The UDM framework

Arpad Bakay, Endre Magyari

List of contributors:

- **Tihamer Levendovszky**
- **Tamas Paka**

Institute for Software-Integrated Systems
Vanderbilt University

September 2006

Table of Contents

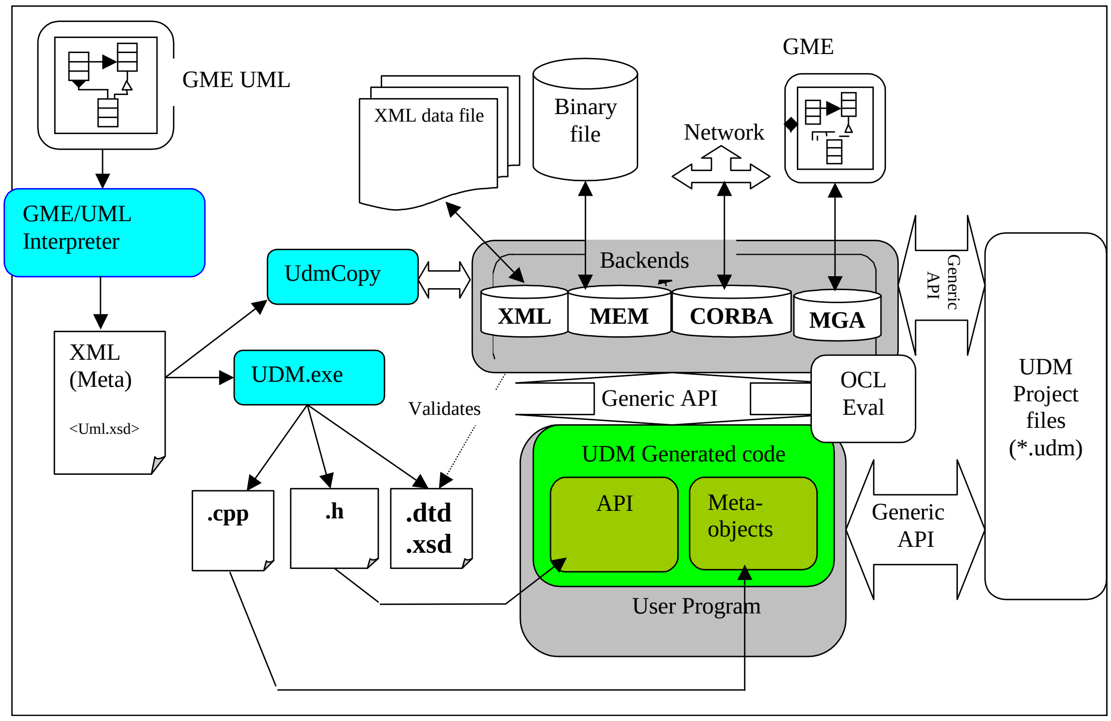
Contents

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 XSD/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 **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) or XML schema definition (XSD).

- 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 a native persistence format(also called as MEM files or Memory(Static) Backend), and supports two other persistence formats (XML or GME) – these can be used for persistence. Otherwise, Udm can also be used to create converters to and from native formats.
- From version 2.00, Udm supports a persistence format(Udm Project file, *.udm) for a set of datanetworks with the possibility of having cross-datanetwork links between them. The GME/UML modeling environment also supports multi-package UML diagrams, with the possibility of having cross-package associations. The instance of a cross-package association is a cross-datanetwork link. 7

The typical process of using the UDM is as follows:

- First, a UML metamodel is created in GME/UML.
- The information in the UML diagram is converted to an XML file using 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 XSD file Uml.xsd). In the case when the UML diagram contains multiple packages, the diagram is converted to a UDM Project file.
- 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.

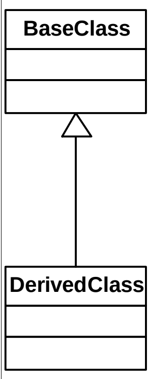
For some typical operations, the Udm package also includes generic programs that work on Udm data. Since these are generic, the data structure information is supplied at run-time as files in the XML UML format.

- 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 {

...
  class BaseClass :
        public Udm::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 Uml2XML 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:

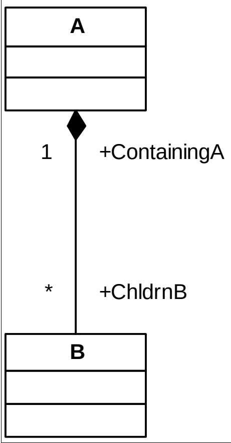
- An un initialized instance variable is an empty reference (reference to Udm::Null).
- Several variables may refer to the same instance object, and simple assignment of variables does not imply the creation of a new instance.
- New objects are always created by the *ClassName*::Create() static member 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 UML2XML tool currently does not preserve the abstractness of classes, thus all classes are regarded as non-abstract.

