The UDM framework

Arpad Bakay, Endre Magyari

List of contributors:

- **Tihamer Levendovszky**

Institute for Software-Integrated Systems
Vanderbilt University
March 2003

Table of Contents

1. Architectural description

The UDM (Universal Data Model) framework includes the development process and set of supporting tools that are used to generate C++ programmatic interfaces from UML class diagrams of data structures. These interfaces and the underlying libraries provide convenient programmatic access and automatically configured persistence services for data structures as described in the input UML diagram.

The storage technologies currently supported are as follows:

- **XML** with an automatically generated DTD file,
- **MGA**, the native interface of the GME modeling environment [1]
- **Memory**-based storage.

The framework consists of the following modules (also shown in Figure 1):

- The **VisioUML2XML** tool, which converts Visio UML models into XML format.
- The **GME UML** environment with the GME UML paradigm and interpreter, which generates equivalent XML files from GME UML models.
- The **Udm** program, which reads in the XML files generated by the above tools, and generates the metamodel-dependent portion of the UDM API: a C++ source file, a C++ header file, and an XML document type description (DTD).
- The generic **Udm include headers** and **libraries** to be linked to the user's program.
- **Utility programs** to manipulate and query Udm data (**UdmCopy**, **UdmPat**).

UDM can be best used where

- Object-oriented approach is followed to describe the data structures. It basically means that the object structure is defined first in the form of a class diagram, and only the attributes, associations and methods defined there are used in the program.
- The data has no native persistence format, so Udm's supported persistence formats (XML or GME) can be used for persistence. Otherwise, Udm can also be used to create converters to and from native formats.

The typical process of using the UDM is as follows:

- First, a UML metamodel is created in either of the two supported modeling tools (Visio or GME).
- The information in the UML diagram is converted to an XML file using either the VisioUML2XML tool, or the GME interpreter supplied with the GME UML environment. The format of these XML files is Udm's representation of UML class diagram information (partially described by the DTD file Uml.dtd).
- The Udm.exe program is used to generate the paradigm-dependent API files.
- The user includes these files, along with other, generic Udm headers libraries into a C++ project.
- Changes in the UML diagrams are followed by the same procedure. Since most changes also change the generated API, modifications in the programs that use it may also be required.

The simplest case is when a single UDM interface is used either for reading or writing data structures. Other scenarios, like translators, may use several different UDM API-s in the same program.

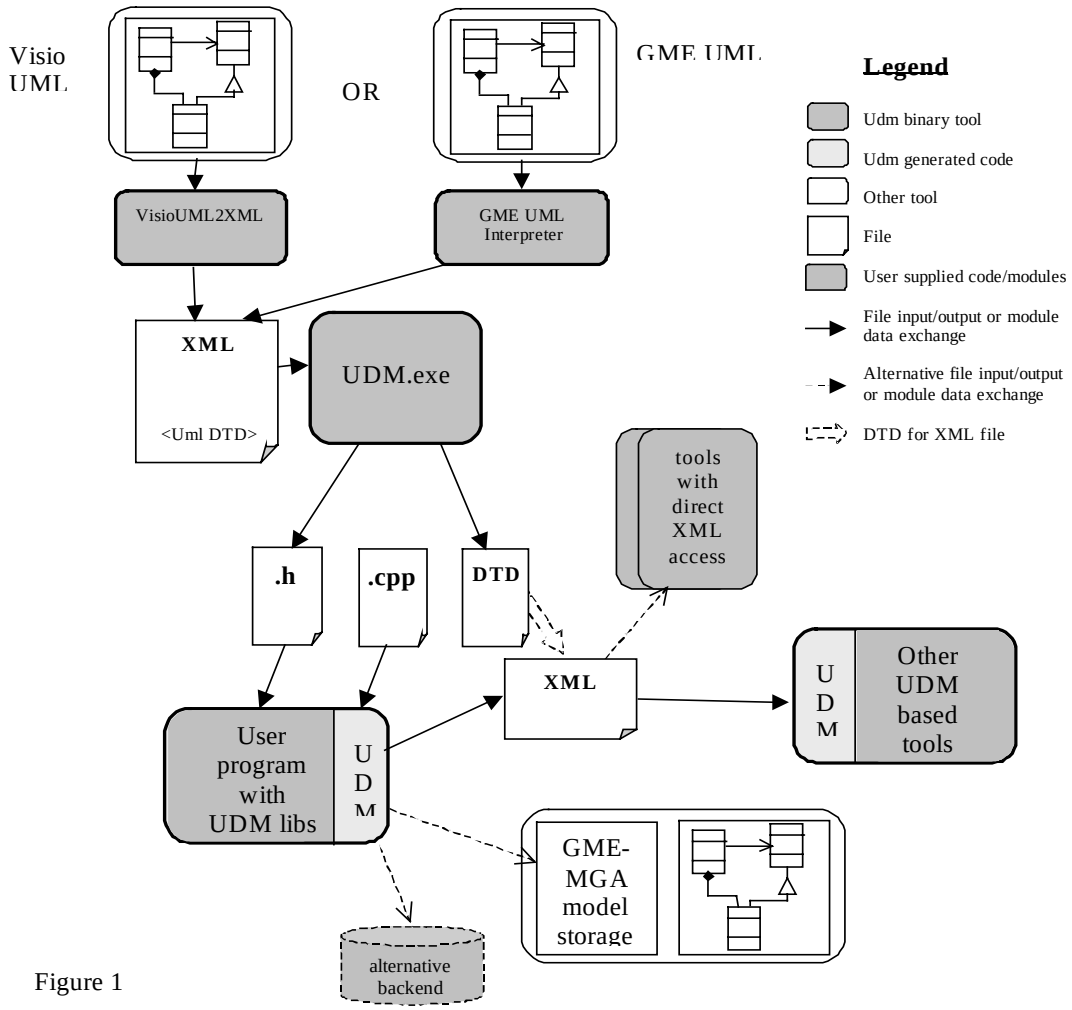


Figure 1

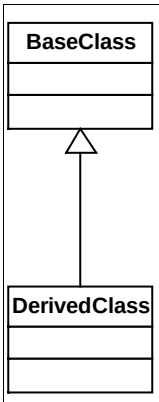
For some typical operations, the Udm package also includes generic programs that work on Udm data. Since these are generic, the data structure information is supplied at run-time as files in the XML UML format.

- UdmCopy is used to port data between different persistent technologies (i.e. XML to GME or vice versa).
- UdmPat is a simple utility that can be used to interpret Udm data and generate text output through a simple pattern based query language.

2. The UDM API

The UML description of the data structure is translated into C++ class definitions in a generated API that is convenient to the programmer, and gives access to all components of the data structure.

2.1 Mapping UML classes to UDM




```
Namespace ExampleDiagram {
  Udm::Object Object;
...
  class BaseClass :
        public Object {
...
  };
  class DerivedClass :
        public BaseClass {
...
  };
}
```

```
// assignment to same type
int f1(DerivedClass &c)  {
  DerivedClass c2 = c;
}

// from derived to base
int f2(BaseClass &b, DerivedClass c)  {
  b = c;
}

// from base to derived
int f3(DerivedClass &c, BaseClass b)  {
  // check if compatible
  if(b.meta() == DerivedClass::meta) {
    c = b;
  }
}
```

Class Diagram	UDM Header	User code example

Figure 1.

For each (abstract or concrete) *UML class* in the source diagram, a C++ class is defined with the corresponding name (Fig 1.). All the classes belong to a namespace, which is by default named after the UML diagram (unless overridden by optional parameters to the VisioUml2XML or Udm tools).
The class definitions allow the definition of instance variables. Such variables are not true objects, just handles (references) to existing instance objects, which reside in the backend (similar to the handle-body idiom defined by Coplien [3].). This has the following consequences:

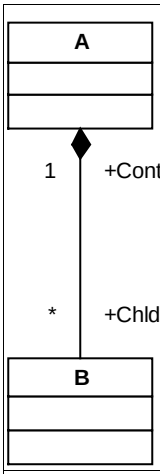
- An un initialized instance variable is an empty reference (reference to Udm::Null).
- Several variables may refer to the same instance object, and simple assignment of variables does not imply the creation of a new instance.
- New objects are always created by the *ClassName*::Create() static member function, which is automatically defined for all classes (not shown in Fig. 1). This function expects the specification of a parent object (except for a single root object, every object must have a parent; see 2.3 for details), and an optional child role.

Inheritance in the UML diagram is reflected as C++ public inheritance. Consequently, all attribute, composition, and association access methods of the base class are seamlessly reflected in the derived classes. Multiple inheritance is also supported, with C++ inheritance relations converted to virtual as necessary.

All classes in the hierarchy are descendants of Udm::Object, which defines the generic functionality of objects in UDM.
An object can decide its real type through the meta() method. Each class has a static and constant type description object (see 2.4) accessible as xxx::meta. These together make it possible to determine compatibility between classes, objects and variables. The type information objects also provide reflection information (associations, attributes).

Note: The VisioUML2XML tool currently does not preserve the abstractness of classes, thus all classes are regarded as non-abstract.

2.2 Composition




```
class B :  public Object {
  Udm::ParentAttr<A> ContainingA();
  Udm::ParentAttr<A> parent()
}


class A :  public Object {
  Udm::ChildrenAttr<B> ChldrnB();
  Udm::ChildrenAttr< B>
                B_kind_children();
}
```

```
void Siblings( B &bb) {
   A aa = b.ContainingA();

  set<B> sibs = A.ChldrnB();
  set<B> iterator bi;
  for(bi = sibs.begin();
      bi != sibs.end();
      bi++) {
    // process (*bi)
  }
}

void ChangeParent( B &bb,
                   A &newaa) {
  bb.parent() = newaa;
}
```

Class Diagram	UDM Header	User code example

Figure 2.

UML composition (containment) relationships are translated into access methods at both the 'child' and 'parent' side (Fig. 2.).
These access methods return instances of wrapper classes that can be used to read and assign new value to the relationships:

Udm::ParentAttr<*xx*> represents the single parent of an object. It can be assigned to an object of type *xx*, or its value can be changed to any *xx* instance (see the example in Fig 2)

Udm::ChildAttr<*xx*> is used if the maximum multiplicity at the child side is 1. It represents the single child of an object. It can be assigned to an object of type *xx*, or its value can be changed to any *xx* instance.

Udm::ChildrenAttr<*xx*> is used if the maximum multiplicity at the child side is >1. It represents the set of child objects. It can be assigned to an object of type set<*xx*>, or its value can be changed to a new set<*xx*> of *xx* instances.

Objects can access their parent and children in two ways:

1 . Access by composition relationship. Returns objects only if they are linked together with the composition specified. Examples are ContainingA() and ChildrenB() in Fig. 2 above. This is the more commonly used access method.

2. Access by parent/child type. Returns all parent/child objects that match the specified datatype (either directly or through inheritance), regardless of the composition relationship used. Examples are parent() and b_kind_children() in Fig. 2 above.

If there is only one possible relationship between two objects, the two access methods are equivalent. Otherwise the set of objects returned by the type-based access are never smaller than the corresponding relationship-based access.

The two access methods generated on the parent side are:

a. An access based on childroles. If the role has a name on the child side, the access method is named after the rolename. If the rolename is empty, the name of the access method is '*xxx_child*' or '*xxx_children*' (depending on the maximum child multiplicity) where *xxx* is the classname at the child side of the composition.

An access based on child class types. The name of the access method is '*xxx_kind_child*' or '*xxx_kind_children*' (depending on the maximum child multiplicity) where *xxx* is the classname at the child side of the composition.

Further two access methods are generated on the child side as well:

b. A role-based access, which is named after the parent side role name of the composition relationship, or, if that name is empty, following the pattern '*yyy_xxx_parent*', where *yyy* is the rolename on the child side, and *xxx* is the classname on the parent side (if the child rolename is also empty, the form '*xxx_parent*' is used).

c. For the child-to-parent direction, there are no type-based access methods provided, but an additional catch-all 'parent()' method is generated, which returns the actual parent, whichever it is. Its return type is selected as the most specific type still compatible with all possible parents. If the method name 'parent()' is already taken (i.e. the UML diagram defines a composition child role for this class where the parent end is named 'parent'), this catch-all parent() method is omitted.

2.3 Creating and deleting objects

A basic concept of UDM data networks is objects are organized in a single tree, i.e. except for a single root object, all objects have a containing parent. Generally, the lifetime of UDM objects is bound to their containment in the object tree. New objects are always created with their parent explicitly specified, and likewise, an object is deleted when no other object contains it any longer.

Each UDM class has a static *Create* method, which is used to create new object instances:

```
class A :  public Object {
        ...
        static A Create(const Object &parent,
                        const CompositionChildRole &role = Udm::NULLCHILDROLE
                       );
        ...
}
```

The second argument is optional, if there is only one valid composition between the parent and the new child. Otherwise, it needs to be specified to resolve ambiguity.

Objects can be deleted by making it orphan. This can be accomplished in several ways:

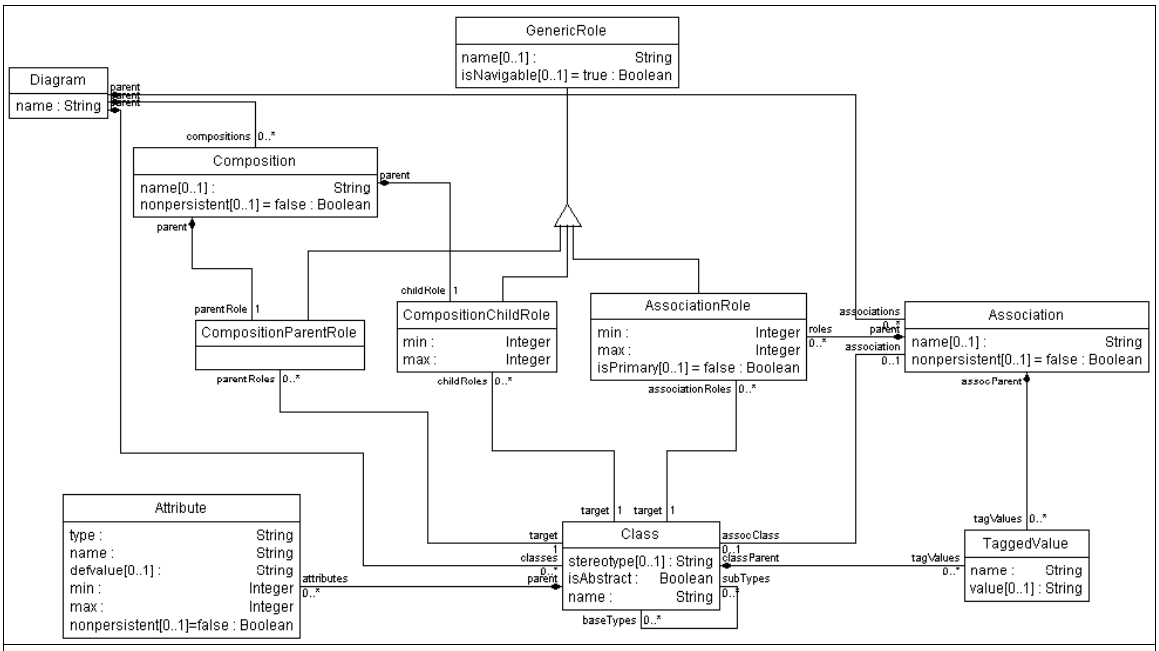
1. By setting the parent() (of an object to NULL)
2. By assigning a new child set to the parent, omitting from the set the objects to be deleted.

Note 1.: Most backend technologies immediately delete objects in this case. The XML backend however, allows objects to temporarily exist without a parent, and as long as there is a reference to it, it can be accessed, reconnected to another parent, etc. However, such orphans are never recorded in the persistent format (i.e. the XML file). Users of this technique in XML must be aware that this feature is not provided by all backends.

Note 2.: Some backend techniques (e.g. XML) allow multiple childroles to be simultaneously active, as long as the parent object is the same (e.g. a 'door' in a 'hall' may be an 'entrance' and a 'fire_exit' at the same time). Removing the object from the child set of one role only, still keeps the object bound to its parent by the other. Thus, to delete an object, all roles must be removed, or the role-independent 'parent()' access function is to be assigned NULL. Note that multi-role composition is not supported by most backends.

2.4 Accessing the meta-information

The meta-information is also accessible through the API. This capability can be used to support reflection, dynamic type identification, etc. This meta-interface is actually also a UDM datanetwork by itself (with a memory-based, StaticDataNetwork backend), so the styles of the two interfaces are the same. The UML diagram of the meta-datanetwork is shown in Fig. 3.



The UML diagram of the meta (meta-metamodel)

Figure 3.

Each of the objects mentioned in this section (class object, attribute object, composition parent- and child role objects, and association role object) have a static member variable ‘meta’, which provides access to the corresponding meta-information. Related association roles are further bound together by the association object, which is accessible from the association roles and vice versa. The same technique, a composition class is used to bind corresponding composition child and parent roles together. For the UDM interface, all these objects are instances of the C++ classes Class, Attribute, AssociationRole, CompositionParentRole, CompositionChidRole, Association, and Composition, respectively (all of these class names are contained in the ‘Uml’ namespace, which is separate from the namespace of the generated data interface).
Since all the metainformation are implemented by StaticDataNetworks, which are backends in the memory, you can even change runtime both the meta, and the meta-meta models of UDM and the currently used API/ClassDiagram.
This feature raises new possibilities. Even more, you can create your metamodel runtime, based on the meta-meta model (UML), and create/open a data network (any backend) based with the runtime create diagram as meta information.

2.5 Attributes

Since version UDM 1.30 UDM uses a new syntax to specify attributes. The new syntax is compliant with the UML notation guide. (OMG-UML, v1.4)

[volatile] [*visibility*] *name* : *type-expression*[*multiplicity ordering*] = *initial-value*

-The ‘volatile’ keyword is optional. If it’s present it means that the attribute is non-persistent, and it’s not recorded in the backend. This obsolates the previous way of declaring non-persistent attributes.
-The *visibility* is one of:
    - \+ public
    - \# protected
    - \- private
    - ~ package

    The *visibility* marker may be suppressed. Visibility may also be specified by keywords(*public, protected, private, package*). This is currently not implemented in UDM, everything gets generates as ‘public’. This will make sense only when operations will be supported. However, the visibility information can be obtained as meta-information, and the Uml meta-metamodel is extended in this direction.
-*Name* is an identifier string that represents the name of the attribute.
-*Type* is either *String*, *Real*, *Boolean* or *Integer*.
-*Multiplicity* shows the ordering of the attribute, as specified in the UML notation guide. Examples:

    - 0..1
    - 1
    - 0..*
    - 1..*
    - 1..6
    - 1..3,7..10, 15, 19..*

-The *ordering* property is meaningful if the multiplicity upper bound is greater than 1. (array attributes) . Currently, UDM supports this feature in all backend types, and in case of non-persistent attributes as well.
-The *initial-value* is defined as a value or a list of values – in case of array attributes - of the specified *type*. This is fully supported by UDM.

