

## Proposed Changes for the OMF

This is basically a list of changes that i'm going to cram into the existing OMF implementation to improve certain aspects of the implementation. The main reason for all this is to really boost the level of extensibility and reduce build/link time for developers using the C++ toolkit.

### ***Libraries instead of Library***

I'm going to completely redesign the class allocation for libraries and how they're deployed. As it stands right now, the entire thing is crammed into a single .dll that seems to be about 80 MB or so in size. It takes forever (~2 minutes on a decent PC) to link a C++ program against it and that's just not right. I'm proposing two modifications to the scheme to get this right:

First, the OMF core and MOF (Model) implementations will be merged into the same library. It seems that a tighter coupling between the MOF and the library itself is actually a good thing. Besides, by moving the MOF into the OMF core, I'll be able to actually build a fairly comprehensive set of capabilities into the MOF that enables the construction of different metamodel "services" within the OMF. More on this later.

Second, I'm going to completely redefine how code is generated and linked. For starters, we're going to consider the organization of UML. UML is actually divided into several top-level packages that are all imported by the UML package (hence everything eventually ends up in the UML namespace). Fortunately, there are no nested packages in UML to worry about (there are in CWM, however). Ideally, I want each package to be its own DLL responsible for loading some subsection of a model. Ideally...

Now there may be a small problem with some of the programmatic stuff. A long, long time ago, I noticed that the CWM used some of the same identifiers as the UML. In terms of different libraries being loaded, this is going to result in linker redundancies and will prevent the loading of libraries. One solution would be to wrap all entities defined within a metamodel into the namespace of the metamodel itself. This is actually kind of a good idea. Here's where it starts to suck. With this scheme, the outermost UML package (the one that imports the rest) is now officially UML::UML. That's pretty annoying. However, we can kludge something together that imports the inner namespace. Probably something like this:

```
namespace UML
{
    using namespace UML::UML
}
```

There are, of course, other options for doing this – specially named namespaces, namespace assignments and the like. All in all, it will work, but for the time being, I'm not going to worry about it. I really need to check the CWM and some other metamodels to see how they work.

One with the gory details. Each package gets its own library. Enough said. However,

none of these libraries are going to be implicitly linked against any others (no linker dependencies). Instead, one of the new services provided by the OMF is going to be that of library loading and package resolution. When each library is loaded, a special methods embedded within the library. This will return singleton instances of packages.

## ***No More Per-Model Extents***

I took a lot of time to get the idea of having different extents for different models in memory just right, but in the end, it may have turned to be useless and certainly more difficult to manage. The new scheme gets rid of this altogether. The idea is that we want to be able to query and operate across models and this per-model abstraction really hurts us in that affect. So instead, I'm going to cram all of our extent information from different models into the same extents. It turns out that querying on a per-model basis is a simple filter on top of the original query so we don't have to lose any sleep over that.

Moreover, the construction chain for model elements absolutely requires the existence of a model for creation. This makes it exceedingly difficult to work with so-called model fragments. It should be possible to have a lightweight in-memory chunk of a model (e.g., a single class with some attributes and methods) that isn't tied to an existng model, but simply free-floating in the system. The ability to work with disconnected model fragments might lead to some interesting applications.

## ***A New Type System***

Man have I hated the type system that I wrote. It's really awkward. Fortunately, while experimenting with implementing UML2, I wrote a completely new type system that does a pretty good job of covering the mistakes of the old one. For example, containers are truly dynamic this time – no more weird mix of dynamic and template containers. Moreover, I've completely stripped the reflective layer from them and they work quite well with native C++ types. I think you'll like it.

## ***No More Qt***

I'm dumping Qt and moving on to bigger and better things. Namely, the STL, Boost and, when required, specific optimized libraries for doing stuff like libXML. That's about all I'm going to say on this topic.

## ***XMI Upgrades***

I've got lots of improvements to the XMI and persistence layers slated.

## ***Objects on Demand***

Originally, I wrote the XMI persistence as a service in the reflective layer. This actually turned out to be a pretty good idea. However, it was ill-abstracted, bulky, and pretty slow. This time, I'm going to build the abstraction a little better (for other than XMI persistence, per se), but more as an optimization reason. When I was building a reverse engineering tool I noticed that i had a memory footprint of ~90 MB from parsing XML

and creating UML elements. This isn't quite acceptable. Part of this, I know comes from having the entire UML and MOF metamodels in memory at the same time.

Here's what I'm thinking. Objects need to be either created or requested from a model/metamodel. My new idea is to only create the objects (and give them ids of course), but to completely avoid configuring the objects. That is to say, we don't populate any attributes or references with values. This leaves the model exceedingly lightweight and pretty much just a collection of objects mapped (conveniently) to id's. However, when the attributes of an object are requested, we will probably need to configure the attributes and references of the object.

Unfortunately, this does cause some problems. Consider the case where the reflective layer is used to immediately query links between all objects in the model. This isn't quite the best thing to have happen because the associate extents can't be populated until a later point in time. However, there may be some heuristic optimizations that let us build the relation structures without actually fully configuring the model.

## **Graceful Imports**

I've never liked my import mechanism. It's kind of clumsy. I'm going to completely re-examine how XMI imports work and build a better implementation.

## **Differencing**

I never did implement the XMI differencing stuff. I need to design and implement this. Theoretically, it isn't actually all that complicated, but there may be some odd stuff there. This is going to require some pretty serious analysis.

## ***Other OMF Services***

Rather than embed some of the services within the class hierarchy (notably reflection), I'm going to build them as separate services that can be used as requested by the user.

## ***Reflection***

Rather than ingraing reflection into the class hierarchy (as I just mentioned), the reflective service will be just that. A set of decoupled interfaces that provide reflective introspection (not particularly modification) to the metamodels. This is going to entail a couple different aspects. First, some of the generic get/set stuff is going to have to be added to the hierarchy as a general capability of a model object. This is one of the main features provided by the Ref::RefFeatured class – that's going to go away and the get/set/add/remove/clear abstraction will be directly embedded into the base-most model element class and will operate directly on the memory model.

Secondly, we're going to analyze some of the more modern concepts of reflection and build our model that way. A lot of research on source code reflection discusses the concept as source code reflection repositories. This is actually what we have. For example, if we want to get the reflective information about a class, we're going to have to

go the XMI file to find out what it's all about. In short, we don't have to load the XMI file until the user wants to get some reflective information about that metamodel. This is the tie-in to the persistence stuff.

I think the basic design for this should be pretty easy. Since all the model elements are going to be derived from a fairly simple class that offers generic get/set methods and dynamic method invocation, the reflective API can simply be wrappers around instances of those classes.