2.2 Composition

	`class B :  public Udm::Object {`
	`  Udm::ParentAttr<A> ContainingA();`
	`  Udm::ParentAttr<A> parent()`
	`}`
	
	`class A :  public Udm::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 Udm::Object {
        ...
        static A Create(const Udm::Object &parent,
                         const Udm::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:

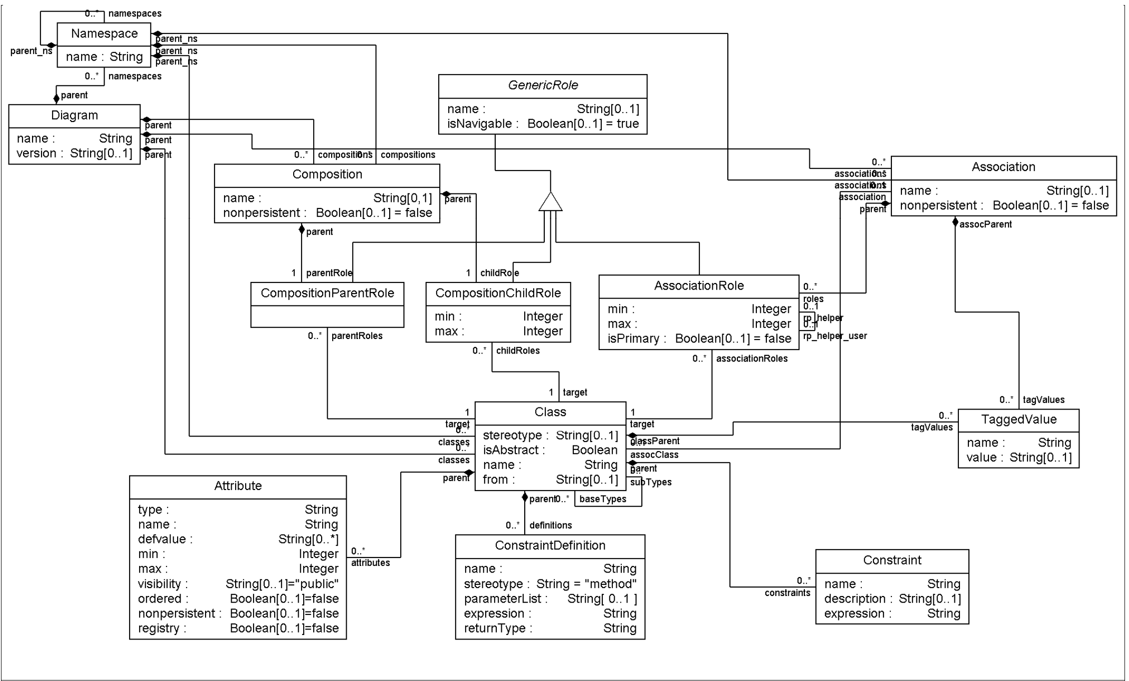
1. By setting the parent() (of an object to NULL)
2. By assigning a new child set to the parent, omitting from the set the objects to be deleted.

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 obsoletes 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*, *Text*, *Real*, *Boolean* or *Integer*. *Text* is the same *String* but it gets persisted as XML Text Node in DOM backend.

-*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..*

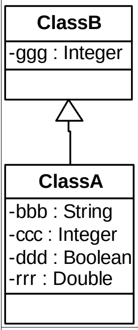
-The *ordering* property is meaningful if the multiplicity upper bound is greater than 1. (array attributes) . Currently, UDM supports this feature in all backend types, and in case of non-persistent attributes as well.

-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 ClassB :  public Udm::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;
}
```

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

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 ‘\’

2.5.2 User code with array attributes

In user code, array attributes can be set/get in the same way 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.”;
```

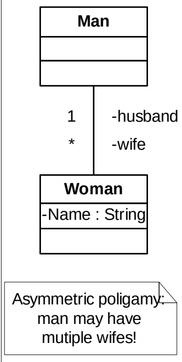
Note: When the array is smaller than the index, then the same note applies as in the case of assignment operator before.
The += operator does not work – and also does not make sense – for boolean arrays.

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 Diagram

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

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

UDM Header

```
// 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();
}
```

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

Cross-package associations are the associations with endpoints and/or associated class in different Uml packages. Such an association can be modeled in the GME/UML environment by creating a ClassCopy object pointing to a class in a different package and setting up an association to it.

