1 General

1.1 Introduction

One of the main advantages of the object model implemented in Voyager is the release to release binary compatibility. This means that there is no need to recompile any applications or libraries if there is a new version of a class library in the system. Classes packaged in dynamically linked libraries may be improved and extended without breaking the interface.

In addition it's possible to extend and even replace existant classes without having access to the source code. Only header files are necessary for creating a new subclass. So binary only releases of classes or even whole libraries are possible without fearing to be lost in the future if the maintainer decides not to improve the lib anymore. Future updates of the object model or other classes will not render the unmaintained classes useless because they still work.

The design of the object model is language neutral. So it's possible to write classes in any language which has bindings for the model and use these classes from any language.

2 Implementation

When an application is using the object model the following steps are usually performed:

- Initializing of the object subsystem using `nomTkInit()`.
- Creating the global object NOMClassMgrObject using `nomEnvironmentNew()`.
- Creating the desired objects and working with them.

Most of the magic happens in `nomEnvironmentNew()`. This function creates the three basic object classes NOMObject, NOMClass and NOMClassMgr which are the core of the object system.

2.1 nomTkInit()

This function initializes the basic structures for housekeeping. For example the semaphor is created which protects the central lists in a multithreaded environment.

2.2 nomEnvironmentNew()

Main purpose of nomEnvironmentNew is to create the three central classes and the NOMClassMgrObject.

2.2.1 Creating the NOMObject classobject

First the NOMObject classobject is created using the following call:

```
nomObj=NOMObjectNewClass(NOMObject_MajorVersion, NOMObject_MinorVersion);
```

`NOMObjectNewClass()` is a function implemented in the header file *.ih created by the object compiler from the IDL file defining NOMObject. Yes, that's correct this function is not found in the *.c file but in the *.ih file which is included by the implementation file *.c.

```
/*
 * Class Creation and Initialization
 */
NOMClass * SOMLINK NOMObjectNewClass (gulong ulMajor,
            gulong ulMinor)
{
    NOMClass *result;

    result=nomBuildClass(1, &NOMObjectSCI, ulMajor, ulMinor);
    return result;
}
```

In `nomBuildClass()` the classobject is actually created. This will be described later.

The *.ih file also contains static global structures which define essential object data. For this reason the *.ih file must only be included by the class implementation file. If you include the file twice in different source files you end up with two different structures (because of the static keyword) and you will experience bugs which are difficult to hunt down.

> Only include the *.ih file in the class implementation source file. Never ever include it in addition into another source file.

2.2.2 Creating the NOMClass classobject

After the NOMObject, which is the base class of all other classes in the object model, NOMClass is created.

```
nomCls=NOMClassNewClass(NOMClass_MajorVersion, NOMClass_MinorVersion);
```

NOMClass is the base class for classobjects. As before this call translates into a call to nomBuildClass.

```
/*
 * Class Creation and Initialization
 */
NOMClass * NOMLINK NOMClassNewClass (gulong ulMajor,
            gulong ulMinor)
{
    NOMClass *result;
    NOMObjectNewClass(NOMObject_MajorVersion,NOMObject_MinorVersion);

    result=NomBuildClass(1, &NOMClassSCI, ulMajor, ulMinor);
    return result;
}
```

One thing is to notice here. Before actually calling `nomBuildClass()` there is a call to `NOMObjectNewClass()` which will build the NOMObject classobject as shown before. In fact the explicit building of NOMObject is not necessary because when creating a class the class creation procedure will automatically build all parent classes if not already done. NOMClass is a subclass of NOMObject so NOMObject would be created automatically.

The explicit call is there for clarity of the control flow in the source.

2.2.3 Creating the NOMClassMgrObject

A call to

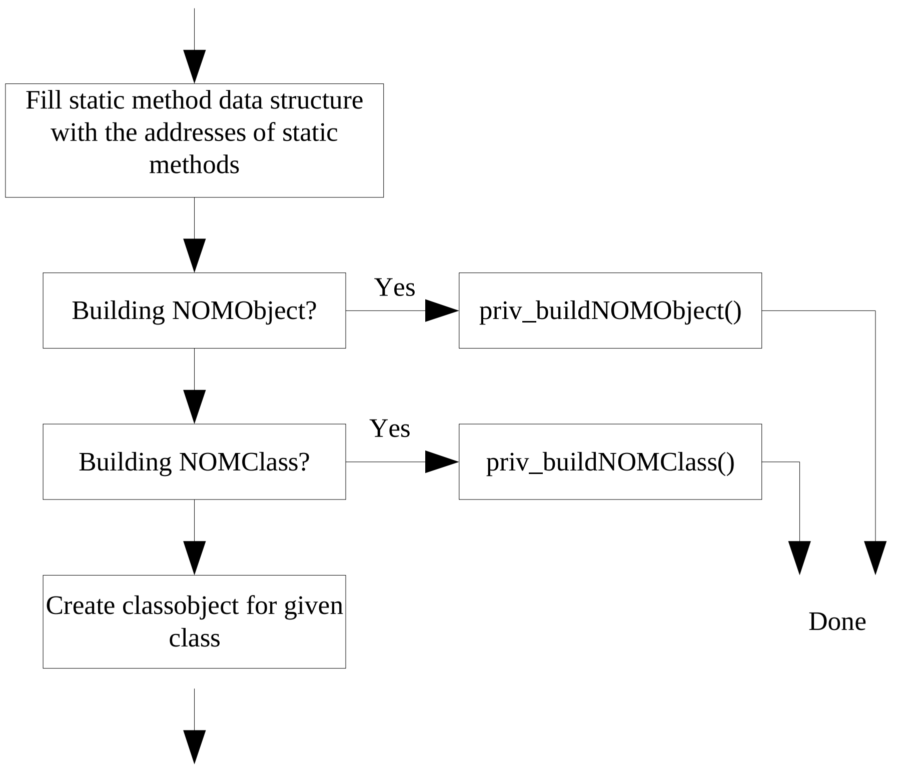
```
        NOMClassMgrObject=NOMClassMgrNew();
```

finally creates the NOMClassMgrObject which is returned to the caller of `nomEnvironmentNew()`.