Examples of valid attribute specifiers:

```
public ModelName:String="Default Lamp Name"
ArrayStr:String[0..* ordered]="second","first"
ArrayInt:Integer[1..* ordered]=5,4,3,2
ArrayBool:Boolean[0..* ordered]=false,true,false,true
ArrayReal:Real[0..* ordered]=9,8,7,6
volatile public TempArrayStr:String[0..* ordered]="tempdef3","tempdef2","abcdef"
volatile TempArrayInt:Integer[0..* ordered]=9,8,5,6
volatile TempArrayReal:Real[0..* ordered]=40,35,26,17
volatile TempArrayBoolean:Boolean[0..* ordered]=true,false,true,false
```

For each *UML attribute* an access method is defined in the corresponding C++ class (Fig. 4.). These access methods are named after the attribute name, and return an object. These objects can be converted into or assigned new values of a suitable data type. Supported UML data types are String, Integer, Boolean, and Real (defined as Double in Visio

diagrams), which are mapped to the C++ data types string (as defined in the Standard Template Library [STL]), integer, bool, and double respectively.

Class Diagram	UDM Header	User code example

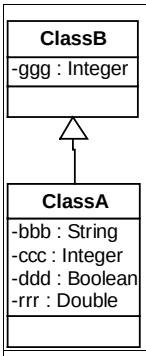
	`class ClassB : public Object {`
	`  Udm::IntegerAttr ggg();`
	`}`
	`class ClassA : public ClassB {`
	`  Udm::StringAttr bbb();`
	`  Udm::BooleanAttr ddd();`
	`  Udm::IntegerAttr ccc();`
	`  Udm::RealAttr rrr();`
	`}`
		`void ProcessAttrs( ClassA &a) {`
		`  a.bbb() = "foo";`
		`  if(a.ddd()) {`
		`  a.rrr() = a.ccc() + 5.0;`
		`  }`
		`  a.ddd() = false;`
		`  // works for baseclass`
		`  // attrs as well`
		`  a.ggg() = 55;`
		`}`

Figure 4.

C++ inheritance mechanism provides that attributes defined in a baseclass are also accessible in its subclasses (as long as names are unique).
UML diagrams may contain access permissions on attributes, but these are not reflected in the generated interface, where all attributes are considered public.

2.5.1 Array Attributes:

Udm starting from release 25 June 2002 supports attributes of type array.
Arrays can be of Integer, String, Real and Boolean. In the API, they are mapped to vectors of coresponding C++ types.

An attribute of type array should be defined as follows:

ArrayStr:String[0..*] //attribute is optional

or

ArrayStr:String[1..*] //attribute is required

When using attributes of type array with MGA backend, these attributes must be declared in the GME Meta as attributes of type string. This is because MGA does not support array attributes, and they are implemented in MGA using attributes of type string.

Array attributes in MGA and XML backends are not really implemented, because the underlying backend does not support them. However, UDM provides this functionality by using the string attributes of the backends.

In MEM backend, there is real suppport for array attributes and it has no performance penalty over simple-value attributes.

If UDM XML/MGA DataNetworks which use attributes of type array are directly edited, it should be considered how UDM represents arrays as strings in these backends:

Array of integers and floats:	values delimited by ';'
Array of booleans:	sequence of 'true's or 'false's (case insensitive) delimited by ';'
Array of strings:	strings delimited by ';', ';' can be escaped by '\' '\' can be escaped by '\'

User code with array attributes

In user code, array attributes can be set/get in the same was as with simple attributes.

The only difference is that not values, but vectors of values are accessed.

Examples:

-Checking the size of the array:

```
if (person.name())
{
      //the array has elements
}

if (!person.name())
{
      //the array has no elements
}

vector<string> names = person.name();
int size = names.size();
//the array has size elements
```

-getting the attribute values:

```
vector<string> names = person.name();
```

-setting the attribute values:

```
vector<string> names;
names.push("John");
names.push("Jerry");
person.name() = names;
```

-adding values to the attribute:

```
vector<string> names;
names.push("John");
names.push("Jerry");
person.name() += names;
```

-getting the value at an index in the array

```
string name_0 = person.name()[0];
```

```
string name_1 = person.name()[1];
```

Note: Index is a 0 based index. If no such element exists, either string(), or double() or long() or bool() is returned. One may check the size of the vector before addressing individual elements. In such cases the value is undefined.

-setting the value at an index in the array

```
person.name()[1] = “Jerry”;
```

Note: If the array is smaller than the index then it’s filled automatically to the size requested by index with string(), double(), long() or bool(); These values are undefined. Then, the value at the position is changed to rval.

-adding a value to the value at an index int the array

```
person.name()[1] += “,  jr.”;
```

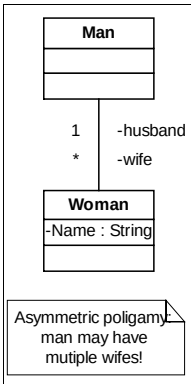
Note: When the array is smaller than the index, then the same note applies as in the case of assignment operator before.
The += operator does not work – and also does not make sense – for boolean arrays.

Note: The = and the += operators for array of strings, integers, and reals are defined for all related types:

- In case of string arrays, you can use the operators with right values of type string and const char *.
- In case of real and integer arrays, you can use the operators with right values of any numeric type: float, double, long, int

Udm currently does not support UML class operations.

2.6 Associations




```
class Man :  public Object {
  Udm::AssocAttr<Woman> wife();
}

class Woman :  public Object {
  Udm::PointerAttr<Man> husband();
  Udm::StringAttr  Name();
}
```

```
// assignment to pointerAttr
void wedding1(Man h, Woman w) {
    w.husband() = h;

}

// equivalent
// assignment to assocAttr
void wedding1(Man h, Woman w) {
    h.wife() += w;

}

// query
int wifenames(Man h, string &s) {
   set<Woman> wives = h.wife();
   set<Woman>::iterator wi;
   wi = wives.begin();
   while(wi != ws.end()) {
     s += wi->Name();
     wi++;
   }
   return wives.size();
}
```

Class Diagram	**UDM Header**	**User code example**

Figure 5.

Associations without an association class (Fig. 5.) are accessed in a way very similar to compositions, with the only difference that associations are symmetric. The access methods for both ends of the association are named after the corresponding association role names. Associations without names at either end are considered non-navigable in that direction, thus no corresponding access method is generated. The type of the wrapper object returned by an access method again depends on the cardinality of the corresponding end of the association: it can be read as or written by a single variable (if the maximum cardinality is 1) or by an STL set<T> of compatible objects.

Associations represented by association class are also supported. Association classes display the nature of both classes and associations, i.e. besides the associated peers they also have attributes, and may participate in associations or containment relationships. This parallelism is maintained as much as possible, but there are differences to be observed (e.g. when creating new associations).

The UDM metamodel (Uml.xml) has two optional parameters for associations:

- **isPrimary** can be optionally assigned to one of the two association roles, if that role is regarded as the primary logical direction of the association (looking from the middle of the association). In the UML standard it is usually represented by an arrow prepended or appended to the association name. E.g . if an association represents the 'feeds on' relationship between an animal and something edible, then the primary association role will probably be the one that points to the food.
- **IsNavigable** is a flag that optionally indicates that an association is not navigable in either direction. If this attribute is false, no access method is generated to navigate the association in that direction (in the tools, UML tools navigability is indicated with arrows at the ends

2.7 Constraints

Currently, *UML constraints* are not checked by UDM, but a data backend may optionally refuse violating operations.

2.8 Non-persistent class attributes

Attributes of automatically defined UDM classes are by default persistent, i.e. they are stored in the output data. However, it is often desirable to use data in classes that are not persistent, but used to store temporary information. Such attributes should be declared using the 'volatile' keyword when declaring the attributes.

Non-persistent attributes support the same data types as the ordinary ones.

2.9 Archetype, derived and instance objects

Since version 1.62, UDM supports the sub-typing and instantiation of existing UDM objects, allowing thus the existence of *subtypes* and *instances*. Such instantiated and/or sub-typed(derived) objects have an *archetype* object, which is the other end of this relationship, and it is the object that they are sub-typed(derived) or instantiated from.

Objects with hierarchy (containing children) can also be derived and/or instantiated. In such cases the whole sub-tree, rooted at the archetype object, is derived and/or instantiated. If an object is directly derived/instantiated from it's archetype we call it *primarily derived*. This means that a primarily derived object can be created only with explicit user code, like calling CreateDerived() or CreateInstance(). An object is *inherited child* when one of it's parents is a primarily derived object. Such inherited child objects are created whenever an object with hierarchy is derived or instantiated. Every object contained in the sub-tree rooted at the archetype will have their corresponding derived/instantiated inherited child object in the hierarchy rooted at the new derived/instantiated object, which is primarily derived.

Attribute values of instantiated/derived objects are kept synchronous with the values of the corresponding attributes in archetype object as long as they are not modified alone(only through their archetype). Once an attribute's value is modified alone(directly on derived/instantiated object), the attribute becomes "desynched" from the archetype, which means that it's value it is not synchronized to the corresponding attribute's value in archetype.

Derived objects can be modified, but inherited child objects(children and links) can not be removed.

Instance objects can not be modified, but inherited child objects of instance objects can serve as connection end-points for connection objects(links) outside the instance object.

Methods for creating & manipulating derived and instantiated objects:

Creation:	CreateDerived(), CreateInstance()
Query:	Derived(), Instances(), Archetype()

(All non-static member functions of the generated classes.)

3. Using the UDM API

3.1 Generating UDM source files

1. Use Visio or GME with the UML paradigm to create a UML diagram for the data model of the target application.
2. Use the VisioUML2XML tool, or the UML2XML GME interpreter to generate XML files from the diagrams.
3. Run Udm.exe to generate a .cpp and a .h file from the XML format. (Udm.exe also generates an XML DTD file.)

Udm.exe, VisioUML2XML, and the UML2XML interpreter are all included in the *Udm binary release*. It also contains the include files, static and dynamic libraries needed to build a UDM application, along with documentation and examples.

3.2 Creating a C++ project that uses UDM

The easiest way is to start from an existing project (e.g CreateLampModel.dsp), but the following procedures also include instructions for doing it the hard way:

The following components are needed (and supplied in the distribution):

- -Microsoft Visual C++ 6.0 (not included in the distribution)
- -The Xerces 1.2 XML Parser headers, library and DLL
- -The SGI 3.2 Standard Template Library headers
- -The UDM libs (UdmBase.lib, UdmDom.lib and UdmGME.lib)
- -The UDM headers

Your project needs to include the UDM API sources (generated in the previous step) and your own source files (at least one that defines the main() or WinMain() function).

Step-by step instructions:

1. Create a new project (e.g. a Win32 Console application, or an MFC application) in Microsoft Visual Studio.
2. Make sure the project includes the SGI STL headers (which are at <install-dir>/include/stl in the distribution)
   - –add the STL path to *Project/Settings/C++/Preprocessor/Additional include headers*
   - –or add the STL path to the system-wide settings in *Tools/Options/Directories/Header files* (make sure STL precedes the standard Microsoft includes)
3. Make sure your project includes the generic UDM headers by adding the header directory (normally include/udm) to either of the settings mentioned above, or by copying the header files to your project directory.
4. Include the UDM base library UdmBase.lib, and at least one of the backend support libraries like UdmDom.lib (which requires Xerces-c_1.lib) and UdmGME.lib) in

Project/Settings/Link/General/Object/Library modules. For the debug builds, use the corresponding debug versions (where the basename ends with '_D'). These libraries are normally located in the Lib/Udm subdirectory of the distribution (the location for the Xerces-c_1.lib and Xerces-c_1D.lib is Lib/Xerces). You will need to add these directories to the input library path, or specify libraries with pathnames, or set the system-wide settings for paths in Tools/Options/Directories/Library files accordingly.

5. Make sure all the configurations of your project refer to the 'Multithreaded DLL' or 'Debug Multithreaded DLL' run-time libraries in the *Project/Settings/C++/Code Generation/Use run-time library* option. The libraries use C++ exceptions, so you probably also want to enable exception handling.
6. Add the generated UDM API files (a C++ and an H file) to the project. In addition to these, your project will need to have at least one source file, which defines the main() (or WinMain()) function.
7. The files that use the API need to include the UDM API header file. The module that creates the UDM project object (UdmDom::DomDataNetwork if the XML backend is used), will also need to include the respective backend definition header file ('UdmDom.h' or 'UdmGme.h') as well (this file is located among the standard UDM headers, e.g. in include/udm).
8. If your project uses the precompiled header features, including all headers in the master header file (StdAfx.h) of the project is recommended. In any case, however, the API .cpp file should be set individually to '*Not using precompiled headers'* in the *Project/Settings/C++/Precompiled headers/* pane.
9. Set additional options according to your preferences.

To run any application that uses the XML backend, the Xerces DLL (xerces-c_1_2.dll) must be either in your system path or in the directory of the executable file.

Also, the generated .DTD file must be available to load XML files. It must be either in the current execution directory, or be attached to the project as a resource of the same name (e.g. "SAMPLEDIAGRAM.DTD", including double quotes), and the custom, "DTD" resource type. To add resources, an .rc file (and possibly a resource.h file) need to be inserted into the project.

3.3 Generic program structure

A minimal UDM-based application consists of a user-supplied source file (which #includes the .h file generated by the API generator, and possibly other UDM headers), the generated C++ file, and the stationary UDM modules, which are provided as libraries.

Udm is intended not to restrict the program structure in any way. It can be used at any location or at any time, either throughout the whole application, or just within single procedures.

Working with UDM begins with creating and opening a DataNetwork object (see below). Objects in the data tree can be accessed through the root object, which is in turn

accessible through the data network, and then navigating (or building up) the rest of the tree. The DataNetwork object and the data objects in the tree are the primary objects used throughout the application.

Errors in the operation of the UDM result in a **udm_exception** exception thrown. The **what()** method of the error object contains information on the error.

3.4 Udm::DataNetwork objects and member functions

The abstract DataNetwork type is used to represent a UDM data tree, or 'project'. DataNetwork interface is used to manipulate the data network as a whole, like creating, loading, closing, and transaction control. DataNetwork is an abstract class, and each backend provides an implementation to this type, like DomDataNetwork, GmeDataNetwork or StaticDataNetwork.

If several data trees are to be used, a corresponding number of DataNetwork objects are used. Data stored in different data trees are completely separated, thus no cross-relations are allowed. Since Udm generates the interfaces into user-defined namespaces, class name clashes can also be avoided.

```
DomDataNetwork::DomDataNetwork (const Udm::UdmDiagram &metainfo);
GMEDataNetwork::GMEDataNetwork (const Udm::UdmDiagram &metainfo);
StaticDataNetwork::StaticDataNetwork (const Udm::UdmDiagram &metainfo);
```

These constructors create DataNetwork-derived instances and attach them to a UdmDiagram. The metainfo parameter represents a metamodel, and is normally available from the generated header file, where it can be referred to as '***Namespace*::diagram' .** (The Udm generated headers must be included anyway. If only a single Udm diagram is used, the *'using namespace'* directive can be used to avoid the necessity to use namespace qualifiers. With multiple diagrams, however, this may lead to name clashes, so the use of explicitly qualified names may be necessary.)

A newly constructed DataNetwork needs a backend storage created or opened into it.

```
void DataNetwork::CreateNew(
        const string &systemname,
        const string &metalocator,
        const Uml::Class &rootclass,
        enum Udm::BackendSemantics sem = Udm::CHANGES_PERSIST_ALWAYS
);
```

Create a new data tree in a backend storage.
systemname is the name used by the backend (a filename or database connection string).
metalocator specifies the metainfo used by the backend (e.g. a DTD for XML backends, or a paradigm for GME).
rootclass is a class from the diagram, an instance of which is created as the root object of the new data network.
sem is one of the predefined semantic constants:

- **Udm::CHANGES_PERSIST_ALWAYS:** committed sequences are immediately recorded in the original backend (CloseNoUpdate, CloseAs, SaveAs are not implemented)
- **Udm::CHANGES_PERSIST_DEFAULT:** committed sequences are recorded in current backed unless CloseNoUpdate, CloseAs, or SaveAs is used
- **Udm::CHANGES_LOST_DEFAULT:** committed sequences are lost unless Close, CloseAs, or SaveAs are used

(Not all constants are accepted by all backends.)

```
void DataNetwork::OpenExisting(
        const string &systemname,
        const string &metalocator,
        enum Udm::BackendSemantics sem = Udm::CHANGES_PERSIST_ALWAYS
);
```

Read and open a data tree from a backend storage.
systemname is the name used by the backend (a filename or database connection name).
metalocator specifies the metainfo used by the backend. This parameter may be omitted (by passing an empty string), unless the meta used to create the data is not locatable, or the metainfo is deliberately changed.
sem is one of the predefined semantic constants, as described above

```
void DataNetwork::CloseWithUpdate();
```

Close the data network and record changes in the backend storage. Closing is not mandatory, since the destructor will close the data network anyway

```
void DataNetwork::CloseNoUpdate();
```
Close the data network without recording changes in the backend storage. This function may not be implemented with all backends.

```
void DataNetwork::SaveAs(string systemname);
```

Save the info into a data network into a new backend database specified by **systemname.** This function may not be implemented with all backends.

```
void DataNetwork::CloseAs(string systemname);
```

Close the datanetwork, and save it to a new database specified by the **systemname**. This function may not be implemented with all backends.

```
void DataNetwork::CommitEditSequence();
void DataNetwork::AbortEditSequence();
```

Edit sequences are the abstractions of simple (non-nestable and automatically chained) transactions.

CommitEditSequence finalizes changes in an edit sequence (transaction). Depending on the backend semantics specified when opening the file, the data may or may not be physically flushed to storage.

AbortEditSequence revokes changes done through the previous edit sequence (transaction).

Using either of these methods automatically terminates an edit sequence and starts a new one. CloseWithUpdate, CloseNoUpdate, SaveAs and CloseAs also terminate edit sequences, while CreateNew, OpenExisting and SaveAs automatically initiate new ones. Thus the explicit use of CommitEditSequence and AbortEditSequence is optional, but if they are not used, all edit operations on an open data network are regarded to form a single edit sequence.

bool DataNetwork::isOpen();

Check if the datanetwork is open.