2.7 Constraints

Starting from release 1.70, UDM supports Constraints, which can be evaluated on the data-network.

For the time being, only the UML paradigm of GME is extended to include constraints and constraint definitions into a class diagram, in other cases the set of these rules has to be created manually. Because in run-time a meta-model of a data-network may be modified, the set of the constraints can be extended as well.

In order to use the OCL evaluation all constraints has to be compiled (syntax and semantic check). Nevertheless, if the set of the constraints is changing run-time, the constraints can be used only after they have been compiled.

For compilation the user has to call initially the Ocl::Intialize function supplying the appropriate Uml::Diagram of the data network. By default during the compilation all errors are collected. In case of errors a udm_exception is thrown, and they are printed into the standard output.

After the compilation the valid constraints can be evaluated with the Ocl::Evaluator class.

The constructor of the class has only one mandatory argument: the object as the context of the evaluation. Calling Check() all constraints can be evaluated.

After constraint compilation and evaluation, call Ocl::UnInitialize() and pass the Uml::Diagram of the data network. This call will release the resources allocated by Ocl::Initialize() during constraint compilation.

Both the compilation and the evaluation processes can be modified by options.

Option structures:

Ocl::SErrorNotification:

- bStdOutEnabled : If this flag is true, then errors are printed into the standard output. (default is TRUE)
- eExceptionKind : With this member the user can set the type of exception throwing.
    - ENT_NONE : No exception is thrown

    - ENT_FIRST : Immediately after the first error, an exception is thrown
    - ENT_ALL : An exception is thrown with the error summary. (default)

Ocl::SEvaluationOptions:

- bLogicalShortCircuit : If this flag is true, then logical operators (&&, ||, =>) will be considered as short-circuit operators. (default is TRUE)
- bIteratorShortCircuit : If this flag is true, then iterators will return immediately after the result is available. (default is TRUE)
- bTrackingEnabled : If this flag is true, then the constraint evaluation can be followed easily when an exception is thrown during the evaluation or the constraint is violated.(default is TRUE)
- sErrorNotification : With this member we can set the error notification (defaults are the same as SerrorNotification)
- iContainmentDepth : With this option we can evaluate all constraints for the sub-tree whose root is the object passed to the constructor of Ocl::Evaluator.
    - D = 0 : Only the root object will be examined.
    - D > 0 : The evaluation will reach the objects which are in level D from the root (e.g. if D = 1, then the root object and its immediate children will be examined.
    - D < 0 : The whole tree will be examined. (default)
- bSkipInvalid: If this flag is true, unparsed and failed constraints are skipped during evaluation. (default is FALSE)

Constraint Intialization:

bool Ocl::Initialize(const Uml::Diagram& objDiagram, const Ocl::SErrorNotification& sErrorNotification = Ocl::SErrorNotification());

The function collects all constraints and Constraint Definition residing in the meta-model and tries to compile them. Depending on the notification options in case of error, either false is returned or udm_exception is thrown.

Constraint Evaluation:

Ocl::Evaluator(const Udm::Object& objObject, const set<Uml::Constraint>& setConstraints = set<Uml::Constraint>());

This constructor can be used to supply the arguments of an evaluation. The context always has to be specified. If the set of the constraints is empty, then all the constraints which can be applied to the current object will be evaluated. If the set is not empty, then only this set will be evaluated, and only for the objects which fall into the examination range and on which the constraint(s) can be applied.

EEvaluationResult Ocl::Evaluator::Check(const SEvaluationOptions& sOptions = SEvaluationOptions()) const;

The evaluation can be performed with this method supplying the options. Depending on the options, in case of a run-time exception or constraint violation an udm_exception may be thrown.

EEvaluationResult has the values:

- CER_FALSE : There were no constraints which returned in undefined. Only constraint violation occurred.
- CER_TRUE : All constraints were satisfied.

CER_UNDEFINED : There was at least one constraint which returned in undefined for an object.

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 its value 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()
Query attribute "desynched" status:	
	Udm::Object::getAttrStatus()

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

3. Using the UDM API

3.1 Generating UDM source files

1. Run the MetaGME2Uml interpreter on your MetaGME model, or use GME with the UML paradigm to create a UML diagram for the data model of the target application.
2. Use the UML2XML GME interpreter on the UML diagram to generate UDM XML files.
3. Run Udm.exe to generate a .cpp and a .h file from the XML format. (Udm.exe also generates an XML XSD file or DTD.)