`NOMClassMgrNew()` is actually a macro defined in the *.h file created by the object compiler. This macro expands into a call to `NOMClassMgrNewClass()` which will build the classobject for the class (if not yet done) and into a call to `nomNew(NOMClass* nomSelf)` which creates the object.

2.3 nomBuildClass()

This function is called to create a class that means a classobject. The implementation provides three different code paths depending if NOMObject is built, NOMClass or any other class.


Fill static method data structure with the addresses of static methods
Building NOMObject?
Yes
priv_buildNOMObject()
Building NOMClass?
Yes
priv_buildNOMClass()
Create classobject for given class
Done

3 IDL compiler

The IDL compiler used for the object model is based on ORBit2 (release 2.14.0 as of this writing). ORBit2 creates CORBA bindings from a given IDL file containing class descriptions in the CORBA IDL language.

To minimize necessary changes the core of the compiler wasn't changed but only the C emitter part. Special features only available in NOM are implemented as macros which are evaluated by the preprocessor which is run by ORBit2 at the beginning of IDL file processing. The resulting code is specially marked so the C emitter can recognize it.

3.1 Starting the compiler

To be written...

3.2 Example IDL file

```
/* Include declaration of parent class */
#include "wpobject.idl"

interface WPFolder : WPObject
{
      /* Specify the version of this class */
      NOMCLASSVERSION(1, 0 );

      /* Methods of this class */
      long wpFolderEchoString(in string input);
      long wpFolderMultIt(in long n1, in long n2);
      long foo_wpEchoString(in string input);

      /* Attributes are supported */
      attribute long fldr_theLong;

      /* Overriding a method of a parent */
      NOMOVERRIDE(wpEchoString);

      /* Declare a (private) instance variable */
      NOMINSTANCEVAR(long theLong);

      /* Usual preprocessor magic is possible */
#ifdef __MYDEFINE__
      attribute long privateAttrLong;
#endif
};
```

3.3 Compiler macros

In the following section all the macros are described which are used when writing IDL files for the Voyager object model.

3.3.1 NOMCLASSVERSION macro

Example:

```
NOMCLASSVERSION( ulMajor, ulMinor);
```

Mandatory

yes

Description

Used to specify the version of the class. When loading classes the system tries to load all the parent classes first. The version information is used to make sure an appropriate parent class release is available and loaded.

Implementation

```
#define NOMCLASSVERSION(a, b ) \
                  const long MajorVersion = (a) ;\
                  const long MinorVersion = (b)
```

The macro expands into constants:

```
const long MajorVersion = ulMajor;
const long MinorVersion = ulMinor;
```

3.3.2 NOMOVERRIDE macro

Example:

```
NOMOVERRIDE(wpEchoString);
```

Mandatory:

no

Description:

Methods introduced by some parent can be overriden by subclasses. This is similar to virtual methods in C++ with the only difference that this overriding is done during loading of the class at runtime.

Implementation:

```
/* Helper */
#define _doconc(a,b) a ## b
#define _conc(a,b) _doconc(a,b)


#define NOMOVERRIDE(a) _conc(void _,\
            _conc(a,_conc(__OVERRIDE__,__LINE__())))
```

The macro expands into:

```
      void _wpEchoString__OVERRIDE__14();
```

The number is the line in the source where the macro is found. This is done so when having several classes in a file all the generated statements are unique even if the names of the overriden method are the same.

The generated line is a method definition only used as a placeholder in the code. The unmodified ORBit2 compiler would treat it like any other defined method. Voyager's IDL compiler checks for any occurence of such lines (it searches for the `__OVERRIDE__` string) and processes it in a special way.

3.3.3 NOMINSTANCEVAR macro

Example:

```
NOMINSTANCEVAR(long theLong);
```

Mandatory:

no

Description:

In addition to attributes wich are defined by CORBA the Voyager object model also supports so called instance variables. These variables are used similar to attributes but can only be accessed from within the class implementation code. They are completely hidden from other code.

If you want to access instance variables from the outside of your class implementation you have to define your own access methods.

Implementation:

```
/* Helper */
#define _doconc(a,b) a ## b
#define _conc(a,b) _doconc(a,b)


/* The ',' in 'attri,bute' is no typo */
#define NOMINSTANCEVAR(a) _conc(attri,bute _conc(a, _conc
(__INSTANCEVAR__, __LINE__)))
```

The macro expands into:

```
attribute long theLong__INSTANCEVAR__17;
```

The number is the line in the source where the macro is found. This is done so when having several classes in a file all the generated statements are unique even if the names of the defined instance variables are the same.

As can be seen a specially named attribute is defined. The C emitter of the compiler checks for attributes of this kind and processes them in a special way.

- No `_get_<attributeName>()` method is generated
- No `_set_<attributeName>()` method is generated