bastion of laziness and chaos, or does it have some useful role? I’m already using createObject<> in lots of places, so I need to decide this soon.

# Root object = Root element?

This morning I had the thought that perhaps a manifest’s <objects> element should be replaced by the root object itself, such that the object’s sub-objects are intrinsically linked to the object, for example via the “children” node for a DisplayObjectContainer. Loading a package would truly be loading that one object—with the other objects coming along for the ride. Hm. But this would really make a “package” into a mere sponsor for an object. Not that that’s necessarily wrong. But I’m not persuaded.