Udm.exe, MetaGME2Uml, and the UML2XML interpreter are all included in the *Udm binary release*. It also contains the include header files 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 samples/C++/CreateLampModel/CreateLampModel.vcproj in the UDM installation), but the following procedures also include instructions for doing it from scratch:

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

-Microsoft Visual Studio 2008 or 2010 (not included in the distribution)
-The UDM DLL and import library (UdmDll_3_2.dll, UdmDll.lib). The import library is included automatically. Do not specify it under "Additional Dependencies" in your project.
-The UDM headers
-The Xerces XML Parser DLL

Your project needs to include the domain-specific 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 your project includes the generic UDM headers by adding the header directory $(UDM_PATH)/include to *Project/Properties/C++/General/Additional Include Directories*.
3. Include the UDM DLL import library by defining preprocessor macro UDM_DYNAMIC_LINKING in *Project/Properties/ C++/Preprocessor/Preprocessor*. You will also need to add $(UDM_PATH)\lib

to the input library path in Project/Properties/Linker/General/Additional Library Directories
4. Add the generated UDM API files (one or more C++ and H files) 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.
5. The files that use the API need to #include the generated UDM API header file.
6. 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 generated API .cpp file should be set <u>individually</u> to '*Not using precompiled headers'* in the *Project/Properties/C++/Precompiled Headers/* pane.
7. Set additional options according to your preferences, but do not change the default for the run-time library in the *Project/Properties/C++/Code Generation/Runtime Library* option from 'Multithreaded DLL'(Release configuration) or 'Debug Multithreaded DLL'(Debug configuration). The libraries use C++ exceptions, so do not disable exception handling. Additionally, do not specify ITERATOR_DEBUG_LEVEL or _SECURE_SCL.

Installing UDM adds both the UDM DLL and the Xerces DLL (xerces-c_2.dll) to the DLL search path. These DLLs are required to run a UDM program.

Also, the generated XSD 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.XSD", including double quotes), and the custom, "XSD" resource type. To add resources, an .rc file (and possibly a resource.h file) needs to be inserted into the project. Alternatively, invoke Udm.exe with the -g option to integrate the XSD into the code.

3.3 Generic program structure

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

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. an XSD for XML backends, or a paradigm name 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_LOST_DEFAULT:** committed sequences are lost unless CloseWithUpdate, CloseAs, or SaveAs are used. This option is preferred.
- **Udm::CHANGES_PERSIST_ALWAYS:** committed sequences are immediately recorded in the original backend (no CloseNoUpdate, CloseAs, SaveAs will be used). This option is deprecated, since a failure during saving is indicated by an

exception, and the runtime may be running the destructor due to a different exception. The C++ Standard guarantees terminate() will be called under these circumstances.

- **Udm::CHANGES_PERSIST_DEFAULT:** committed sequences are recorded in current backed unless CloseNoUpdate, CloseAs, or SaveAs is used. This option is deprecated.

(Not all constants are accepted by all backends.)