Object DataNetwork::GetRootObject();

Get the root object of the data network, This is the object created when the whole backlend storage for this data tree was initialized.

const Uml::Diagram & DataNetwork::GetRootMeta();

Get the metainfo of the data network. This is the value specified in the constructor.

You can also use the **SmartDataNetwork** class, which initializes a smart data network.

SmartDataNetwork::SmartDataNetwork (const Udm::UdmDiagram &metainfo);

A **SmartDataNetwork** will try to guess the type of the backend based on the **systemname** parameter, in the following way:

- ***.xml** means **DomDataNetwork**
- ***.mga** means **GmeDataNetwork**
- ***.mem** means **StaticDataNetwork** (no storage)

3.5 UDM Data objects

Data objects are instances of classes defined in UML. It is relatively straightforward and intuitive to work with them. Use the generated API header file as a reference to follow the instructions below.

Objects that are instances of UML Classes are all direct or indirect descendants **Udm::Object** (which is also typedef'd as **Object** in the generated namespaces).

3.5.1 Creation

```
static CLASS Create(
        const Object &parent,
        const Uml::CompositionChildRole &role = Udm::NULLCHILDROLE
);
```

To **create** an object, the **Create** static function of the class is used. The parent object of the new object must be specified. The second parameter is the composition child role, which can be omitted, if the containment relationship is unambiguous otherwise. E.g:

```
// create a room within a house
Room livingroom = Room::Create(myhouse);
// a new switch is created within an engine. It functions as an emergency switch
Switch secondaryswitch = Switch::Create(engine_11, EmergencySwitch::meta);
```

3.5.2 Attributes

For each attribute specified in the UML Diagram (or in the optional diagram of additional, non-persistent attributes), an access method is specified. This access method returns a smart object, which can be used to set and get the value of the attribute. The value used to set and get must match the type of the attribute, so there are four smart objects defined for the four supported attribute types: StringAttr (use STL strings to set/get), BooleanAttr (use bool), IntegerAttr (use int), RealAttr (use double).

E.g:

```
livingroom.Area() = 550.7;
livingroom.IsHeated() = true;
int height =  myhouse.Height();
myhouse.Address() = "1745 25th St";
```

Note that from the API point of view, there is no difference between persistent and temporary attributes.

3.5.3 Inheritance

Inheritance is automatic for the most part, since inheritance is implemented as C++ class inheritance. E.g. you can access attributes defined in your base class just as the locally defined ones.

bool ::IsDerivedFrom(const Class &derived, const Class &base)

IsDerivedFrom() is used to find it out runtime if two meta-classes are in an inheritance relationship. (Both direct and indirect types of inheritance are detected.)

static CLASS Cast(const Object &a);

To use an object as a more specialized class than its current type, the **Cast()** method can be used. Since the true type of objects is known to the API, it can also cast between unrelated types, i.e. from one baseclass to another. If, however, the type is unrelated to the true type of an object, an exception is thrown.

3.5.4 Parents and children

For each navigable composition relationship defined in the UML diagram, an access method is defined, which returns a **ParentAttr<CLASS>** from the child, and a **ChildAttr<CLASS>** or **ChildrenAttr<CLASS>** from the parent, depending on the maximum multiplicity of the composition. All access objects are settable, but of course setting one of them results in a change on the other side. The **ChildrenAttr<CLASS>** can return or set the children of the object either as a **set<>** or as a **vector<>.** When setting the children with a **vector<>** of children, their order is preserved and will be the same when retrieving it as a **vector<>**. By default, children are ordered by the order they are created. However, with MGA backend this behavior is not guaranteed, and the order of children is undefined. Of course, **set<>** operations also work, but since **set<>**s are ordered containers, the order of children cannot be controlled. (The order key is the object's uniqueId, which again cannot be controlled)

ParentAttr and **ChildAttr** contain a single value, it can be simply set/read from/to a CLASS object. **ChildrenAttr** must be assigned and read through sets of compatible objects. To add/remove members of **ChildrenAttr** one by one, the **+=** and **-=** operators can be used.

E.g.:
bathroom.parent() = myhouse;
House house = bathroom.parent();

set<Room> allrooms = myhouse.rooms();
allrooms.insert(newroom);
myhouse.rooms() = allrooms;
// the above 3 lines are equivalent with this one, and set could be <vector> as well
myhouse.rooms() += newroom;

3.5.5 Associations

For each navigable association relationship defined in the UML diagram, an access method is defined, **Udm::AssocAttr<CLASS>** or **Udm::PointerAttr<CLASS>** from the parent, depending on the maximum multiplicity of the association. CLASS here is the most specific common type of the objects on the other side of the relation (. Both access objects are settable, and of course setting one of them results in a change on the other side.

Udm::PointerAttr contains a single value, it can be simply set/read from/to a CLASS object. **Udm::AssocAttr** must be assigned and read through sets of compatible objects. To add/remove members of **Udm::AssocAttr** one by one, the **+=** and **-=** operators can be used.

E.g.:

```
salesman.Clients() += tomwatson;  // Udm::AssocAttr
set<Person> salesclients = salesman.Clients();
salesman.Mother() = annamary;     // Udm::PointerAttr  (everyone has one mother)
```

3.5.6 Association Classes

Association classes are basically ternary relations between two related Objects, and the association class itself. The access methods of association classes return an extended version of access objects returned by simple associations.

The related objects return **Udm::AClassAssocAttr<ACLASS>** or **Udm::AClassPointerAttr<ACLASS>** objects depending on the maximum multiplicity of the association (as seen from the object queried). In contrary to the simple associations, these objects are parameterized by ACLASS, the type of the association class. They can be freely used to read the association(s), but direct assignment is not permitted, except if an **Udm::AClassAssocAttr** is 'shrunk', i.e. assigned a set that is just a subset of its current value. Similarly assignment of NULL to a AclassPointerAttr is also permitted.

To add new associations with classes, the **AddLink()** and **SetLink()** methods are used.

```
Udm::AClassAssocAttr<ACLASS> AddLink(
        TARGETCLASS peer,
        Object aclassparent
);
Udm::AClassPointerAttr<ACLASS> SetLink(
        TARGETCLASS peer,
        Object aclassparent
);
```

peer is the object to associate with, and **aclassparent** is the object which will be the parent of the association class.

From the association class objects (which are normal objects, accessible through their containing parents, or through other relationships), the peers on both ends can be accessed through the **Udm::AssocEndAttr<CLASS>** access objects. This can be used both to read and to write the target objects.

4. The UDM API and persistence technologies

4.1 Limitations on the UML capabilitites

Reflecting the architecture of several backend services (GME and XML), UDM only supports single-rooted hierarchies. This means that all objects (including instances of association classes) must have a parent, except for a single root object, which is created along with the creation of the database. This is usually not a serious restriction, since creating a top-level container to store multiple, logically 'root' objects is a generally applicable workaround.

Depending on the backend technology used, the framework may be able to handle temporarily orphan objects. However, in other backends (e.g. GME), orphan objects are automatically deleted.

- Some UML features of less importance or acceptance are not supported:
  - Attributes with multiple values, special data types (Variant, Date, reference to other objects)
  - Access qualifiers (public, private, protected)
  - Qualifier attributes and specifications on associations
  - N-ary associations
  - Constraints

- Uml Class names must be unique and non-empty. Attribute names must also be non-empty. Other names may be empty.
  Auxiliary names are generated for empty association end names. These names correspond the name the target class, with the first letter turned lowercase, and –if the maximum cardinality is >1- 'pluralized' using a simple algorithm (i.e. 's' or 'es' appended).

  Inside any Class, the following names will appear as methods, so they must be unique:
  - Attributes names
  - Association end names (on the remote end viewed from the class).
  - Composition parentrole names (for compositions where the class is the child).
  - Composition childrole names (where the class is the parent).
  - Auxiliary names generated if one of the above names are empty.

- Throughout UDM, performance is considered to be secondary to elegance, so this approach is probably not optimal for large data trees (>10000-100000 objects depending the backend).

4.2 The XML backend

The XML backend is based on an extended version of the standard XML-DOM interface. The UDM generator tool - along with the API definition files- also generates the DTD for the XML files compatible with the input interface definition. The basic XML-processing facilities provided are loading (parsing) an XML file into the backend, and saving the backend data structure into an XML file.

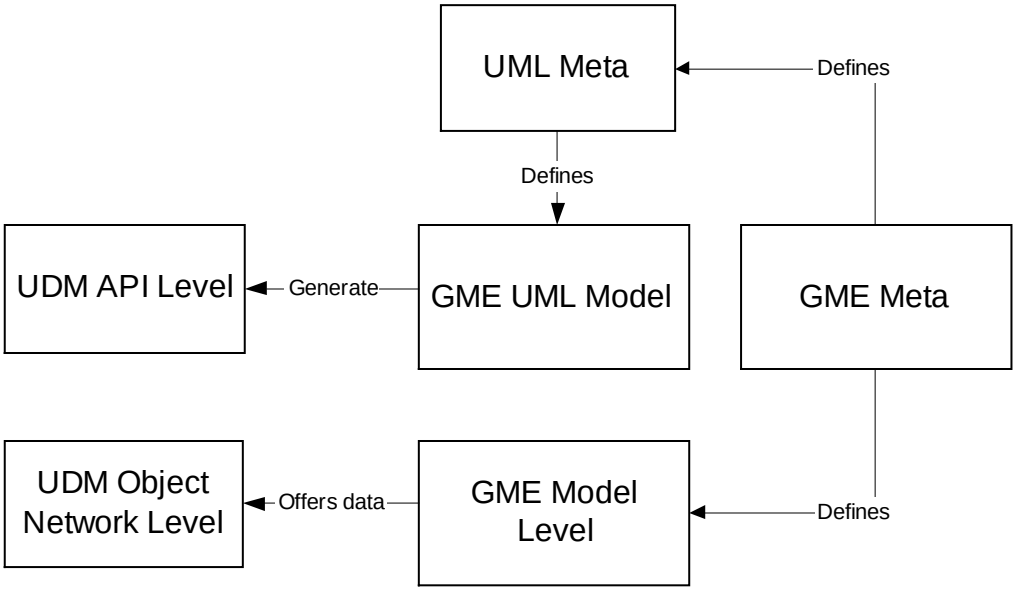
The XML backend imposes the following restrictions:

- -XML does not have a good technique for representing associations and especially lacks support for association classes. These UDM features are thus implemented through specially named XML ID/IDREF attributes.
- -The current version does not do on-the-fly validation when the data tree is modified, and it is not validated when it is saved either, so the validity of the generated XML document is not guaranteed in all aspects. The recommended way to validate XML output is to reload it again as the last step of generation.
- -There is single root object, which is created as the first object when a data tree is opened (or loaded). The root object cannot be changed.
- -The XML backend permits objects to exist without parents temporarily. If such an object is later attached to any parent, it will become a normal object and recorded in the persistent data.

4.3 The GME backend

1. General Concepts

The general structure of the UDM-GME interoperation is depicted in this figure:



In the GME meta editor a UML class diagram paradigm has been defined. This UML class diagram will define the classes to be instantiated at run time. The *udm.exe* utility (for detailed information please refer to [1]) takes this UML class diagram as its input and generates the corresponding C++ API with these classes and relationships defined by the class diagram. Afterwards one can instantiate these classes manually creating an object network, navigate through the generated relationships and use the provided meta information at run time.

However we can use GME to create the actual run-time object network. This document elaborates on the steps necessary to set up such a GME-UDM environment.

The GME backend connects the interface to the MGA library [2]. The operation of the MGA library is also based on meta-information stored in its proprietary data format. Since it is essential to keep the meta-information on both sides consistent, there is a GME-based tool available for generating a matching pair of MGA meta-data and UDM generator input (in XML format) from a UML class diagram.

The GME backend provides functions for creating, opening, and closing GME databases, as well as an API for simple transaction control. It is also possible to create GME interpreters and other components based on UDM.

If UDM is used with the GME backend the following limitations and caveats have to be considered:

1. **Names of UML classes and associations must be unique (or empty).**
   UML does not support non-unique class and association names (although multiple associations or composition with empty names are OK, see below). For this reason, the corresponding GME paradigm is also expected to have globally unique kindnames (athough GME in general permits using the same name for metaobjects in different contexts).

2. **Attributes**
   Normal GME attributes are accessed through UML attributes defined under the same name. The type of the UML attribute must match the GME type specified in the paradigm. Every UML attribute (which is ever used) must have a corresponding attribute in the paradigm, but it is not a problem, if an existing GME attribute has no counterpart in the UML (and is thus inaccessible).

3. **Special attributes**
   The two built-in GME object properties, name and position can be accessed through special UDM string attributes called ‘name’ and ‘position’. Of course they are only accessible through UDM if they are defined in the UML diagram. (If the GME paradigm happens to have attributes called ‘name’ or ‘position’, those will be inaccessible through UDM.)

4. **Associations**
   Associations in the input UML must be defined in a way so that it can be mapped onto one of the association types supported by GME: references, sets, or connections. To resolve ambiguities, stereotypes at either end of an association or the naming of the association and its role names must provide sufficient hints for identifying the applicable GME association type.
   GME imposes complex rules on the relative location of objects bound together by an association. Since UDM currently does not capture these rules (in UML, these could only be represented as constraints, which are currently not supported), operations violating those rules pass through the UDM layer, but they are refused by the GME backend.
   Objects that manifest as folders in GME cannot be involved in associations.

   The trickiest part is how to map UML associations onto any of the GME concepts (references, sets, connections). There are two ways to make sure GME-UDM has the necessary clues to find out the mapping:

   a. Assigning special names to association ends in the UML diagram.

-If an association has no association class and has a navigable end named ‘ref’, the association is considered to be a reference (with the class at the other end of the association being the reference object)

-If an association has no association class and has a navigable end named ‘members’, the association is considered to be a set (with the class at the other end of the assoc being the set object)

-If an association has a navigable end named ‘dst’ and the other end, if navigable is named ‘src’, then the association is a connection. The association must either have an association class, or it must have a non-empty and visible name, which identifies to the GME connection kind to be used.

b.Using special hints in the GME paradigm.
If the indications listed at a./ are not present, the GME paradigm must contain special info to enable mapping:

-If an association (without association class) in the UML diagram is to be implemented by a GME reference, the name of the association name must either be empty or equal to the name of the reference class (i.e. the class found at one of its ends). The GME meta-reference must have the GME registry value ‘/rname’ set to the other association end name in the UML diagram (see [1] and [2] for a discussion on the GME registry). If the UML reference is navigable in both directions, the GME meta-reference registry value ‘/rrname’ must also be set according to the association end name on the side of the reference class.

-If an association (without association class) in the UML diagram is to be implemented by a GME set, the name of the association name must either be empty or equal to the name of the set class (i.e. the class found at one of its ends). The GME meta-reference must have the registry value ‘/mname’ set to the other association end name in the UML diagram. If the UML reference is navigable in both directions, the GME meta-reference registry value ‘/sname’ must also be set according to the association end name on the side of the set’s class.

-If an association in the UML diagram is to be implemented by a GME connection, it must either have an association class or a non-empty

association name. The name of the association class or (if there is no association class) the name of the association must correspond to the name of the GME meta-connection being used. Furthermore the meta-connection needs to have the registry values '/sname' and '/dname' be set to the two association end names of the UML connection (if the connection is not navigable in both directions, '/dname' must be set to the end name of the navigable direction, and '/sname' may remain unset).

Note that if a paradigm allows a class to be involved in several different incoming or several outgoing connections, approach a./ (i.e. special names in the UML diagram) is usually not feasible, since association name ends ('src' or 'dst' in this case) must uniquely identify a single association from all UML classes.

The mapping from UML associations to GME concepts is determined when the GME DataNetwork is created or opened. Associations that cannot be mapped result in an error. Some other errors like invalid connections or references, or trying to establish associations involving folders, result in an error later, when the operation is attempted.

5.Aspects and constraints

Aspects and constraints are not supported by UDM.

6.Step-by-step description

1. Create the metamodel in using the GME metamodeling environment. The association role names must remain the default the "src" and "dst", because the meta interpreter requires this. The real role names **must be set** in the Attributes edit box in the GME "Attributes" panel in the following syntax:

```
sName=<role_source_name>
dName=<role_destination_name>
```

2. Test the paradigm created in the previous step.
3. Using the python script included in the distribution (located in the **distribution_folder\Gme2Uml\Python** folder), generate the UML class diagram from the GME metamodel. The script takes a GME meta as input (in XML format, use File/Export XML menu item in GME) and produces the class diagram also in XML format. If you do not have python installed you can download the free Cygwin from www.redhat.com web site.
4. Set the real role names for each association.
5. Add a class named RootFolder to the model which must be a root of the containment hierarchy.
6. Generate the API with *udm.exe*.
7. Use your test model from step 2.

4.4 The Memory(Static) backend

UDM stores internally the meta-meta and the meta objects in static, memory based implementation of the UDM base objects. These static objects can be used for a custom object network as well, if one doesn't need runtime, on-the-fly persistency, for example, when only the API is needed. For this, all you have to do is to create your objects in a **StaticDataNetwork.** This way, your program will operate on memory-based objects, thus will be much faster.

Since as mentioned earlier, the meta-model objects, and the meta-meta-objects are stored by these StaticObjects, and they can be accessed exactly the same manner as you can access other UDM objects. This means, that your program, at runtime, can possibly change the meta-model and the meta-meta-model information. However, this may be dangerous. If you change the meta-model, some parts of your generated API become invalid. If you change the meta-meta-model, some parts of UDM become invalid.

However, the static data network can be dumped in an efficient binary format to a file. This file will have a *".mem"* extension, and it's not for human consumption. But one can use this for fast data backup and read-back. It's important to understand that changes in a data network loaded from a binary file are *not* recorded on the fly to the file. They may be recorded upon an explicit SaveAs() or CloseWithUpdate() call, or by the destructor when the data network variable goes out of the scope, and it was opened with either **CHANGES_PERSIST_ALWAYS** or **CHANGES_PERSIST_DEFAULT** backend semantics.

The StaticDataNetwork is not a "type-safe" data network. That means, if you are using the lower-lever API (ObjectImpl calls), you have to specify the type of the object, as an Uml::Class, upon creation. StaticDataNetwork objects, StaticObjects store only a reference to the supplied type in the constructors. For this reason, you must make sure that those Uml::Class-es which are referenced by StaticObjects as their types are not freed until there is no reference.

