Notes on Fresh Object Naming

ObjectManager enforces the rule that every object of a given class has a unique name. This rule is needed in order to provide correct functionality for ObjectManager::GetObject(). Without it, objects of a given class that share a common name (including the “NULL” name) would be hidden—only the first object with that name could be found through this function.

There is no problem with this rule intrinsically. But there are two issues with the object naming system that I should address. First, there is the potential (likelihood) of name collision when loading objects from XML files. Second, the current system for generating default names is expensive both in space and time.

# Name Collision when Loading

The problem that can currently occur could happen like this.

An object “instance100” gets created. It is saved to an XML file and that name is recorded.

The application is restarted. Another object of the same class--possibly the same object in some sense--gets created with the same name. Now the XML file generated during the first run is loaded. During the load, an object named “instance100” is loaded from the file. We now have a name collision. This error will be detected and reported during the call to CreateObject(), because that function requires !HasObject().

The problem can also happen in the opposite direction, and here it is more insidious. After the first run, the application is run again but *no* object of name “instance100” is created. Now the XML file is loaded, so “instance100” is now created. But now as the program continues to execute, eventually an default-named object of name “instance100” will be generated. (See Object::Object()’s facility for generating object names based on the number of objects created during this run of the application.) This error will not be detected, because RegisterObject() does not currently check for name uniqueness.

Which raises a third error case not directly connected with loading. If a user deliberately makes an object named identically to another object, but uses “new” rather than CreateObject(), a collision will occur. This is not necessarily a big problem because the only failure is that GetObject() or GetObjectIter() will no longer find that object—they will find the earlier-registered object..

The second and third problems point to a partial solution of including name checking in RegisterObject(). This is trouble for performance, but only in Debug mode.

The first problem is deeper. It points to the need for something like namespaces or packages. A possible solution would be to add a “namespace” string member to Object, which is null by default. (A chain of namespaces would allow nesting, but for now let’s assume a singlet namespace.) By default an object has the null namespace. But when an object is loaded from file, it is given a namespace named after that file. This would prevent the first and second problems above, and makes fairly good philosophical sense. In a way, it honors the implication that is assumed by the current default naming system that s\_nObjectsCreated is the last word in how many objects there are. s\_nObjectsCreated *is* the number of objects created—in the default namespace, that is.

The full implications of the namespace system are unclear, however. For example, if an object within a namespace (that is, within an XML file) references another object, it would presumably do so without explicitly indicating the namespace. Resolving the object name is no problem because we know where the object came from. But if an object references an object in another namespace, how does it find it. Another way to see the same problem: when objects are saved in an XML file, does their namespace change immediately? If so, then how do other objects know about it? If not, then why is it that merely loading an XML file can change an object’s namespace? There just needs to be more thought given here.

# Expensive Default Names

By default, Object constructors use the “default name”, which is simply the empty string. But since every object must have a unique name, the empty string is not retained as the object’s name. Rather, a (hopefully) unique name is generated in the Object constructor.

This unique name is pretty expensive both in time and space, and this expense feels unnecessary. Yet removing the expense is surprisingly and annoyingly non-trivial.

The expense of the default name generation in time is that it requires a stringstream to concatenate two strings and to convert a number to a string. This isn’t rocket science but it isn’t fast either.

The expense of the default name generation in space is that it occupies heap space with a string that is at least 9 or 10 characters long (not counting additional heap overhead).

Default names are also a bit more expensive than they need to be for string comparisons, because the first 9 or 10 characters are uniform (“instance….”). A more jumbled name would give better performance for object searches.

So the current default name generation scheme is annoyingly wasteful and basically brings no benefits.

The simple (but not-quite-workable) solution would be to store an ordinal number in every object giving that object a unique numeric identifier (at least within the null namespace). Default-named objects would store no string name, but when asked their name they would generate one from this ordinal.

The problem with this solution is twofold. First, it is probably slower rather than faster. Second, generating and then returning a string gives you the usual painful tradeoff between returning a string by value, thus causing a copy, or passing it as a non-const reference argument, which makes for ugly syntax in the caller. Memory usage is much better in this case, but it’s still a clunky solution.

One possible help is to change some current calls to Object::GetName() into calls to Object::HasName(). This allows the object itself to check the incoming name to see if it *would* match it if it had a name. This *is* a prudent solution, but not all calls to GetName() can be eliminated.

A middle-ground solution might be to mark m\_name as mutable, start it as null, and then generate and store the name only as needed—basically, lazy-generate the name. The problem with this solution in this context is that Fresh *does* call GetName() (or at least HasName()) for virtually every object fairly often, so no object would be allowed to be lazy for long.

Incidentally, having GetName() return a non-reference has caused performance problems in the past. We really do have to be careful.

Another interesting solution would be to generate names as GUIDs. This would help with any uniqueness issues, including those I discussed in the section above. GUIDs are potentially smaller to store than string names. And comparing GUIDs will generally be very fast.