It's recommended to create or open data networks with Udm::CHANGES_LOST_DEFAULT and use CloseWithUpdate, CloseAs or SaveAs to save your changes because Udm::CHANGES_PERSIST_ALWAYS and Udm::CHANGES_PERSIST_DEFAULT are being deprecated.

```
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.

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 backend storage for this data tree was initialized.

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

Get the metainfo of the data network.

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**

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(s) as a reference to follow the instructions below.

Objects that are instances of UML Classes are all direct or indirect descendants **Udm::Object**.

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 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.

3.6 UDM Project functionality

3.6.1 The basic idea

The goal to achieve was to allow links with endpoints in different UDM Datanetworks. Thus, a structure which is capable to handle a set of DataNetwork had to be introduced: *UdmProject*.

To provide this functionality, Udm is using Udm for maintaining a special datanetwork to hold the metainformation for cross-package associations and cross-package links – the instances of cross-package associations. The UDM projects which contain meta-information and cross-package associations are called meta UDM projects and those which contain instance datanetworks and cross-datanetwork links are called instance UDM projects.

A Meta UDM project is the product of the GME UML interpreter whenever more than one package is modeled.

When a UML model which contains more than one UML package is interpreted, automatically a meta UDM project is generated. During this process, for each UML package XML metainformation is generated, as in the normal scenario, without cross-package associations. Beyond these, a special XML metainformation is also generated, which represents a single class diagram, with “proxy classes” to all the classes which are involved in cross-package associations. These classes are named “<class_name>_cross_ph_<package_name>”, <class_name> being the name of the class which is involved in a cross-package association, and <package_name> being the name of the package which contains the class. Each of these classes are generated with two attributes, *rem_sysname* and *rem_id*. At the instance level, in instance UDM projects, the value of these attributes will identify the real UDM objects, where the proxy UDM object points to. Cross-datanetwork links basically are links between these UDM proxy objects.