You can overcome this situation by using the SafeTypeContainer static global class, which allows you to get a copy of a type that no doubt will exist until it is referenced. To help those willing to write paradigm & backend independent UDM code, there is a DataNetwork call which can tell whether the backend currently in use is type safe or not. An example code for this is for example in the UdmCopy function:

```
if (p_dstBackend->IsTypeSafe())
        p_dstChild=p_dstRoot->createChild(theOther(*p_currRole), p_srcChild->type());
else
{
        const Uml::Class & safe_type = Uml::SafeTypeContainer::GetSafeType(p_srcChild->type());
        p_dstChild=p_dstRoot->createChild(theOther(*p_currRole), safe_type);
}
```

5. UDM and XMI

5.1 What is XMI?

XMI is an interchange format for metadata that is defined in terms of the Meta Object Facility(MOF) standard, a pair of parallel mappings between MOF metamodels and XML DTDs, and between MOF metadata and XML documents.

The XMI specification also specifies the UML.dtd concrete XML DTD, required for the transfer of UML models (class diagrams) using XMI.

From version 1.40, the UDM framework supports XMI version 1.0 by providing conversion tools between XMI/UML(v1.3) and UDM metadata.

This makes possible that:

a.) any UDM classdiagram can be exported in XMI format an thus can serve as an input for other tools, like code-generators

b.) XMI metadata exported by other UML modelling tools lcan serve as an input for the UDM framework, UDM API can be generated directly from XMI metadata.

5.2 Conversion to/from XMI

The conversion between UDM metadata format and XMI is done by a set of XSLT transformation scripts processed with Xalan XSLT processor engine.

UdmXmi.dll contains the XSLT scripts and the code which does the transformation.

The tools **UdmToXmi.exe** and **XmiToUdm.exe** are command line utilities to transform between the two formats. Both of them has two mandatory arguments: source and destination XML filename. Since the source XML needs to be parsed, the tools expect the corresponding UML.DTD file to be in the same directory.

5.3 XMI input for Udm.exe

From version 1.40 the UDM.exe code-generator accepts metadata in both UDM and XMI format. When the input is in XMI format, the use of the -d switch is mandatory: the path specified this way should contain the UDM version of the UML.DTD. In the current directory or in the same directory with the input XMI should be the OMG version of the UML.DTD. This is needed to parse the XMI input.

5. UDM Internal Architecture

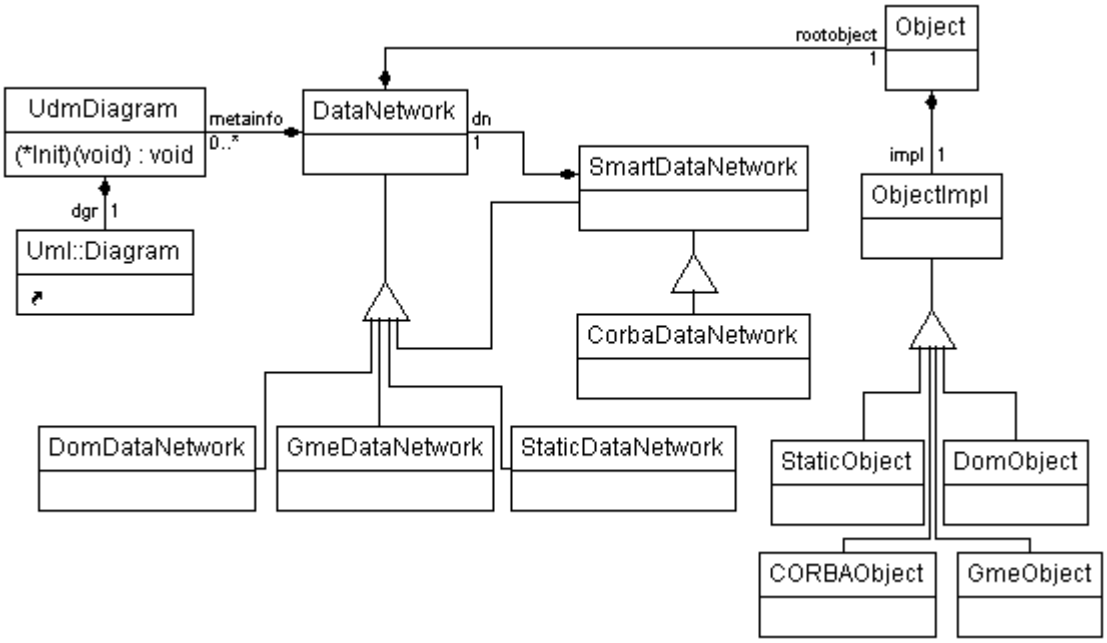


Figure - Udm Internal Arhitecture

5.1 Data Network calls

•virtual void DataNetwork(const Udm::UdmDiagram &metaroot);

The one and only DataNetwork constructor. The meta information is passed as an UdmDiagram.

Implemented in all the back-ends.

•Unsigned long uniqueId()

Retrieves the unique Id of the DataNetwork. Implemented at the abstract DataNetwork class.

•virtual void CreateNew(const string &systemname,
const string &metalocator,
const Uml::Class &rootclass,
enum BackendSemantics sem = CHANGES_PERSIST_ALWAYS) =0;

Creates a new datanetwork.

Systemname - the name of the file
Metalocator - only DOM backend will need this, and it must be the name of the corresponding .dtd file, without the extension
Rootclass - type of the root object
Sem - default behavior of the backend. See earlier in this document.

Implemented in all the back-ends.

•virtual void OpenExisting(const string &systemname,
const string &metalocator = "",
enum BackendSemantics sem = CHANGES_PERSIST_ALWAYS) =0;

Opens an existing data network from a file.
Systemname - the name of the file
Metalocator - DOM will need if the XML file header does not contain correct or relevant information about the location of the corresponding .dtd file.
Sem - the default behavior of the backend.

Implemented in all the back-ends.

•Void CloseWithUpdate() = 0;

Closes an opened data network and records the changes in the file.
Implemented in all the back-ends.

•Void CloseNoUpdate();

Closes an opened data network without recording the changes in the file.
Implemented in all the back-ends.

•Void SaveAs(string systemname);

Saves an opened data network to a different file name. Does not close it.
Implemented in all the back-ends.

•Void CloseAs(string systemname);

Closes an opened data network. If it was opened with CHANGES_PERSIST_ALWAYS or CHANGES_PERSIST_DEFAULT then it saves before closing it.
Implemented in all the back-ends.
.

•Bool isOpen();

Determines whether a data network is opened (either with CreateNew, or with OpenExisting) or not.
Implemented in all the back-ends.

•Void CommitEditSequence() ;

Transaction control. Commits an edit sequence. Implemented only in GME backend.

•Void AbortEditSequence() ;

Transaction control. Aborts an edit sequence. Implemented only in GME backend.

•Object GetRootObject();

Obtains the root object of the data network. Implemented in all the back-ends.

•Object ObjectById(Object::uniqueId_type);

Obtains an object by its ID.
Implemented in all the back-ends.

Note:
DOM & GME back-ends maintain unique Ids for objects within the same data network, and these Ids are usually preserved when saving to and reloading form a file.
STATIC back-ends maintain a unique Id in a process, and Static Object Ids are not preserved when saving to and reloading from a file. Thus, StaticDataNetwork::ObjectById() returns the object even if it does not belong to the data network, but exists. If it does not exist, exception is thrown.

•static void RegisterBackend(const string &sig,
const string &ext,
DataNetwork *(*crea)(const UdmDiagram &))

A backend can be registered to Smart Data Network operation.
When invoked with extension *ext*, the function ***crea*** will be invoked for creating the data network.
Sig is a descriptive name, characteristic to the backend.

•static void UnRegisterBackends()

Un-register all registered back-ends.

•static string DumpBackendNames()

Dump registered back-end names to a string.

•static DataNetwork *CreateBackend(string filename, const UdmDiagram &metainfo)

Creates a backend by deducing its type from the *filename*'s extension.

•virtual bool IsTypeSafe()

Returns whether the backend copies (safe) or just references the types passed to constructor function.

5.2 Object calls

•ObjectImpl *__impl() const

Retrieves the implementation of the object. Returned implementation is not cloned by the function itself. If one creates an object out of this pointer, it should first clone it.

•Object()

Default constructor. The implementation will be a special NULL, Udm::__null.

•Object(ObjectImpl *i)

Copy constructor, that takes an implementation and creates an object. Previous implementation, if not NULL, is released.

•Object(const Object &a)

Copy constructor, that takes an object and with its implementation creates a new object.
new implementation is cloned.

•const Object &operator =(const Object &a)

Assignment operator; takes an object and copies its implementation to the assigned object.
Previous implementation is released; new implementation is cloned.

•static Object Cast(const Object &a, const Uml::Class &meta)

Static cast function that casts Object *a* to type *meta*; It throws exception when types are unrelated, and thus the cast is invalid.

•static Object Create(const Uml::Class &meta,
const Object &parent,
const Uml::CompositionChildRole &role)

Static object factory function which creates an object of type *meta* with *parent*, via *role*.
Parent and meta parameters are mandatory. Role can be Udm::__null, if the containment is unambiguous.

•string getStringAttr(const Uml::Attribute &meta) const
•void setStringAttr(const Uml::Attribute &meta, const string &a)

•bool getBooleanAttr(const Uml::Attribute &meta) const
•void setBooleanAttr(const Uml::Attribute &meta, bool a)
•long getIntegerAttr(const Uml::Attribute &meta) const
•void setIntegerAttr(const Uml::Attribute &meta, long a)
•double getRealAttr(const Uml::Attribute &meta) const
•void setRealAttr(const Uml::Attribute &meta, double a)

Attribute set'ers and get'ers. Meta mandatory parameter specifies which attribute to set or to get.

•set<Object> getAssociation(const Uml::AssociationRole &meta) const

Retrieves all the associated objects with a given role name. If no role specified, an empty set is returned.
If this function is invoked on one end of the association with association classes, the association classes are returned.

•void setAssociation(const Uml::AssociationRole &meta, const set<Object> &a)

Resets the associated objects with a given role name to the new set a;

•const Uml::Class &type() const;

Retrieves the type of an object.

• set<Udm::Object> derived() const;

Retrieves the derived objects (subtypes) of an object.
For each class is generated a Derived() function which returns objects of the same type.

•set<Udm::Object> instances() const;

Retrieves the derived objects (subtypes) of an object.
For each class is generated an Instances() function which returns objects of the same type.

•Udm::Object archetype() const;

Retrieves the archetype object of an object.
For eacg class is generated an Archetype() function which returns objects of the same type.

•bool hasRealArchetype() const;

Returns true if the object is primary derived. Returns false if is not. Throws exception if it's not derived at all.

•bool operator ==(const Object &a) const
•bool operator !=(const Object &a) const
•bool operator <(const Object &a) const
•bool operator >(const Object &a) const
•bool operator <=(const Object &a) const
•bool operator >=(const Object &a) const
•bool operator !() const
•operator bool() const

Operators that return a boolean value. The operations are actually evaluated on the unique Ids of the operands' implementations, and the result is returned.

5.3 ObjectImpl calls

•virtual ObjectImpl *clone();

Returns itself, and increment its reference counter.

•virtual void release();

Decrements the reference counter.

•virtual Udm::DataNetwork *__getdn();

Returns the data network this implementation belongs to.

•virtual const Uml::Class &type() const;

Returns the type of the ObjectImpl.

•virtual uniqueId_type uniqueId() const;

Returns the unique Id of the ObjectImpl.

•virtual string getStringAttr(const Uml::Attribute &meta) const;
•virtual void setStringAttr(const Uml::Attribute &meta, const string &a);
•virtual bool getBooleanAttr(const Uml::Attribute &meta) const;
•virtual void setBooleanAttr(const Uml::Attribute &meta, bool a);
•virtual long getIntegerAttr(const Uml::Attribute &meta) const;
•virtual void setIntegerAttr(const Uml::Attribute &meta, long a);
•virtual double getRealAttr(const Uml::Attribute &meta) const;
•virtual void setRealAttr(const Uml::Attribute &meta, double a);

Attribute setters and getters. Meta always specifies the attribute to operate.

•virtual ObjectImpl *getParent(const Uml::CompositionParentRole &role) const;

Retrieves the parent, if the role matches. If not, returns Udm::__null;
Note that in all back-ends an object can have only a single parent.

•virtual void setParent(ObjectImpl *a, const Uml::CompositionParentRole &role);

Sets the parent via the specified parent *role*. If the composition is not ambiguous, *role* can be omitted.

•virtual void detach();

Removes the object from the data network by removing all its associations and from its parent children.

•virtual vector<ObjectImpl*> getChildren(const Uml::CompositionChildRole &meta, const Uml::Class &cls) const;
Returns the children of type *cls* via child role *meta.*
Both of them can be omitted, in any combination.
If *meta* is omitted, all the children of type *cls* are returned.
If *cls* is omitted, all the children via role *meta* are returned.
If both parameters are omitted, all the children are returned.

•virtual void setChildren(const Uml::CompositionChildRole &meta, const vector<ObjectImpl*> &a);

Resets some subsets of children. Elements of vector *a* can be of different types.
Child role can be omitted when for all the objects in vector *a* the containment is unambiguous.
If child role is omitted, the function will deduce an unambiguous composition child role for each distinct type in vector *a,* All existing children via rolenames deduced this way are detached if they are not present in the new vector/

•virtual ObjectImpl *createChild(const Uml::CompositionChildRole &childrole, const Uml::Class &meta)

Creates a child of the given type and via the given child role. Child role can be omitted if the containment between these two type is unambiguous.

•virtual vector<ObjectImpl*> getAssociation(const Uml::AssociationRole &meta, int mode = Udm::TARGETFROMPEER) const;

Retrieves the associations via the given association role.
When there is an association with an association class, the mode parameter specifies which objects are to be returned:

TARGETFROMPEER	-	the other end of the association(default)
CLASSFROMTARGET	-	the association class (from either end)
TAGETFROMCLASS	-	either end from the association class

•virtual vector<Udm::ObjectImpl*> getDerived() const;

Retrieves the derived objects (subtypes) of an object.

•virtual vector<Udm::ObjectImpl*> getInstances() const;

Retrieves the derived objects (subtypes) of an object.

•virtual Udm::ObjectImpl* getArchetype() const;

Retrieves the archetype object of an object.

•bool hasRealArchetype() const;

Returns true if the object is primary derived. Returns false if is not. Throws exception if it's not derived at all.

- virtual void setAssociation(const Uml::AssociationRole &meta, const vector<ObjectImpl*> &nvect, int mode = Udm::TARGETFROMPEER);

  Resets the associated objects via the provided association role. The role can't be omitted.
  Mode has the same meaning as above, except that TARGETFROMPEER is invalid when setting up an association with association class.

5.4 UML (Meta) calls

- const AssociationRole theOther(const AssociationRole &role);
- const CompositionChildRole theOther(const CompositionParentRole &role);
- const CompositionParentRole theOther(const CompositionChildRole &role);

  Gets the other end of two-legged Uml classes: Composition, Association

- Class classByName(const Diagram &d, const string &name);

  Finds a class by its name in a diagram.

- set<Class> AncestorClasses(const Class &c);

  Get all the classes specified as ancestors, including self

- set<Class> DescendantClasses(const Class &c);

  Get all the classes specified as descendants, including self

- set<Class> ContainerClasses(const Class &c);

  Get classes directly specified for this class as parents (both ancestors and descendants are ignored)

- set<Class> ContainedClasses(const Class &c);

  Get classes directly specified for this class as children (both ancestors and descendants are ignored)

- set<Class> CommonAncestorClasses(const set<Class> &cs);

  Get classes that are ancestors of all the classes in a set

- set<Class> AncestorContainerClasses(const Class &c);

  Get classes that are ancestors of all the classes in a set

- set<Class> AncestorContainedClasses(const Class &c);

  Get classes this object or its ancestors specify as parents (ancestors in child are extracted, descendants of parents are ignored)

- set<Class> AncestorContainedDescendantClasses(const Class &c);

  Get classes this object or its ancestors specify as children (ancestors in parent are extracted, descendants of children are ignored)

- set<AssociationRole> AssociationTargetRoles(const Class &c);

  All the other ends of associations (ancestors are ignored)

- set<AssociationRole> AncestorAssociationTargetRoles(const Class &c);

  All the other ends of associations this class can have (including those defined in ancestors)

- set<AssociationRole> AncestorAssociationRoles(const Class &c);

All local ends of associations this class can have (including those defined in ancestors)

- set<CompositionParentRole> CompositionPeerParentRoles(const Class &c);

  All the other ends of compositions defined for this class as child (ancestors are ignored)

- set<CompositionParentRole> AncestorCompositionPeerParentRoles(const Class &c);

  All the parent ends of compositions this class can participate in (including those defined for ancestors)

- set<Attribute> AncestorAttributes(const Class &c);

  All attributes this class can have (including those defined in ancestors)

- Composition matchChildToParent(Class c, Class p);

  Find the single way a class can be contained by another, return NULL none or if multiple roles are found.

- bool IsDerivedFrom(const Class &derived, const Class &base);

  Returns true if derived derives from base.

- bool IsAssocClass(const Class &cl);

  Returns true if class cl is an association class.

- bool IsAssocClass(const Association &ass);

  Returns true if the association belongs to an association class.

5.5 Miscellaneous calls

- Void UdmUtil:: CopyObjectHierarchy(Udm::ObjectImpl* p_srcRoot, Udm::ObjectImpl* p_dstRoot, Udm::DataNetwork* p_dstBackend);

  Utility that copies a sub-tree from a data network to another data network.
  Consistent (same) meta information is assumed in both data networks. If there are links pointing out from the sub tree of the source data network, exception is thrown.

- int CopyObjectHierarchy(Udm::ObjectImpl* p_srcRoot, Udm::ObjectImpl* p_dstRoot, Udm::DataNetwork* p_dstBackend, copy_assoc_map &cam)

  Same function, but it provides a mapping of some source objects (those inserted by the caller in the map) to their respective destination objects.

- String UdmUtil:: ExtractName(Udm::Object ob);

  This function will search for an attribute named "name" and it found, it will extract its name.

- Static const Uml::Class& Uml::SafeTypeContainer::GetSafeType(const Uml::Class &a);

  Static function to obtain a type object which won't be deleted until released with RemoveSafeType().

- Static void Uml::SafeTypeContainer::RemoveSafeType(const Uml::Class &a);

  Release a safe type object.