Normally, instance UDM projects are created from scratch, which means that the framework creates the empty datanetworks and handles to these datanetworks can be obtained from the project. Cross-datanetwork links created in such cases are persistent and can be saved.

However, there was a need to create cross-links between the objects of existing UDM datanetworks, without having to recreate these UDM datanetworks as part of a project. This was made possible with the trade-off that cross-datanetwork links created in such projects can not be saved and are available during runtime and the cross-datanetwork links are hold in a non-persitent static UDM datanetwork. This type of UDM Project is called StaticUdmProject.

<u>3.6.2 DataNetworkSpecifiers structs</u>

When creating an instance UDM project, all the datanetworks which are going to be parts of the Project have to be specified. In the case of persistent instance UDM projects, the struct *Udm::DataNetworkSpecifier* is used, and it contains the systemname, the metalocator and the root class of the datanetwork, which will be created by the framework – as part of the instance UDM project. In the case of static (non-persistent) UDM projects, existing datanetworks can be imported into a StaticUdmProject with the struct *Udm:: StaticDataNetworkSpecifier*. To specifiy a staticdatanetwork, only it's systemname, and a pointer to the datanetwork is required.

<u>3.6.3 Dynamic meta specifier struct</u>

Normaly, when creating or opening an instance UDM project, the framework needs to locate the metainformation (Uml::Diagram) for each loaded UDM datanetwork inside the project. In most cases, these are compiled in and UDM has a mechanism (Metadepository) to locate by name the metainformation. However, some UDM applications prefer to load dynamically the metainformation instead of having it compiled-in, thus the lookup mechanism through UDM MetaDepository will fail. In these cases, additional meta datanetworks can be submitted to UdmProject.

<u>3.7 The UDM Metadepository</u>

Since version 2.00, all compiled-in metainformation (UML Diagrams) are registered in the framework and indexed by the name of the Diagram. Registered UML diagrams can be retrieved using the static member functions of the Udm::MetaDepository class.

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 XSD for the XML files compatible with the input interface definition. The basic XML-processing facilities provided are loading (parsing) an XML file or string into the backend, and saving the backend data structure into an XML file or string.

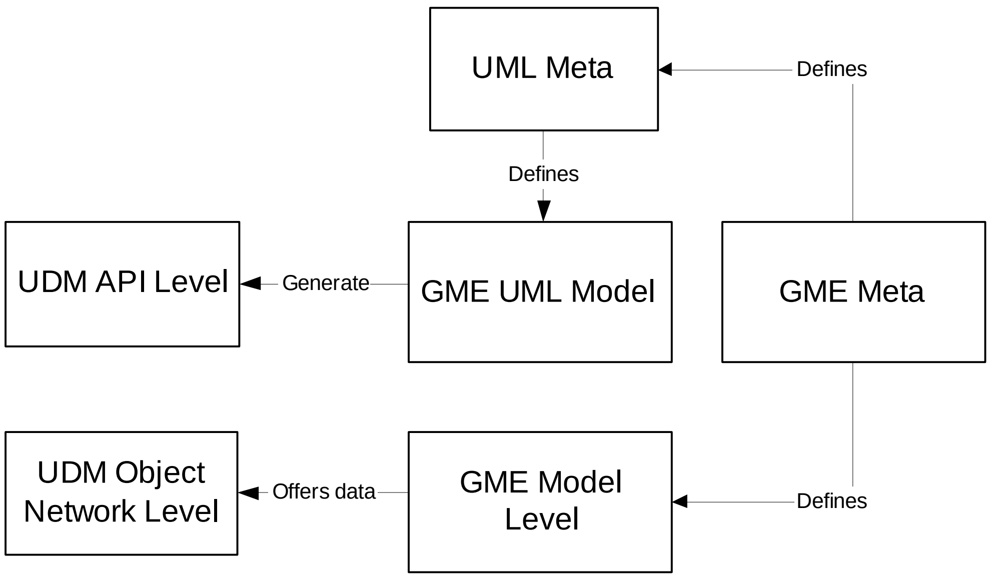
The XML backend imposes the following restrictions:

- -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 (although 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).

   GME model annotations can be accessed if the corresponding UML classes have an attribute called *annotations* defined in the UML diagram. The type of the attribute must be array of Strings and also have the *registry* property.

   Each element of an array is a mapping of one GME annotations registry node:
      registry node path without leading "annotations" + "=" + registry node value

   Literals "=" and "\" in registry node paths and values are escaped by "\".

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.)

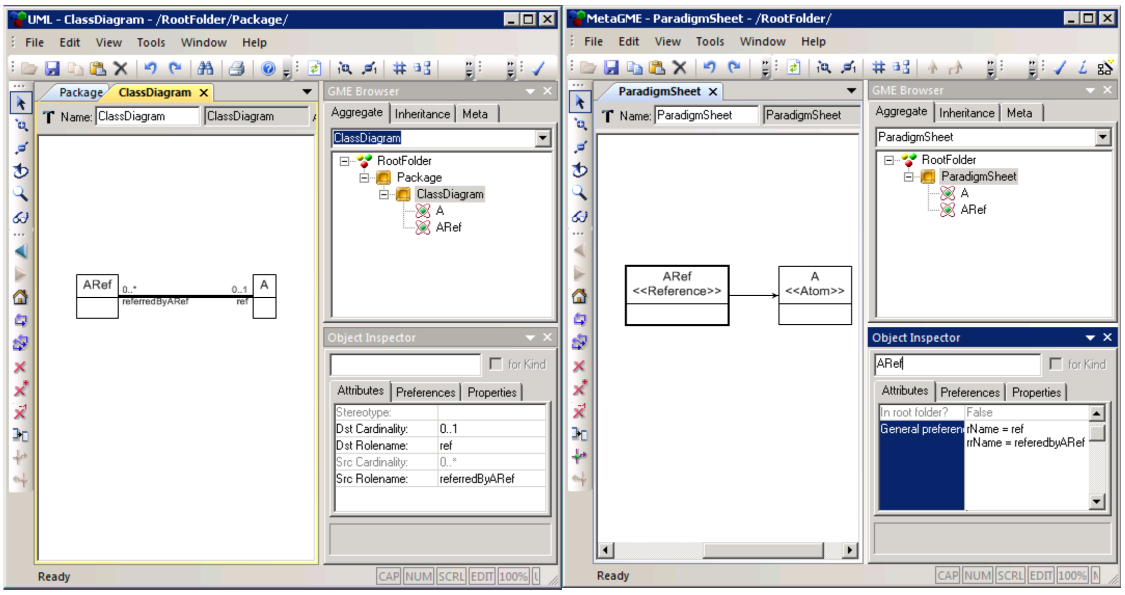
4. **Associations**
   Associations in the input UML must be defined in a way so that they can be mapped onto one of the association types supported by GME: references, sets, or connections. To resolve ambiguities, stereotypes at either end of an association or the naming of the association and its role 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):



Name: ClassDiagram

Stereotype:	
Dst Cardinality:	0..1
Dst Rolename:	ref
Src Cardinality:	0..*
Src Rolename:	referredByARef

Name: ParadigmSheet

ARef

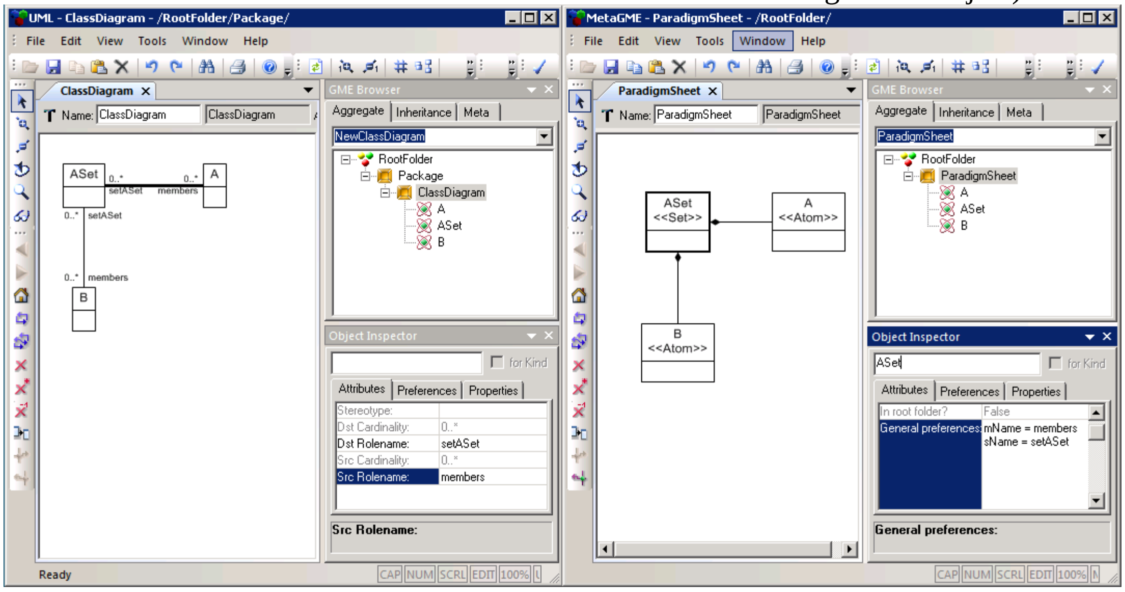
In root folder?	False

General preferen	rName = ref
	rrName = referedbyARef

Figure 6: UML model for references and the equivalent GME model

-

-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):

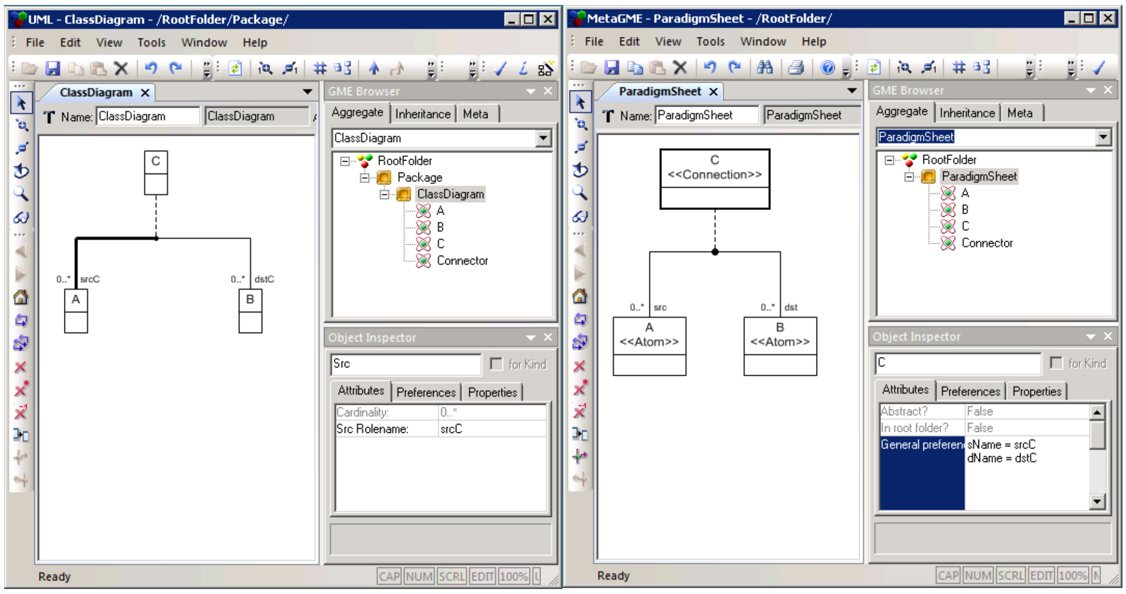


Stereotype:	
Dst Cardinality:	0..*
Dst Rolename:	setASet
Src Cardinality:	0..*
Src Rolename:	members

In root folder?	False

General preferences	mName = members
	sName = setASet

Figure 7: UML model for sets and the equivalent GME model



Cardinality:	0..*
Src Rolename:	srcC

Abstract?	False
In root folder?	False

General preferences	sName = srcC
	dName = dstC

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:Figure 8: UML model for association with association class and the equivalent GME model

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:

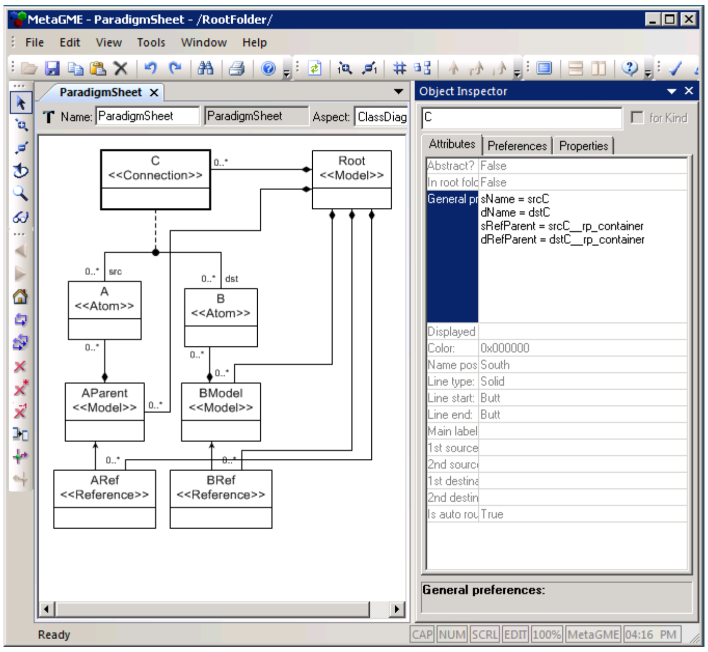
-If an association (without association class) in the UML diagram is to be implemented by a GME reference, the name of the association name must either be empty or equal to the name of the reference class (i.e. the class found at one of its ends). The GME meta-reference must have the GME registry value '/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. See figure 6 above for an example.

-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. See figure 7 above for an example.

-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). See figure 8 above for an example.

-If the previous type of GME connection needs to be used with model reference ports, then the GME meta-connection needs to have one or two more registry values, '/sRefParent' and '/dRefParent'.