5.6 UDM TOMI interface

These calls are defined as member functions of the class Udm::Object

- Structure to identify an association:

```
struct AssociationInfo
{
        // clsAssociation can be empty (UML::Class()) - simple association.
        const Uml::Class & clsAssociation;
        string strSrcRoleName;
        string strDstRoleName;
};
```

- Structure to identify a composition:

```
struct CompositionInfo
{
        string strParentRoleName;
        string strChildRoleName;
};
```

- Retrieves the adjacent objects of an object associated via simple association or association class.
  The returned set can be empty. Composition relationships are not considered here.

```
set<Object> GetAdjacentObjects();
```

- Retrieves the adjacent objects of an object. The adjacent objects are of the type of clsType or derived from it. The returned set can be empty. Composition relationships are not considered here.

```
set<Object> GetAdjacentObjects(const Uml::Class & clsDstType);
```

- Retrieves the adjacent objects of an object via link instance of ascType.
  The adjacent objects are of the type of clsType or derived from it.
  The returned set can be empty. Composition relationships are not considered here.
  Parameter clsType can be null.

```
set<Object> GetAdjacentObjects(const Uml::Class & clsDstType, const AssociationInfo& ascType);
```

- Get attributes by name. These are INEFFICIENT functions using iteration. These functions return false if the attribute with the specified type and attribute name does not exist. Hence these parameters are specified in the metamodel, it can be serious error. If no problem they retrieve true.

```
bool GetIntValue(string strAttrName, int& value);
bool GetStrValue( string strAttrName, string& value);
bool GetRealValue( string strAttrName, double& value);
bool GetBoolValue( string strAttrName, bool& value);
virtual bool SetIntValue( string strAttrName, int value);
virtual bool SetStrValue(string strAttrName, const string& value);
virtual bool SetRealValue( string strAttrName, double value);
virtual bool SetBoolValue( string strAttrName, bool value);
```

- Returns the parent object.

```
Object GetParent();
```

- Retrieves all children not considering types and role names.

```
set<Object> GetChildObjects(Object object);
```

- Retrieves all children considering child types but not role names.

```
set<Object> GetChildObjects(const Uml::Class & clsType);
```

- Retrieves all children considering role names and child types. To ignore child types set clsChildType to Uml::Class().

```
set<Object> GetChildObjects(const CompositionInfo& cmpType, const Uml::Class & clsChildType);
```

- Gets the objects of the association class from between two objects.

```
set<Object> GetAssociationClassObjects(Object dstObject, const AssociationInfo& ascType);
```

- Gets the two peers from an object of association class type

```
pair<Object,Object> GetPeersFromAssociationClassObject();
```

- Creates an object of clsType

```
Object CreateObject(const Uml::Class & clsType);
```

- Creates a link of a simple association or an association with association class. If ascType.clsAssociation is not valid a simple association will be tried. On error results in false, true otherwise.

```
bool CreateLink(Object dstObject, const AssociationInfo& ascType);
```

- Removes an object from the persistent storage.

```
void DeleteObject();
```

6.Interpretted UDM

6.1 Udm Cint

UdmCint is a runtime, interactive ANSI C++ interpretter which on-the-fly compiles and evaluates arbitrary code on UDM objects. The language is about 85% coverage of ANSI C++. The standard C library and the standard template library (STL) are supported. For more information about Cint please see the Cint documentation, which is included in the binary distribution.

The UdmCint framework consists of:

- **UdmCint API**, which is a set of calls which can be used in compiled context to set up an interpretter context.

- **UdmPseudoInterface API** , which is a set of classes that can be used in both the compiled and the interpreted context. Thus, the data structures defined here are used to exchange data between the intepreted and compiled context.

6.1.1 The Udm Cint API:

The Udm Cint API is defined in the header file UdmInt.h, which must be included when this functionality is needed.

```
•void cint_init(char * cint_switches, void (*msg_handler)(char *));

          Initializes the Cint interpeter engine.

          cint_switches  –switches that need to be passed to the engine
          msg_handler    -a call-back function which can handle error and warning
                         messages

•void cint_destroy();

          Clears the Cint interpreter context, and all the memory and resources
          allocated for it. After this call, the interperter shouldn't be used.

•void cint_add_object(Udm::Object o, string name);

          Defines a UdmPseudoObject in the interpreted context, in the current
          scope. Basically, this call will place a single line which declares a
          UdmPseudoObject with the given name, and it will initialize it with the
          proper object and datanetwork ids of the object o. Thus, the scope of this
          variable will depend on the current context of the interpreter. If it's
          outside of any block, it will have a global scope, if it's issued in a
          block, than it will have a local scope, which is the current block.

          The only way to remove a global variable added with this call is to
          destroy and re-init the interpreter.

•void cint_ipretter();

          Starts an interactive Udm Cint session. Arbitrary code can be executed
          containing declaration, conditional statements, control loops, etc.
          By hitting <ENTER> at the end of the line the code will be executed,
          unless there are unbalanced begin block '{' characters. In such cases it
```

```
            will wait for further input lines until the block is closed with the
            balancing '}' characters, and then the block will be evaluated. When
            entering a block in multiple lines, the prompt always changes accordingly,
            displaying an  extra '{' character for each unbalanced '{'. When
            evaluation occurs, the return value of the expression is printed.

•int cint_int_calc(string code);
•double cint_double_calc(string code);

            Evaluates the expression, which can not contain declaration or conditional
            statements. Depending on the expected type of the return value of the
            expression, the more appropiate function should be used.

•int cint_int_eval(string code);
•double cint_double_eval(string code);

            Evaluates the expression, which can contain declaration or conditional
            statements. However, it can not contain function defintions. Depending on
            the expected type of the return value of the expression, the more
            appropiate function should be used.

•char* cint_load_text(const char *namedmacro);

            cint_load_text()loads null terminated string as source code. It can
            contain any kind of C/C++ syntax that Cint supports. The text is once
            saved into a temporary file, then read by Cint. The tempoprary file is
            automatically removed when it is unloaded.

            cint_load_text() returns name of the temporary file in a static buffer.
            You can use that name for unloading.  In an event of failure,
            cint_load_text() returns 0.

•int cint_loadfile(const char *filename);

            cint_loadfile() loads source code or DLL(Dynamic Link Library) at runtime.
            This is just a wrapper function. Refer to the G__loadfile() Cint API call
            documentation.

•int cint_unloadfile(const char *filename);

            cint_unloadfile() unloads a loaded file at runtime. This is just a wrapper
            function. Refer to the G__unloadfile() Cint API call documentation.
```

6.2.2 The UdmPseudoInterface API:

Basically, the most important class defined by this API is the class ***UdmPseudoObject***, which is a wrapper of the Udm::Object class, and can be initialized with an object and a data network Id. The class ***UdmPseudoObjectS*** represents an unordered collection of ***UdmPseudoObject***-s. Strings are passed by reference, as objects of type ***cint_string***, class which is also defined by this API. Two more structs are defined, ***AssociationInfo*** and ***CompositionInfo***, which are basically the very same thing as the structs introduced by the TOMI interface, *Udm::Object::AssociationInfo* and *Udm::Object::CompositionInfo*, the difference is the type of the members, while their semantic remains the same.

This API is defined in UdmCint.h. The classes defined here are automatically known to the interpreter context, so it does not have to be included in the beginning of the interpreted code. Even if the intended use of this API is to be used in interpreted context, however, as the examples will show, they can be used in compiled context as well. In this

particular situation, the include file UdmCint.h needs to be included in the compiled code.

In dependency order:

class cint_string

cint_string objects are meant to be created in interpreter context, and on the stack. This way, there are no memory allocation/deallocation and buffer overflow issues between the interpreted and the precompiled code; The allocation always occurs when the variable is declared, with a predefined size. The cint_string always knows how many bytes were allocated, and will refuse copying more bytes to the buffer. The deallocation will occur when the variable (in the interpreter's context) goes out of scope or (in case of global variables) the interpreter is destroyed, or, with the *operator=(cint_string &frm)* operator an other cint_string is assigned.

cint_string();

creates a NULL string, no allocation occurs.
This instance is not able to hold any characters.

cint_string(int length);

creates a new string which can accommodate *length* characters.
However, the value of the string is undefined at this stage.

cint_string(const char * frm);

creates a new string out from the NULL terminated *frm*. The length will be equal to strlen(frm). The string is copied, so *frm* can be released.

cint_string& operator=(const cint_string& frm);

assigns a new string to *this*. If *this* is not a NULL string, then it's buffer is freed first. A new buffer is allocated, and a copy of *frm*'s buffer is stored, so *frm* can go out of scope.

bool CopyFrom(const char * frm);

copies a string into *this*'s buffer. *This* is expected to be a not-NULL string. (i.e. constructed with either with *cint_string(int length)* or *cint_string(const char * frm)*). If *frm* is longer than the buffer of *this*, then only that count of characters are copied which the buffer can accommodate. If the size of *frm* is less then the length of the buffer, then the buffer is padded with null bytes. (*strncpy* behaviour). Returns with the value of the *overflow* member variable.

bool operator !() const;

returns true if the string is a NULL string. (no currently allocated buffer)

operator bool() const;

returns true if the string is not a NULL string. (there is a currently allocated buffer)

int comp(const char * compare_to) const;

strcmp() functionality.

bool overflow;

public member variable which is set to true by the *CopyFrom()* function, if the string to be copied was larger than the buffer.

const char * buffer() const;

returns a pointer to the internal buffer. It should not be freed by the caller.

struct AssociationInfo

this struct has the same semantics as *Udm::Object::AssociationInfo* and is used to call TOMI functions from the interpreted context.

cint_string assoc_class;
the name of the association class, if any. It can be a NULL cint_string, which means that the structs represents an association w/o an association class.

cint_string SrcRolename;
cint_string DstRolename;

the name of the roles. Cannot be a NULL cint_string.

AssociationInfo(const char *src, const char *dst, const char *assoc);
AssociationInfo(cint_string src, cint_string dst, cint_string assoc);

Constructors just for convenience.

struct CompositionInfo

this struct has the same semantics as *Udm::Object::CompostionInfo* and is used to call TOMI functions from the interpreted context.

cint_string parentRole;
cint_string childRole;

the name of the roles. Cannot be a NULL cint_string.

CompositionInfo(const char *pr, const char *cr);
CompositionInfo(cint_string pr, cint_string cr);

Constructors for convenience.

class UdmPseudoObjectS

this is an unordered container of *UdmPseudoObject*-s, a class which is defined right after this one in this document.

This class is meant to be instantiated on the stack, in the interpreted context, with a predefined size parameter. References to objects created this way then can be passed to those methods of class *UdmPseudoObject*, which are supposed to return a set of *UdmPseudoObject*-s.

UdmPseudoObjectS(int length);

A buffer is allocated which can hold *length* number of *UdmPseudoObject*-s. If length is zero, no allocation occurs.

UdmPseudoObjectS(const UdmPseudoObjectS &frm);

A buffer is allocated and all *UdmPseudoObject*-s are copied from *frm*, if any.

UdmPseudoObjectS& operator=(const UdmPseudoObjectS &frm);

Current buffer is freed. A new buffer is allocated and all *UdmPseudoObject*-s are copied from *frm,* if any.

bool operator !() const;

returns true if the container does not contain any *UdmPseudoObject*.

operator bool() const;

returns true if the container contains at least one *UdmPseudoObject.*

UdmPseudoObject& operator[](const int index) const;

returns a reference to the *UdmPseudoObject* at the zero-based *index* position in the container. If the *index* is beyond the length of the container, the *overflow* public bool variable is set to true and the *UdmPseudoObject* at the last position is being returned.

void SetAt(const int index, UdmPseudoObject& item);

copies *item* to the *UdmPseudoObject* at the zero-based *index* position in the container. If the *index* is beyond the length of the container, the *overflow* public bool variable is set to true and the *UdmPseudoObject* at the last position is being set.

bool overflow;

indicates an index overflow during *SetAt()* and *[]* operations.

int GetLength() const;

returns the length of the container.

class UdmPseudoObject

this class is a wrapper class for *Udm::Object*. Basically, it has a corresponding method for all the Udm::Object methods, the Udm::ObjectImpl methods. These methods delegate the call to the corresponding *Udm::Object* or *Udm::ObjectImpl* method, after doing some trivial transformation between different types.

UdmPseudoObject();

constructs a NULL object

UdmPseudoObject(unsigned long dn_id, unsigned long ob_id);

constructs an object with the given data network and object id

UdmPseudoObject& operator=(const UdmPseudoObject& frm);

assignment operator.

static bool GetLastError(cint_string& buffer);

retrieves the last error, if any. The result is stored in *buffer.*
It should be checked after any method call on this object that returned false. The return value is true, if there was an error, and in this case the description is copied to *buffer*. Invoking this method clears the error condition. Only the last error can be retrieved.

bool type(cint_string &buffer) const;

stores the name of the object's type in *buffer*. Returns false, if there was an error.

bool uniqueId(unsigned long &id) const;

stores the uniqueId of the object in *id.* Returns false, if there was an error.

bool getParent(const cint_string &prole, UdmPseudoObject &value) const;

stores a UdmPseudoObject in *value* which is bound to the parent object. *prole* is the name of the parentrole, which can be NULL. The method behaves the same way as *ObjectImpl::getParent()* does.
Returns false, if there was an error.

bool setParent(UdmPseudoObject &parent, const cint_string &prole);

set a the object bound to *parent* as parent for the object.

prole is the name of the parentrole, which can be NULL. The method behaves the same way as *ObjectImpl::setParent()* does.
Returns false, if there was an error.

bool detach();

removes the object from the object network.
Behaves the same was as *ObjectImpl::detach()* does.
Returns false, if there was an error.

bool getChildren(const cint_string &crole, const cint_string &kind, UdmPseudoObjectS &value) const;

stores UdmPseudoObjects bound to the children in *value.*
crole is the name of the child role, which can be NULL.
Kind is the name of the *Uml::Class* type of children to be returned, which also can be NULL.
Behaves the same way as *ObjectImpl::getChildren()* does.
If *crole* and/or *kind are* NULL, *ObjectImpl::getChildren()* will be called accordingly, with NULL CompositionChildRole or NULL Class, respectively.
Returns false, if there was an error.

bool setChildren(const cint_string &crole, const UdmPseudoObjectS &value);

sets the object's children via role *crole* to the objects bound to UdmPseudoObjects in *value.*
crole is the name of the child role, which can be NULL.
Behaves the same way as *ObjectImpl::setChildren()* does.
If the *crole* is NULL, then *ObjectImpl::setChildren()* will be invoked accordingly, with a NULL CompositionChildRole.
Returns false, if there was an error.

static const TARGETFROMPEER = 0;
static const TARGETFROMCLASS = 1;
static const CLASSFROMTARGET = 2;

static constants which can also be found with exactly these values in the *Udm* namespace.

bool getAssociation(const cint_string &ass_role, UdmPseudoObjectS &value, int mode = TARGETFROMPEER) const;

stores UdmPseudoObjects bound to the associations via role with name *ass_role* in *value.*
ass_role is the name of the association role, which cannot be NULL.
Behaves the same way as *ObjectImpl::getAssociation()* does.
Returns false, if there was an error.

bool setAssociation(const cint_string &ass_role, const UdmPseudoObjectS &value, int mode = TARGETFROMPEER);

sets the associated objects via role *ass_role* to the objects bound to UdmPseudoObjects in *value.*
ass_role is the name of the association role, which cannot be NULL.
Behaves the same way as *ObjectImpl::setAssociation()* does.
Returns false, if there was an error.

bool SetIntVal(const cint_string &name, long value);
bool SetRealVal(const cint_string &name, double value);
bool SetStrVal(const cint_string &name, const cint_string & value);
bool SetBoolVal(const cint_string &name, bool value);

attribute setter functions.
name is the name of the attribute to be set, which can not be NULL.
Return false if there was an error. If false is returned, and *GetLastError()* is also false, then the attribute with that name and type does not exist.

bool GetIntVal(const cint_string &name, long &value);
bool GetRealVal(const cint_string &name, double &value);
bool GetStrVal(const cint_string &name, cint_string & value);
bool GetBoolVal(const cint_string &name, bool &value);

attribute getter functions.
name is the name of the attribute to be set, which can not be NULL.
Return false if there was an error. If false is returned, and *GetLastError()* is also false, then the attribute with that name and type does not exist.
GetStrVal() uses *cint_string::CopyFrom()* to store the string in *value*.

bool GetAdjacentObjects(UdmPseudoObjectS &value);
bool GetAdjacentObjects(const cint_string &dst_type, UdmPseudoObjectS &value);
bool GetAdjacentObjects(const cint_string &dst_type, const AssociationInfo & ass, UdmPseudoObjectS & value);
bool GetParent(UdmPseudoObject &value);
bool GetChildObjects(const cint_string & type, UdmPseudoObjectS &value);
bool GetChildObjects(const CompositionInfo &ci, const cint_string & type, UdmPseudoObjectS &value);
bool GetAssociationClassObjects(const UdmPseudoObject &dstObject, const AssociationInfo & ai, UdmPseudoObjectS &value);
bool GetPeersFromAssociationClassObject(UdmPseudoObjectS &value);
bool CreateObject(const cint_string &type, UdmPseudoObject &value);
bool CreateLink(UdmPseudoObject &dst, const AssociationInfo& ass_type);
bool DeleteObject();

the wrapping functions of TOMI API.

Overview of differencies:
Return value:

same as the return value of the corresponding TOMI call, unless the *Udm::Object* could not be found based on the Ids. In this case the TOMI method is not invoked, false is returned, and an error condition is set, *GetLastError()* returns true and gives a description of the error.

cint_string instead of *Uml::Class* argument types:

in these wrapper functions types are identified with a *cint_string* parameter with the name of the class instead of *Uml::Class*. The real *Uml::Class* type parameter is obtained via a *Uml::findClassByName()* on the set of classes of the meta diagram.

Example code using the UdmCint API:

code which creates a *UdmPseudoObject* in interpreted context, and invokes a method on it:

```
int main(int argc, char *argv[])
{
	Udm::SmartDataNetwork nw(diagram);
	nw.CreateNew(argv[1],"LampDiagram", RootFolder.meta);
	{
		RootFolder rrr = RootFolder::Cast(nw.GetRootObject());
		Lamp lamp = Lamp::Create(rrr);
		lamp.name() = "Host Lamp ";
		lamp.ModelName() = "Tester lamp";

		cint_init(NULL, myDisplay);
		cint_add_object(lamp,"host_lamp");

		string code = "host_lamp.SetStrVal(\"name\",\"cint changes the name\");";
		cint_ipretter();
		cint_int_calc(code);
	}
}
```

code which creates a *UdmPseudoObject* in compiled context, and invokes a method on it in interpreted context(UdmCint.h must be included):

```
int main(int argc, char *argv[])
{
	Udm::SmartDataNetwork nw(diagram);
	nw.CreateNew(argv[1],"LampDiagram", RootFolder.meta);
	{
		RootFolder rrr = RootFolder::Cast(nw.GetRootObject());
		Lamp lamp = Lamp::Create(rrr);
		lamp.name() = "Host Lamp ";
		lamp.ModelName() = "Tester lamp";

		cint_init(NULL, myDisplay);
		UdmPseudoObject pseudo_lamp(dn_id, o_id);

		char code[200];
		sprintf(code, "UdmPseudoObject * p_pseudo_lamp = (UdmPseudoObject *) %ld;\n", &pseudo_lamp);
		cint_int_eval(code);
		sprintf(code, "UdmPseudoObject pseudo_lamp = *p_pseudo_lamp;\n");
		cint_int_eval(code);
		sprintf(code,"pseudo_lamp.SetStrVal(\"name\",\"cint changes the name!\");");
		cint_int_eval(code);
	}
}
```

Appendix A

UDM tools documentation

VisioUML2XML

```
NAME
      VisioUML2XML - convert Visio UML class diagrams into Udm XML-UML
representation

AUTHOR
      Jason Garrett  - garrettjt@vuse.vanderbilt.edu
      Arpad Bakay    - arpad.bakay@vanderbilt.edu

SYNOPSIS
      VisioUML2XML [options] [<inputfile>] [<outputfile>]

      <inputfile>             The path to a Visio diagram
      <outputfile>            The path to the output XML
                              (if <inputfile> has '.vsd' extension,
                              output filename defaults to that name
                              with extension changed to '.xml')


      Options:
      -s -silent              Silent mode: do not generate output
                              unless a failure occurs
                              In silent mode at least <inputfile> needs
                              to be specified
      -n -namediag <name>     Specify <name> for the diagram
                              (if <inputfile> has '.vsd' extension,
                              diagram name defaults to the base portion
                              of that name)
      -? -h -help <name>      Display help

      (in addition to '-',  '/' is also considered as option prefix)


DESCRIPTION

      VisioUML2XML converts Microsoft Visio 2000 UML class diagrams to
an XML representation conformant to UML.dtd (see files)
The program starts up Visio and accesses its data through its
Automation interface.

      The program can be run in two modes. In silent mode, the files
and names to work with are specified on the command line as described
above.
```

In GUI mode, a dialog box is presented, which allows setting or changing the options, and control the steps of the conversion individually. The EXPRESS command executes the whole sequence, and in its effect it is similar to the silent mode operation.

The program can handle diagrams with multiple sheets. ‘Enable sheet resolution’ is an option (default on, accessible only from the GUI), to merge identically named classes on separate sheets in the document. Since it is normally a useful feature, the availability as an option has mostly historical reasons.

The program has been tested with Visio 2000 SR1

TIPS AND TRICKS

To create class diagrams for processing by VisisoUML2XML, the “Uml Static Structure” section of the “UML Model Diagram” template is to be used. Due to the use of undocumented functions, probably only Visio2000 can be used, no earlier or later versions.

The conversion tool only processes the following stencils: Class, Binary Association, Association Class, Composition, and Generalization. Others are ignored, and relationships to them result in errors.

In order to enable the conversion tool to understand certain settings, they must be made visible through the ‘Shape Display Options’ settings in the context menu. The settings that may need to be applied in addition to the default ones are: Name (for associations), Attribute Multiplicities, End Navigability.

Association classes can only have straight association lines, so it is not feasible to represent self-associations with classes in an elegant way. The workaround provided for such situations is the following:

1. Create simple, non-stencil auxiliary lines (accessible through the toolbar) to roughly connect the ends of the association and the target classes.
2. Create incoming endpoints on these lines on the end close to the association, and outgoing endpoints at the ends close to the target class.
3. Draw the line ends until the association ends snap onto the ends of the auxiliary lines, and the aux lines snap onto the target class.
4. You may use a sequence of multiple auxiliary lines, but make sure they are all connected with outgoing ends toward the target classes.
5. Unfortunately, the association navigability indicators and end names remain to be displayed at the ends of the original association line.

Theoretically, navigability should be indicated on every relationship in a diagram. Since by default this is not displayed, as a convenience VisioUML2XML provides two options to assume navigability even if arrows are not displayed. These checkboxes are ‘Regard named associations as

navigable’ (for associations with a name), ‘Compositions are always navigable’ (in both directions, regardless of the presence of names).

Navigability determines if UDM generates methods for accessing these relationships. If navigability is set, but there is no name, a name is automatically generated from the target class name. See paragraph 2.2 in the UDM API documentation for details. Automatically generated names may collide, which can only be corrected by specifying names on relationship ends.

The only recorded aspect of attribute multiplicities is whether it is allowed to have 0 values. In this case, the attribute is generated as ‘optional’. The upper limit of attribute multiplicity is always considered to be 1.

Relationship end names specify the names of the roles, while the name of the relationships (visible on the center) specifies the name of the Association or Composition itself. If the name is not visible, the relationship has no name recorded, which is valid. Although not yet used by UDM, the direction information, if present and visible, is also recorded in the XML file. Care should be taken since direction arrows may easily point in the wrong direction: a forward arrow designates that the primary direction is from the first end to the second, (vice versa for the backward arrow). This always applies, even if the second end is on the left, and the first end is on the right, when the direction arrow symbol suggests just the opposite.

LIMITATIONS

The following restrictions apply, mostly due to the rudimentary features of Visio automation interface:

- When the program starts up Visio, it comes up as an application and may hide the VisioUML2XML dialog.
- Although no update operations are initiated, sometimes Visio automatically modifies the input file (e.g. if it is created by a previous version) and thus brings up a confirmation dialog box when the file is about to be closed.
- The program is rather slow.

FILES

uml.dtd :	the generated XML is compliant with this DTD. The standard location of this file is in the /documents subdirectory of the distribution

Last change: 11 Oct 2001

Udm

NAME

Udm - generate API and DTD from UML class Diagram representation

AUTHOR

Miklos Maroti - mmaroti@math.vanderbilt.edu
Arpad Bakay - arpad.bakay@vanderbilt.edu
Endre Magyari - endre.magyari@vanderbilt.edu

SYNOPSIS

Udm <inputname> [<namebase>][–t <tempdiagram>] [-d <DTD path>] -m

<inputname>	The name of the input XML document (Conformant to the uml.dtd file)
<namebase>	The basename of the output documents (.h, .cpp, .dtd), and also the namespace the API is defined in.
-m	A special .cpp format is generated, which creates a static data network which does not require meta information. Uml.cpp, the meta-meta model should be generated with this switch.
-d	the path where the DTD file are searched. If not provided, the PATH is searched if a DTD is needed.
-c	A special .CPP format is generated, which instead of creating the meta-objects as static data network, creates a data network containing objects wrapping CORBA proxies of already existing meta-objects in a remote server.
-s	[Experimental] C# code generation for the Microsoft .Net platform. It can be used together with the –m switch.

DESCRIPTION

Udm generates API and .dtd files from an XML representation of a UML diagram. These XML-s are usually extracted from some UML tool like Visio (using VisioUML2XML), or GME-UML (using the associated interpreter)

The API files are necessary for compiling UDM applications, as described in the ‘Creating a C++ project using UDM’ section of the UDM documentation.

FILES

uml.dtd :	the input XML files are compliant with this DTD. Udm contains a copy of this file as an attached resource. The ‘-d’ option can be used to specify a different DTD file.

Last change: 28 Apr 2001

UdmBackendDump

```
NAME
      UdmBackendDump - generate a human readable textual representation
Of an object network regardless of its meta-model

AUTHOR
      Tihamer Levendovszky- tihamer.levendovszky@vanderbilt.edu

SYNOPSIS
      UdmBackendDump <backend_file.mga>|<backend_file.xml>
<backend_meta_file.xml>

      <backend_file.mga>      The name of the GME DataNetwork file

      <backend_file.xml>      The name of the DOM DataNetwork file

      <backend_meta_file.xml> The name of the DOM DataNetwork XML-
Metafile




DESCRIPTION
      UdmBackendDump generates a textual description of an object
network. It dumps out all the meta(classes, associations, compostions)
and instance(objects, links) entities in a textual representation.
```

Last change: 08 Jun 2002

UdmViz

NAME

UdmViz - generates a textual representation from a Udm backend file to the standard output that serves as an input of the dot.exe from ATT GraphViz graph displaying utility (it can be downloaded from http://www.research.att.com/sw/tools/graphviz/download.html).

AUTHOR

Tihamer Levendovszky- tihamer.levendovszky@vanderbilt.edu

SYNOPSIS

UdmViz <backend_file.mga>|<backend_file.xml> <backend_meta_file.xml> <flags>

<backend_file.mga> The name of the GME DataNetwork file

<backend_file.xml> The name of the DOM DataNetwork file

<backend_meta_file.xml> The name of the DOM DataNetwork XML-Metafile

<flags> [-<a>|<l><al>] –a: aggregate hierarchy –l: links, role names –al: both. Default(if flags are omitted): –l.

DESCRIPTION

UdmViz generates a textual description of an object network for an ATT GraphViz digestible format to the standard output. This is another way to display an object network from a Udm backend.

SAMPLE

Exporting the aggregation hierarchy from model.xml to a GIF picture:

```
UdmViz model.xml meta.xml –a > x.dot
dot -Tgif x.dot > x.gif
```

Last change: 08 Jun 2002

UdmPat

NAME

UdmPat – process Udm data to produce text output as defined in a simple pattern script.

AUTHOR

Arpad Bakay - arpad.bakay@vanderbilt.edu

SYNOPSIS

UdmPat <indata> <diagram> <patternfile>

<indata>: input Udm data. If the file extension is none of the well-known values (e.g. '.mga' or '.xml'), the prefixes 'GME:' or 'DOM:' are used to indicate the backend type. E.g.:
DOM:<domfilename> : open as DOM data (i.e. an XML file)
GME:<projectconnstr> : open as GME data
name_without_prefix.ext : guess type based on extension
<diagram>: the UML-XML file that was used to create the data
<patternfile>: the name of the pattern script

DESCRIPTION

UdmPat reads Udm data while processing a pattern script to generate output to one or several files. The pattern file may contain plain text, which is copied to the output verbatim, and special pattern instructions (funtions like $!EVAL_FORALL and macros like $Name), that are evaluated, and their result is inserted into the output. Pattern instructions have arguments. A significant group of arguments are text arguments, which can again contain any mixture of plain text or pattern instructions.

The most relevant instructions are the ones that retrieve information from the input Udm data. It is possible to access attributes (e.g. $Name), and iterate through associations and containment relationships
(e.g. EVAL_FORALL(EmergencySwitch, "..."), to locate and query objects.

UdmPat always accesses Udm data in read-only mode.

UdmPat is a generic application, i.e. the meta information (class, attribute, relationship definitions) on the data is not provided runtime, but compile-time, in form of an UML XML file, specified by the <diagram> argument.

The pattern is always evaluated in the context of an Udm object, which is used when evaluating pattern instructions. Some pattern instructions ($!EVAL_WITH, and $!EVAL_FORALL) change the context when evaluating their text arguments.

Data printed before an output file is specified is sent to the standard output. Therefore, the first non-comment pattern is typically $!TO_FILE() command, that specifies the output location.

$<varname>
Evaluate variable or attribute <varname> and return its value in the output.

First, the set of variables is searched for <varname> (variables are defined by $!DEFINE, see below)
Next if <varname> is “name”, return the name of the current object
Otherwise read and return current object's the attribute identified by <varname>
If none is defined, report error.

$!EVAL_FORALL(<fieldspec>, <textarg>)
Get any number (including 0) of objects identified by fieldspec iterate through them,
and evaluate arg2 in their context.
<*fieldspec*> may contain one of the following specifications containing access qualifiers:

ChildRole:<rolename>	the children of this childrole
ChildType:<kindname>	the children of this kind
Parent:	the parent of this object if any
Type:	The Uml::Class object that represents the type (metainformation)
AssocPeer:<arolename>	the peer objects of associations of the specified role
AssocClass:<arolename>	the association classes of the specified role

If omitting the above access qualifiers (e.g. ChildType:, AssocPeer:) does not cause ambiguity, the role or kind names (or the keywords ‘parent’ and ‘type’) can also be used alone.

$!EVAL_WITH(<fieldspec>, <textarg>)
Get the single object identified by fieldspec and evaluate arg2 in its context, return the result
See above the specification of <fieldspec>

$!IFEMPTY(<textarg1>, <textarg2>)
If arg1 evaluates to an empty string, evaluate arg2 and return the result

$!TO_FILE(<arg>)
Close current output file, and start writing to the file specified by arg

$!DEFINE(<varname>, <textarg>)
Create or redefine the variable identified by varname to contain the current value of textarg

$!POSTINCR(<varname>)
Increment variable varname by 1, and return its original value (its value must be legible as a number)

$!SEQ(<textarg1>, <textarg2>, ...)
Evaluate textargs one by one and append their result

$!PAD(<width>, <textarg>)
Make sure the output of textarg is at least ABS(<width>) characters

wide. The string is left-justified if <width> > 0, and right justified if < 0

$!COMMENT(<arg>) <arg> is a comment. No output is generated

Separating verbatim text and pattern instructions

The text of the pattern file is copied verbatim to the output, unless a pattern instruction (beginning with $) is detected. A pattern instruction must fit into a single line, but physical lines can be merged into a single logical one by specifying \ as the last character of the first line. ‘$’ and ‘\’ characters are produced using ‘\$’ and ‘\\’ respectively.

Text arguments within functions are evaluated in a similar fashion, but
1. they may be surrounded by double quotes (“ “) to indicate the borders of the string. Otherwise, a string is terminated by the first unquoted ‘,’ or ‘)’
2. ‘\t’ and ‘\n’ is converted to tab and newline characters in the generated output.

FILES

uml.dtd :	the <diagram> file must be compliant with this DTD. UdmPat contains a copy of this file as an attached resource.

Last update: 20 December 2001

UdmCopy

NAME

UdmCopy - replicate a data network in another persistence engine.

AUTHOR

Arpad Bakay - arpad.bakay@vanderbilt.edu

SYNOPSIS

UdmCopy <indata> <outdata> <diagram> [<metalocator>]

<indata>	the name of the input data (filename or GME connection string). If the file extension is none of the well-known values (e.g. '.mga' or '.xml'), the prefixes 'GME:' or 'DOM:' are used to indicate the backend type.
<outdata>	the name of the output data (filename or GME connection string). If the file extension is none of the well-known values (e.g. '.mga' or '.xml'), the prefixes 'GME:' or 'DOM:' are used to indicate the backend type.
<diagram>	the XML representation of the UML diagram that describes the data structure to be copied.
<metalocator>	the name of the backend technology-specific meta-information the new datanetwork will use. If it is a DOM network, <metalocator> is a DTD file specification. If the output is a GME project, <metalocator> is the paradigm name. If <metalocator> is omitted, the name attribute of the diagram in the XML file specified by <diagram> is used as the locator.

DESCRIPTION

UdmCopy copies an existing Udm data network into a new one. Either of them may use any of the supported backend technologies (DOM or GME.

UdmCopy is a generic application, i.e. the meta information (class, attribute, relationship definitions) on the data is not provided runtime, but compile-time, in form of an UML XML file, specified by the <diagram> argument.

The entities specified by the parameters must be compatible with each other. <diagram> must be compatible to both <metalocator> and <indata> (e.g. created by a Udm application that used Udm source generated from <diagram>). <outdata> is generated to be compatible with <diagram>.

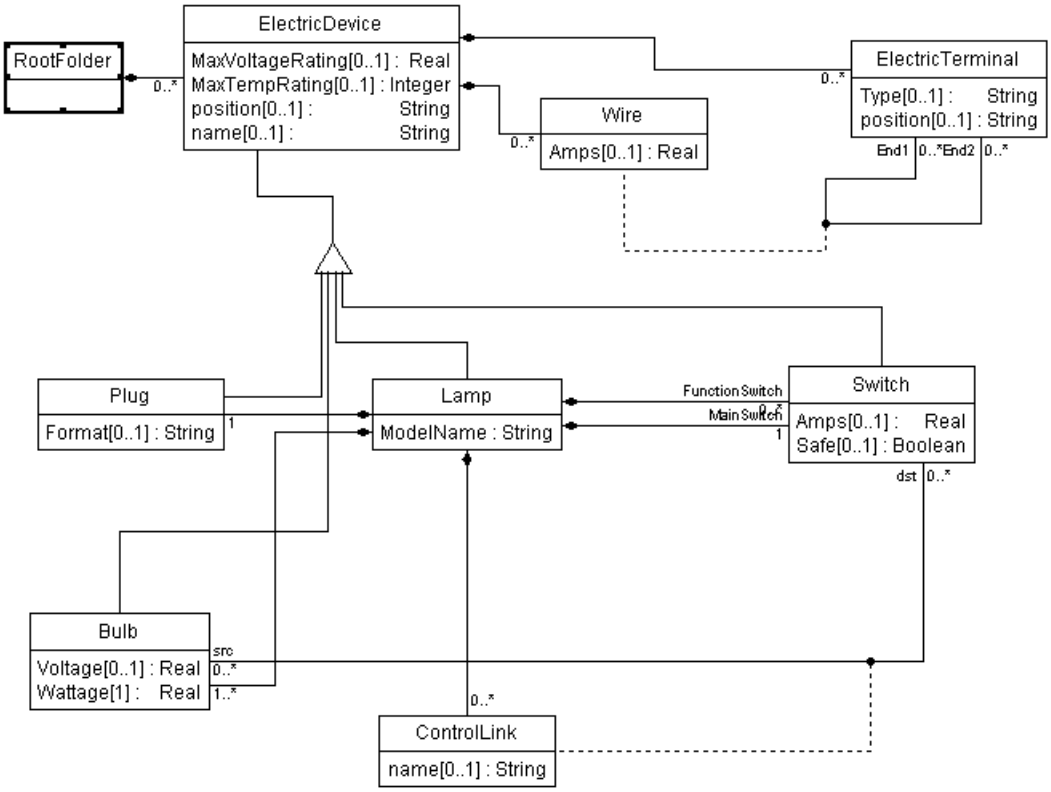
FILES

uml.dtd :	the <diagram> file must be compliant with this DTD. UdmCopy contains a copy of this file as an attached resource. The '-d' option can be used to specify a different DTD file.

Last update: 20 December 2001

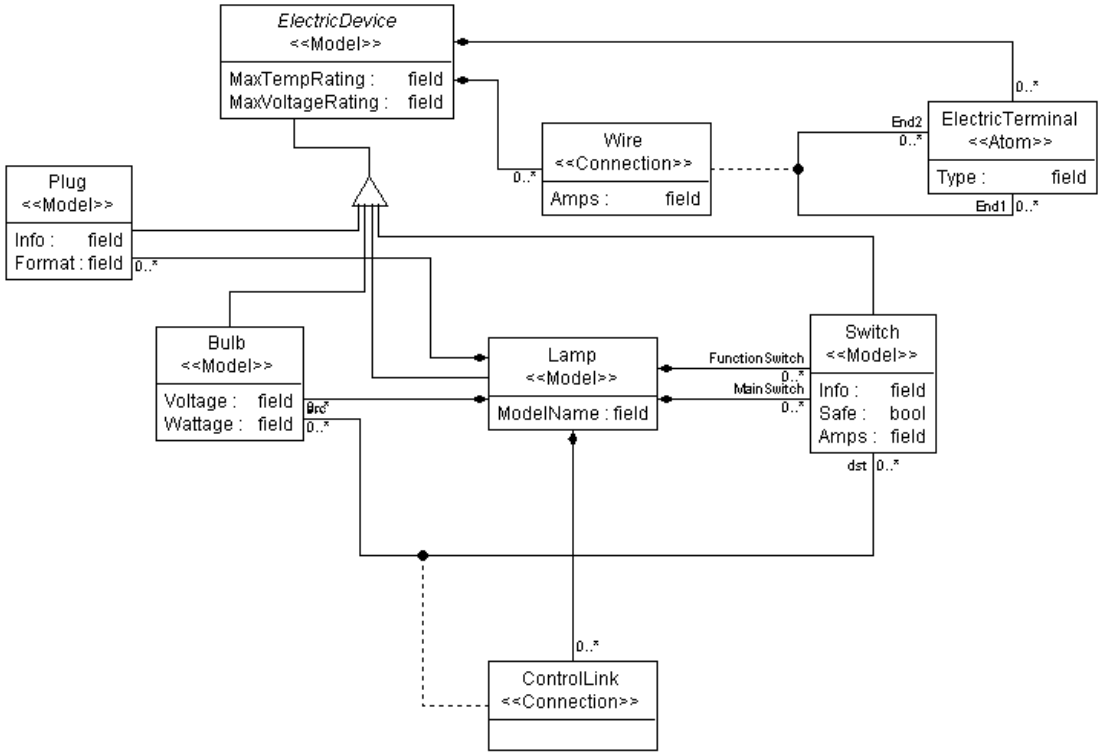
Appendix B

Simple UML class diagram, created in GME2000



Appendix C

Matching GMEMeta 2000 paradigm of the UML ClassDiagram



Appendix D

XML representation of UML information.

(Created with UML2XML GME2000 Interpretter from the Diagram in Appendix B)

```
<?xml version="1.0"?>
<!DOCTYPE Diagram SYSTEM "uml.dtd">
<Diagram name ="LampDiagram">
  <!-- Generated by the UML2XML interpreter -->

<Class _id="id000022" name= "Wire" isAbstract= "false" childRoles= " id000013"  association = "id00005">
  <Attribute name= "Amps" type= "Real" min= "0" max= "1"/>
</Class>
<Class _id="id000023" name= "Plug" isAbstract= "false" baseTypes= " id000024" childRoles= " id000019" >
  <Attribute name= "Format" type= "String" min= "0" max= "1"/>
</Class>
<Class _id="id000024" name= "ElectricDevice" isAbstract= "true" subTypes= " id000026 id000023 id000025 id000029" parentRoles= " id00008 id000012" childRoles= " id000011" >
  <Attribute name= "MaxVoltageRating" type= "Real" min= "0" max= "1"/>
  <Attribute name= "MaxTempRating" type= "Integer" min= "0" max= "1"/>
  <Attribute name= "position" type= "String" min= "0" max= "1"/>
  <Attribute name= "name" type= "String" min= "0" max= "1"/>
</Class>
<Class _id="id000025" name= "Switch" isAbstract= "false" baseTypes= " id000024" associationRoles= " id00001" childRoles= " id00007 id000015" >
  <Attribute name= "Amps" type= "Real" min= "0" max= "1"/>
  <Attribute name= "Safe" type= "Boolean" min= "0" max= "1"/>
</Class>
<Class _id="id000026" name= "Bulb" isAbstract= "false" baseTypes= " id000024" associationRoles= " id00000" childRoles= " id000021" >
  <Attribute name= "Voltage" type= "Real" min= "0" max= "1"/>
  <Attribute name= "Wattage" type= "Real" min= "1" max= "1"/>
</Class>
<Class _id="id000027" name= "ControlLink" isAbstract= "false" childRoles= " id000017"  association = "id00002">
  <Attribute name= "name" type= "String" min= "0" max= "1"/>
</Class>
<Class _id="id000028" name= "ElectricTerminal" isAbstract= "false" associationRoles= " id00003 id00004" childRoles= " id00009" >
  <Attribute name= "Type" type= "String" min= "0" max= "1"/>
  <Attribute name= "position" type= "String" min= "0" max= "1"/>
</Class>
<Class _id="id000029" name= "Lamp" isAbstract= "false" baseTypes= " id000024" parentRoles= " id00006 id000014 id000016 id000018 id000020" >
  <Attribute name= "ModelName" type= "String" min= "1" max= "1"/>
</Class>
<Class _id="id000030" name= "RootFolder" isAbstract= "false" parentRoles= " id000010" >
</Class>
<Association _id="id00002" name= "ControlLink" assocClass = "id000027">
  <AssociationRole _id= "id00000" name= "src" min= "0" max= "-1" target= "id000026"/>
  <AssociationRole _id= "id00001" name= "dst" min= "0" max= "-1" target= "id000025"/>
</Association>
<Association _id="id00005" name= "Wire" assocClass = "id000022">
  <AssociationRole _id= "id00003" name= "End1" min= "0" max= "-1" target= "id000028"/>
  <AssociationRole _id= "id00004" name= "End2" min= "0" max= "-1" target= "id000028"/>
```

```
</Association>
<Composition>
  <CompositionParentRole _id= "id00008" target= "id000024"/>
  <CompositionChildRole _id= "id00009" min= "0" max= "-1" target= "id000028"/>
</Composition>
<Composition>
  <CompositionParentRole _id= "id000012" target= "id000024"/>
  <CompositionChildRole _id= "id000013" min= "0" max= "-1" target= "id000022"/>
</Composition>
<Composition>
  <CompositionParentRole _id= "id00006" target= "id000029"/>
  <CompositionChildRole _id= "id00007" name= "MainSwitch" min= "1" max= "1" target= "id000025"/>
</Composition>
<Composition>
  <CompositionParentRole _id= "id000014" target= "id000029"/>
  <CompositionChildRole _id= "id000015" name= "FunctionSwitch" min= "0" max= "-1" target=
"id000025"/>
</Composition>
<Composition>
  <CompositionParentRole _id= "id000016" target= "id000029"/>
  <CompositionChildRole _id= "id000017" min= "0" max= "-1" target= "id000027"/>
</Composition>
<Composition>
  <CompositionParentRole _id= "id000018" target= "id000029"/>
  <CompositionChildRole _id= "id000019" min= "1" max= "1" target= "id000023"/>
</Composition>
<Composition>
  <CompositionParentRole _id= "id000020" target= "id000029"/>
  <CompositionChildRole _id= "id000021" min= "1" max= "-1" target= "id000026"/>
</Composition>
<Composition>
  <CompositionParentRole _id= "id000010" target= "id000030"/>
  <CompositionChildRole _id= "id000011" min= "0" max= "-1" target= "id000024"/>
</Composition>
</Diagram>
```

Appendix E

DTD file generated from the XML in Appendix D (generated with Udm.exe)

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated on Wed Feb 06 22:34:29 2002 -->

<!ELEMENT Plug (Wire|ElectricTerminal)*>
<!ATTLIST Plug
        MaxTempRating       CDATA    #IMPLIED
        MaxVoltageRating    CDATA    #IMPLIED
        position    CDATA    #IMPLIED
        name        CDATA    #IMPLIED
        Format      CDATA    #IMPLIED
>

<!ELEMENT Wire EMPTY>
<!ATTLIST Wire
        _id         ID          #IMPLIED
        Amps        CDATA    #IMPLIED
        End2_end_               IDREF     #IMPLIED
        End1_end_               IDREF     #IMPLIED
>

<!ELEMENT Switch (Wire|ElectricTerminal)*>
<!ATTLIST Switch
        _id         ID          #IMPLIED
        __child_as NMTOKENS          #IMPLIED
        MaxTempRating       CDATA    #IMPLIED
        MaxVoltageRating    CDATA    #IMPLIED
        position    CDATA    #IMPLIED
        name        CDATA    #IMPLIED
        Safe        (true|false) #IMPLIED
        Amps        CDATA    #IMPLIED
        src         IDREFS   #IMPLIED
>

<!ELEMENT Bulb (Wire|ElectricTerminal)*>
<!ATTLIST Bulb
        _id         ID          #IMPLIED
        MaxTempRating       CDATA    #IMPLIED
        MaxVoltageRating    CDATA    #IMPLIED
        position    CDATA    #IMPLIED
        name        CDATA    #IMPLIED
        Wattage     CDATA    #REQUIRED
        Voltage     CDATA    #IMPLIED
        dst         IDREFS   #IMPLIED
>

<!ELEMENT ControlLink EMPTY>
<!ATTLIST ControlLink
        _id         ID          #IMPLIED
        name        CDATA    #IMPLIED
        dst_end_    IDREF    #IMPLIED
        src_end_    IDREF    #IMPLIED
>

<!ELEMENT ElectricTerminal EMPTY>
<!ATTLIST ElectricTerminal
        _id         ID          #IMPLIED
        position    CDATA    #IMPLIED
        Type        CDATA    #IMPLIED
        End2        IDREFS   #IMPLIED
        End1        IDREFS   #IMPLIED
>

<!ELEMENT Lamp (Plug|Wire|Switch|Bulb|ControlLink|ElectricTerminal)*>
<!ATTLIST Lamp
        MaxTempRating       CDATA    #IMPLIED
```

```
        MaxVoltageRating    CDATA    #IMPLIED
        position    CDATA    #IMPLIED
        name        CDATA    #IMPLIED
        ModelName           CDATA    #REQUIRED
>

<!ELEMENT RootFolder (Plug|Switch|Bulb|Lamp)*>
```

APPENDIX F.
.H file generated from the XML in Appendix D.
(generated with Udm.exe)

```
#ifndef MOBIES_LAMPDIAGRAM_H
#define MOBIES_LAMPDIAGRAM_H
// header file LampDiagram.h generated from diagram LampDiagram
// generated on Wed Feb 06 22:34:29 2002

#ifndef MOBIES_UDMBASE_H
#include "UdmBase.h"
#endif

namespace LampDiagram {

	typedef Udm::Object Object;

	class Plug;
	class Wire;
	class ElectricDevice;
	class Switch;
	class Bulb;
	class ControlLink;
	class ElectricTerminal;
	class Lamp;
	class RootFolder;

	void Initialize();
	extern Udm::UdmDiagram diagram;

	class Wire :  public Object {
	 public:
		static Uml::Class meta;

		Wire() { }
		Wire(Udm::ObjectImpl *impl) : Object(impl) { }
		Wire(const Wire &master) : Object(master) { }
		static Wire Cast(const Object &a) { return __Cast(a, meta); }

		static Wire Create(const Object &parent, const Uml::CompositionChildRole &role = Udm::NULLCHILDROLE) { return
__Create(meta, parent, role); }

		static Uml::Attribute meta_Amps;
		Udm::RealAttr Amps() const { return Udm::RealAttr(impl, meta_Amps); }

		static Uml::CompositionParentRole meta_ElectricDevice_parent;
		Udm::ParentAttr<LampDiagram::ElectricDevice> ElectricDevice_parent() const { return
Udm::ParentAttr<LampDiagram::ElectricDevice>(impl, meta_ElectricDevice_parent); }

		Udm::ParentAttr<LampDiagram::ElectricDevice> parent() const { return
Udm::ParentAttr<LampDiagram::ElectricDevice>(impl, Udm::NULLPARENTROLE); }
		static Uml::AssociationRole meta_End2_end_;
		Udm::AssocEndAttr<LampDiagram::ElectricTerminal> End2_end() const { return
Udm::AssocEndAttr<LampDiagram::ElectricTerminal>(impl, meta_End2_end_); }

		static Uml::AssociationRole meta_End1_end_;
		Udm::AssocEndAttr<LampDiagram::ElectricTerminal> End1_end() const { return
Udm::AssocEndAttr<LampDiagram::ElectricTerminal>(impl, meta_End1_end_); }

	};

	class ElectricDevice :  public Object {
	 public:
		static Uml::Class meta;

		ElectricDevice() { }
		ElectricDevice(Udm::ObjectImpl *impl) : Object(impl) { }
		ElectricDevice(const ElectricDevice &master) : Object(master) { }
		static ElectricDevice Cast(const Object &a) { return __Cast(a, meta); }

		static ElectricDevice Create(const Object &parent, const Uml::CompositionChildRole &role = Udm::NULLCHILDROLE) {
return __Create(meta, parent, role); }

		static Uml::Attribute meta_MaxTempRating;
		Udm::IntegerAttr MaxTempRating() const { return Udm::IntegerAttr(impl, meta_MaxTempRating); }

		static Uml::Attribute meta_MaxVoltageRating;
		Udm::RealAttr MaxVoltageRating() const { return Udm::RealAttr(impl, meta_MaxVoltageRating); }

		static Uml::Attribute meta_position;
		Udm::StringAttr position() const { return Udm::StringAttr(impl, meta_position); }

		static Uml::Attribute meta_name;
		Udm::StringAttr name() const { return Udm::StringAttr(impl, meta_name); }

		static Uml::CompositionChildRole meta_Wire_children;
```

```
                Udm::ChildrenAttr<LampDiagram::Wire> Wire_children() const { return Udm::ChildrenAttr<LampDiagram::Wire>(impl,
meta_Wire_children); }

                static Uml::CompositionChildRole meta_ElectricTerminal_children;
                Udm::ChildrenAttr<LampDiagram::ElectricTerminal> ElectricTerminal_children() const { return
Udm::ChildrenAttr<LampDiagram::ElectricTerminal>(impl, meta_ElectricTerminal_children); }

                Udm::ChildrenAttr<LampDiagram::Wire> Wire_kind_children() const { return
Udm::ChildrenAttr<LampDiagram::Wire>(impl, Udm::NULLCHILDROLE); }

                Udm::ChildrenAttr<LampDiagram::ElectricTerminal> ElectricTerminal_kind_children() const { return
Udm::ChildrenAttr<LampDiagram::ElectricTerminal>(impl, Udm::NULLCHILDROLE); }

                static Uml::CompositionParentRole meta_RootFolder_parent;
                Udm::ParentAttr<LampDiagram::RootFolder> RootFolder_parent() const { return
Udm::ParentAttr<LampDiagram::RootFolder>(impl, meta_RootFolder_parent); }

                Udm::ParentAttr<LampDiagram::RootFolder> parent() const { return Udm::ParentAttr<LampDiagram::RootFolder>(impl,
Udm::NULLPARENTROLE); }
        };

        class Plug :  public ElectricDevice {
         public:
                static Uml::Class meta;

                Plug() { }
                Plug(Udm::ObjectImpl *impl) : ElectricDevice(impl) { }
                Plug(const Plug &master) : ElectricDevice(master) { }
                static Plug Cast(const Object &a) { return __Cast(a, meta); }

                static Plug Create(const Object &parent, const Uml::CompositionChildRole &role = Udm::NULLCHILDROLE) { return
__Create(meta, parent, role); }

                static Uml::Attribute meta_Format;
                Udm::StringAttr Format() const { return Udm::StringAttr(impl, meta_Format); }

                static Uml::CompositionParentRole meta_Lamp_parent;
                Udm::ParentAttr<LampDiagram::Lamp> Lamp_parent() const { return Udm::ParentAttr<LampDiagram::Lamp>(impl,
meta_Lamp_parent); }

                Udm::ParentAttr<LampDiagram::ElectricDevice> parent() const { return
Udm::ParentAttr<LampDiagram::ElectricDevice>(impl, Udm::NULLPARENTROLE); }
        };

        class Switch :  public ElectricDevice {
         public:
                static Uml::Class meta;

                Switch() { }
                Switch(Udm::ObjectImpl *impl) : ElectricDevice(impl) { }
                Switch(const Switch &master) : ElectricDevice(master) { }
                static Switch Cast(const Object &a) { return __Cast(a, meta); }

                static Switch Create(const Object &parent, const Uml::CompositionChildRole &role = Udm::NULLCHILDROLE) { return
__Create(meta, parent, role); }

                static Uml::Attribute meta_Safe;
                Udm::BooleanAttr Safe() const { return Udm::BooleanAttr(impl, meta_Safe); }

                static Uml::Attribute meta_Amps;
                Udm::RealAttr Amps() const { return Udm::RealAttr(impl, meta_Amps); }

                static Uml::AssociationRole meta_src, meta_src_rev;
                Udm::AClassAssocAttr<LampDiagram::ControlLink, LampDiagram::Bulb> src() const { return
Udm::AClassAssocAttr<LampDiagram::ControlLink, LampDiagram::Bulb>(impl, meta_src, meta_src_rev); }

                static Uml::CompositionParentRole meta_FunctionSwitch_Lamp_parent;
                Udm::ParentAttr<LampDiagram::Lamp> FunctionSwitch_Lamp_parent() const { return
Udm::ParentAttr<LampDiagram::Lamp>(impl, meta_FunctionSwitch_Lamp_parent); }

                static Uml::CompositionParentRole meta_MainSwitch_Lamp_parent;
                Udm::ParentAttr<LampDiagram::Lamp> MainSwitch_Lamp_parent() const { return
Udm::ParentAttr<LampDiagram::Lamp>(impl, meta_MainSwitch_Lamp_parent); }

                Udm::ParentAttr<LampDiagram::ElectricDevice> parent() const { return
Udm::ParentAttr<LampDiagram::ElectricDevice>(impl, Udm::NULLPARENTROLE); }
        };

        class Bulb :  public ElectricDevice {
         public:
                static Uml::Class meta;

                Bulb() { }
                Bulb(Udm::ObjectImpl *impl) : ElectricDevice(impl) { }
                Bulb(const Bulb &master) : ElectricDevice(master) { }
                static Bulb Cast(const Object &a) { return __Cast(a, meta); }

                static Bulb Create(const Object &parent, const Uml::CompositionChildRole &role = Udm::NULLCHILDROLE) { return
__Create(meta, parent, role); }

                static Uml::Attribute meta_Wattage;
                Udm::RealAttr Wattage() const { return Udm::RealAttr(impl, meta_Wattage); }
```

```
                                        static Uml::Attribute meta_Voltage;
                                        Udm::RealAttr Voltage() const { return Udm::RealAttr(impl, meta_Voltage); }

                                        static Uml::AssociationRole meta_dst, meta_dst_rev;
                                        Udm::AClassAssocAttr<LampDiagram::ControlLink, LampDiagram::Switch> dst() const { return
Udm::AClassAssocAttr<LampDiagram::ControlLink, LampDiagram::Switch>(impl, meta_dst, meta_dst_rev); }

                                        static Uml::CompositionParentRole meta_Lamp_parent;
                                        Udm::ParentAttr<LampDiagram::Lamp> Lamp_parent() const { return Udm::ParentAttr<LampDiagram::Lamp>(impl,
meta_Lamp_parent); }

                                        Udm::ParentAttr<LampDiagram::ElectricDevice> parent() const { return
Udm::ParentAttr<LampDiagram::ElectricDevice>(impl, Udm::NULLPARENTROLE); }
                                };

                                class ControlLink :  public Object {
                                 public:
                                        static Uml::Class meta;

                                        ControlLink() { }
                                        ControlLink(Udm::ObjectImpl *impl) : Object(impl) { }
                                        ControlLink(const ControlLink &master) : Object(master) { }
                                        static ControlLink Cast(const Object &a) { return __Cast(a, meta); }

                                        static ControlLink Create(const Object &parent, const Uml::CompositionChildRole &role = Udm::NULLCHILDROLE) {
return __Create(meta, parent, role); }

                                        static Uml::Attribute meta_name;
                                        Udm::StringAttr name() const { return Udm::StringAttr(impl, meta_name); }

                                        static Uml::CompositionParentRole meta_Lamp_parent;
                                        Udm::ParentAttr<LampDiagram::Lamp> Lamp_parent() const { return Udm::ParentAttr<LampDiagram::Lamp>(impl,
meta_Lamp_parent); }

                                        Udm::ParentAttr<LampDiagram::ElectricDevice> parent() const { return
Udm::ParentAttr<LampDiagram::ElectricDevice>(impl, Udm::NULLPARENTROLE); }
                                        static Uml::AssociationRole meta_dst_end_;
                                        Udm::AssocEndAttr<LampDiagram::Switch> dst_end() const { return Udm::AssocEndAttr<LampDiagram::Switch>(impl,
meta_dst_end_); }

                                        static Uml::AssociationRole meta_src_end_;
                                        Udm::AssocEndAttr<LampDiagram::Bulb> src_end() const { return Udm::AssocEndAttr<LampDiagram::Bulb>(impl,
meta_src_end_); }

                                };

                                class ElectricTerminal :  public Object {
                                 public:
                                        static Uml::Class meta;

                                        ElectricTerminal() { }
                                        ElectricTerminal(Udm::ObjectImpl *impl) : Object(impl) { }
                                        ElectricTerminal(const ElectricTerminal &master) : Object(master) { }
                                        static ElectricTerminal Cast(const Object &a) { return __Cast(a, meta); }

                                        static ElectricTerminal Create(const Object &parent, const Uml::CompositionChildRole &role = Udm::NULLCHILDROLE)
{ return __Create(meta, parent, role); }

                                        static Uml::Attribute meta_position;
                                        Udm::StringAttr position() const { return Udm::StringAttr(impl, meta_position); }

                                        static Uml::Attribute meta_Type;
                                        Udm::StringAttr Type() const { return Udm::StringAttr(impl, meta_Type); }

                                        static Uml::AssociationRole meta_End1, meta_End1_rev;
                                        Udm::AClassAssocAttr<LampDiagram::Wire, LampDiagram::ElectricTerminal> End1() const { return
Udm::AClassAssocAttr<LampDiagram::Wire, LampDiagram::ElectricTerminal>(impl, meta_End1, meta_End1_rev); }

                                        static Uml::AssociationRole meta_End2, meta_End2_rev;
                                        Udm::AClassAssocAttr<LampDiagram::Wire, LampDiagram::ElectricTerminal> End2() const { return
Udm::AClassAssocAttr<LampDiagram::Wire, LampDiagram::ElectricTerminal>(impl, meta_End2, meta_End2_rev); }

                                        static Uml::CompositionParentRole meta_ElectricDevice_parent;
                                        Udm::ParentAttr<LampDiagram::ElectricDevice> ElectricDevice_parent() const { return
Udm::ParentAttr<LampDiagram::ElectricDevice>(impl, meta_ElectricDevice_parent); }

                                        Udm::ParentAttr<LampDiagram::ElectricDevice> parent() const { return
Udm::ParentAttr<LampDiagram::ElectricDevice>(impl, Udm::NULLPARENTROLE); }
                                };

                                class Lamp :  public ElectricDevice {
                                 public:
                                        static Uml::Class meta;

                                        Lamp() { }
                                        Lamp(Udm::ObjectImpl *impl) : ElectricDevice(impl) { }
                                        Lamp(const Lamp &master) : ElectricDevice(master) { }
                                        static Lamp Cast(const Object &a) { return __Cast(a, meta); }

                                        static Lamp Create(const Object &parent, const Uml::CompositionChildRole &role = Udm::NULLCHILDROLE) { return
__Create(meta, parent, role); }

                                        static Uml::Attribute meta_ModelName;
```

```
                                        Udm::StringAttr ModelName() const { return Udm::StringAttr(impl, meta_ModelName); }

                                        static Uml::CompositionChildRole meta_FunctionSwitch;
                                        Udm::ChildrenAttr<LampDiagram::Switch> FunctionSwitch() const { return
Udm::ChildrenAttr<LampDiagram::Switch>(impl, meta_FunctionSwitch); }

                                        static Uml::CompositionChildRole meta_MainSwitch;
                                        Udm::ChildAttr<LampDiagram::Switch> MainSwitch() const { return Udm::ChildAttr<LampDiagram::Switch>(impl,
meta_MainSwitch); }

                                        static Uml::CompositionChildRole meta_ControlLink_children;
                                        Udm::ChildrenAttr<LampDiagram::ControlLink> ControlLink_children() const { return
Udm::ChildrenAttr<LampDiagram::ControlLink>(impl, meta_ControlLink_children); }

                                        static Uml::CompositionChildRole meta_Plug_child;
                                        Udm::ChildAttr<LampDiagram::Plug> Plug_child() const { return Udm::ChildAttr<LampDiagram::Plug>(impl,
meta_Plug_child); }

                                        static Uml::CompositionChildRole meta_Bulb_children;
                                        Udm::ChildrenAttr<LampDiagram::Bulb> Bulb_children() const { return Udm::ChildrenAttr<LampDiagram::Bulb>(impl,
meta_Bulb_children); }

                                        Udm::ChildrenAttr<LampDiagram::Plug> Plug_kind_children() const { return
Udm::ChildrenAttr<LampDiagram::Plug>(impl, Udm::NULLCHILDROLE); }

                                        Udm::ChildrenAttr<LampDiagram::ElectricDevice> ElectricDevice_kind_children() const { return
Udm::ChildrenAttr<LampDiagram::ElectricDevice>(impl, Udm::NULLCHILDROLE); }

                                        Udm::ChildrenAttr<LampDiagram::Switch> Switch_kind_children() const { return
Udm::ChildrenAttr<LampDiagram::Switch>(impl, Udm::NULLCHILDROLE); }

                                        Udm::ChildrenAttr<LampDiagram::Bulb> Bulb_kind_children() const { return
Udm::ChildrenAttr<LampDiagram::Bulb>(impl, Udm::NULLCHILDROLE); }

                                        Udm::ChildrenAttr<LampDiagram::ControlLink> ControlLink_kind_children() const { return
Udm::ChildrenAttr<LampDiagram::ControlLink>(impl, Udm::NULLCHILDROLE); }

                                        Udm::ParentAttr<Udm::Object> parent() const { return Udm::ParentAttr<Udm::Object>(impl,
Udm::NULLPARENTROLE); }
                                };

                                class RootFolder :  public Object {
                                 public:
                                        static Uml::Class meta;

                                        RootFolder() { }
                                        RootFolder(Udm::ObjectImpl *impl) : Object(impl) { }
                                        RootFolder(const RootFolder &master) : Object(master) { }
                                        static RootFolder Cast(const Object &a) { return __Cast(a, meta); }

                                        static RootFolder Create(const Object &parent, const Uml::CompositionChildRole &role = Udm::NULLCHILDROLE) {
return __Create(meta, parent, role); }

                                        static Uml::CompositionChildRole meta_ElectricDevice_children;
                                        Udm::ChildrenAttr<LampDiagram::ElectricDevice> ElectricDevice_children() const { return
Udm::ChildrenAttr<LampDiagram::ElectricDevice>(impl, meta_ElectricDevice_children); }

                                        Udm::ChildrenAttr<LampDiagram::Plug> Plug_kind_children() const { return
Udm::ChildrenAttr<LampDiagram::Plug>(impl, Udm::NULLCHILDROLE); }

                                        Udm::ChildrenAttr<LampDiagram::ElectricDevice> ElectricDevice_kind_children() const { return
Udm::ChildrenAttr<LampDiagram::ElectricDevice>(impl, Udm::NULLCHILDROLE); }

                                        Udm::ChildrenAttr<LampDiagram::Switch> Switch_kind_children() const { return
Udm::ChildrenAttr<LampDiagram::Switch>(impl, Udm::NULLCHILDROLE); }

                                        Udm::ChildrenAttr<LampDiagram::Bulb> Bulb_kind_children() const { return
Udm::ChildrenAttr<LampDiagram::Bulb>(impl, Udm::NULLCHILDROLE); }

                                        Udm::ChildrenAttr<LampDiagram::Lamp> Lamp_kind_children() const { return
Udm::ChildrenAttr<LampDiagram::Lamp>(impl, Udm::NULLCHILDROLE); }

                                        Udm::ParentAttr<Udm::Object> parent() const { return Udm::ParentAttr<Udm::Object>(impl,
Udm::NULLPARENTROLE); }
                                };

        }

        #endif //MOBIES_LAMPDIAGRAM_H
```

APPENDIX G.

Sample UDM-based program

```
#include "LampDiagram.h"
#include <UdmGme.h>                                    // these must be included to make SmartDataNetwork work
#include <UdmDom.h>

using namespace LampDiagram;

int main(int argc, char* argv[])
{
        if(argc < 2) {
                cout << "Usage: CreateLampModel <Udmdataname>\n";
                return -1;
        }
        try {
                Udm::SmartDataNetwork nw(diagram);

                nw.CreateNew(argv[1],"LampDiagram", RootFolder.meta);
                {
                RootFolder rrr = RootFolder::Cast(nw.GetRootObject());
                Lamp doubleBulbLamp = Lamp::Create(rrr);
                doubleBulbLamp.name() = "HighLight XL 150";
                doubleBulbLamp.MaxVoltageRating() = 220.0;
                doubleBulbLamp.MaxTempRating()    = 200;
                ElectricTerminal termA = ElectricTerminal::Create(doubleBulbLamp);
                ElectricTerminal termB = ElectricTerminal::Create(doubleBulbLamp);
                termA.position() = "(500,20)";
                termB.position() = "(500,85)";

                Plug thePlug = Plug::Create(doubleBulbLamp);
                thePlug.position() = "(500,150)";
                thePlug.name() = "Plug";
                ElectricTerminal plt1 = ElectricTerminal::Create(thePlug);
                plt1.position() = "(20,100)";
                ElectricTerminal plt2 = ElectricTerminal::Create(thePlug);
                plt2.position() = "(20,200)";

// create mainswitch and its terminals
                Switch mainSwitch = Switch::Create(doubleBulbLamp, Lamp::meta_MainSwitch);
                mainSwitch.position() = "(300,100)";
                mainSwitch.name() = "MainSW";
                ElectricTerminal mst1 = ElectricTerminal::Create(mainSwitch);
                mst1.position() = "(20,100)";
                ElectricTerminal mst2 = ElectricTerminal::Create(mainSwitch);
                mst2.position() = "(220,100)";

//create switches and its terminals
                Switch switch1 = Switch::Create(doubleBulbLamp, Lamp::meta_FunctionSwitch);
                switch1.position() = "(300,250)";
                switch1.name() = "SW1";
                ElectricTerminal s1t1 = ElectricTerminal::Create(switch1);
                s1t1.position() = "(20,100)";
                ElectricTerminal s1t2 = ElectricTerminal::Create(switch1);
                s1t2.position() = "(220,100)";

                Switch switch2 = Switch::Create(doubleBulbLamp, Lamp::meta_FunctionSwitch);
                switch2.position() = "(300,250)";
                switch2.name() = "SW2";
                ElectricTerminal s2t1 = ElectricTerminal::Create(switch2);
                s2t1.position() = "(20,100)";
                ElectricTerminal s2t2 = ElectricTerminal::Create(switch2);
                s2t2.position() = "(220,100)";




// Create bulbs with terminals
                Bulb bulb1    = Bulb::Create(doubleBulbLamp);
                bulb1.name() = "Bulb1";
                bulb1.position() = "(100,150)";
                ElectricTerminal b1t1 = ElectricTerminal::Create(bulb1);
                b1t1.position() = "(220,100)";
                ElectricTerminal b1t2 = ElectricTerminal::Create(bulb1);
                b1t2.position() = "(220,200)";

                Bulb bulb2    = Bulb::Create(doubleBulbLamp);
                bulb2.name() = "Bulb2";
                bulb2.position() = "(100,150)";
                ElectricTerminal b2t1 = ElectricTerminal::Create(bulb2);
                b1t1.position() = "(220,100)";
                ElectricTerminal b2t2 = ElectricTerminal::Create(bulb2);
                b1t2.position() = "(220,200)";



// wire up the lamp
```

```
//create wires in the lamp

                        Wire w1 = Wire::Create(doubleBulbLamp);
                        Wire w2 = Wire::Create(doubleBulbLamp);
                        Wire w3 = Wire::Create(doubleBulbLamp);
                        Wire w4 = Wire::Create(doubleBulbLamp);
                        Wire w5 = Wire::Create(doubleBulbLamp);
                        Wire w6 = Wire::Create(doubleBulbLamp);
                        Wire w7 = Wire::Create(doubleBulbLamp);

//connect them

                        //live onnections
                        //#1: connect plug 1 to main switch input
                        w1.End1_end() = plt1;
                        w1.End2_end() = mst1;

                        //#2, #3: connect main switch output to sw1, sw2 inputs
                        w2.End1_end() = mst2;
                        w2.End2_end() = s1t1;

                        w3.End1_end() = mst2;
                        w3.End2_end() = s2t1;

                        //#4: connect sw1 output to bulb 1
                        w4.End1_end() = s1t2;
                        w4.End2_end() = b1t1;

                        //#5: connect sw2 output to bulb 2
                        w5.End1_end() = s2t2;
                        w5.End2_end() = b2t1;

                        //ground connections:
                        //#6, #7 connect plug 2 to bulbs
                        w6.End1_end() = plt2;
                        w6.End2_end() = b1t2;

                        w7.End1_end() = plt2;
                        w7.End2_end() = b2t2;



// assign values to a few attrs

                        w1.Amps() = 3.5;
                        w2.Amps() = 4;
                        w3.Amps() = 5;
                        w4.Amps() = 11;
                        w5.Amps() = 12;
                        w6.Amps() = 13;
                        w7.Amps() = 15.0;



                        bulb1.Wattage() = 55;
                        bulb2.Wattage() = 155;
                        doubleBulbLamp.ModelName() = "SuperDoubleLight 155";

                        s1t1.Type() = string("Copper");
                        s1t2.Type() = string("Aluminium");


                        //creating control links

                        ControlLink cl1 = ControlLink::Create(doubleBulbLamp);
                        cl1.src_end() = bulb1;
                        cl1.dst_end() = switch1;
                        cl1.name() = "Bulb 1 switcher";

                        ControlLink cl2 = ControlLink::Create(doubleBulbLamp);
                        cl2.src_end() = bulb2;
                        cl2.dst_end() = switch2;
                        cl2.name() = "Bulb 2 switcher";



                        set<ControlLink> lks = doubleBulbLamp.ControlLink_kind_children();

                        for(set<ControlLink>::iterator l1 = lks.begin(); l1 != lks.end(); l1++) {
                                cout << "Control: " << string(l1->name()) << " ";
                                cout << "Bulb: " << string(Bulb(l1->src_end()).name()) << " ";
                                cout << "Switch: " << string(Switch(l1->dst_end()).name()) << endl << endl;
                        }

// test how ObjectByID works
                        Udm::Object oo = nw.ObjectById(bulb1.uniqueId());
                        Bulb uu = Bulb::Cast(oo);

                        cout << long(uu.Wattage()) << endl;

                        }
```

```
                    nw.CloseWithUpdate();

        }
        catch(const udm_exception &e)
        {
                    cout << "Exception: " << e.what() << endl;
                    return(-1);

        }

        try {                                                   // reload the data to see if it is valid
                    Udm::SmartDataNetwork nw(diagram);
                    nw.OpenExisting(argv[1],"", Udm::CHANGES_LOST_DEFAULT);
                    {
                      RootFolder doubleBulbLamp = RootFolder::Cast(nw.GetRootObject());
                    }
                    nw.CloseNoUpdate();
                    cout << "Basically, all is OK" << endl ;
        }
        catch(const udm_exception &e)               {
                    cout << "Exception: " << e.what()  << endl;
                    return(-1);

        }
        return 0;
}
```

References:

1.*GME 2000 User's Manual* (Version 2.0, December 2001 or later version) ISIS, Vanderbilt University

2.*The MGA Library* (September 2000 or later version) Arpad Bakay / ISIS, Vanderbilt University

3.*Advanced C++ Programming Styles and Idioms*. COPLIEN, J. O. Addison-Wesley, Reading, Mass., 1992.