These registry values will be used by the MetaGME2Uml tool to generate two extra UML associations that can be used to access/set the chain of references between the association and its source/destination. The extra UML associations end names will be set to those registry values. The values must have the following format: the value used for the '/sname' or '/dname' registry values suffixed with '__rp_container' or '__rp_helper'. See the following two figures for an example GME model and the corresponding UML model:



C

Abstract?	False
In root folc	False

General pr	sName = srcC
	dName = dstC
	sRefParent = srcC__rp_container
	dRefParent = dstC__rp_container

Displayed	
Color:	0x000000
Name pos	South
Line type:	Solid
Line start:	Butt
Line end:	Butt
Main label	
1st source	
2nd sourc	
1st destina	
2nd destin	
Is auto rou	True

General preferences:

Figure 9: GME model with model reference ports

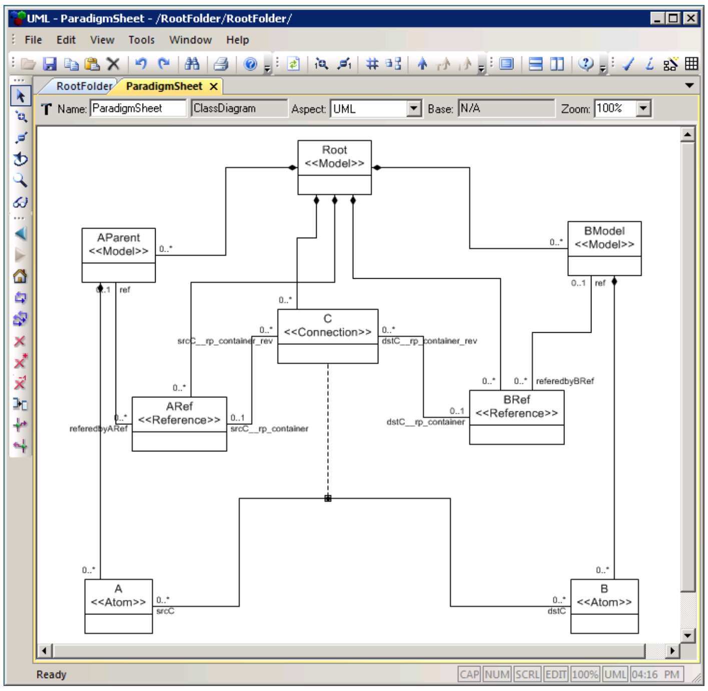


Figure 10: UML model corresponding to the GME model from figure 9

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.Connections between model reference ports

To connect two models that don't have the same parent model, the connection needs be established between one or two references to the involved models, reference(s) belonging to the same parent model as the connection.

While some UDM applications need to access only the connected models and are not interested in the chain of references used to connect the models, other applications need to read and/or manipulate the chain(s) of references used to connect the models. For this type of scenario the Udm framework provides two extra UML associations that make the chain(s) of references available to the application.

To have the two extra UML associations created and to be able thus to access and manipulate the chain of references, the GME meta-connection needs to have one or two more registry values, '/sRefParent' and '/dRefParent'. The values specify the end names of the extra UML associations. The values must have the following format: the value used for the '/sname' or '/dname' registry values suffixed with '__rp_container' or '__rp_helper'.

7.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 MetaGME2Uml GME interpreter included in the distribution, generate the UML class diagram from the GME metamodel.
4.Set the real role names for each association.

5.Generate the API with *udm.exe*.
6.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.

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 can 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 have 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.XSD. 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.

6. UDM Internal Architecture

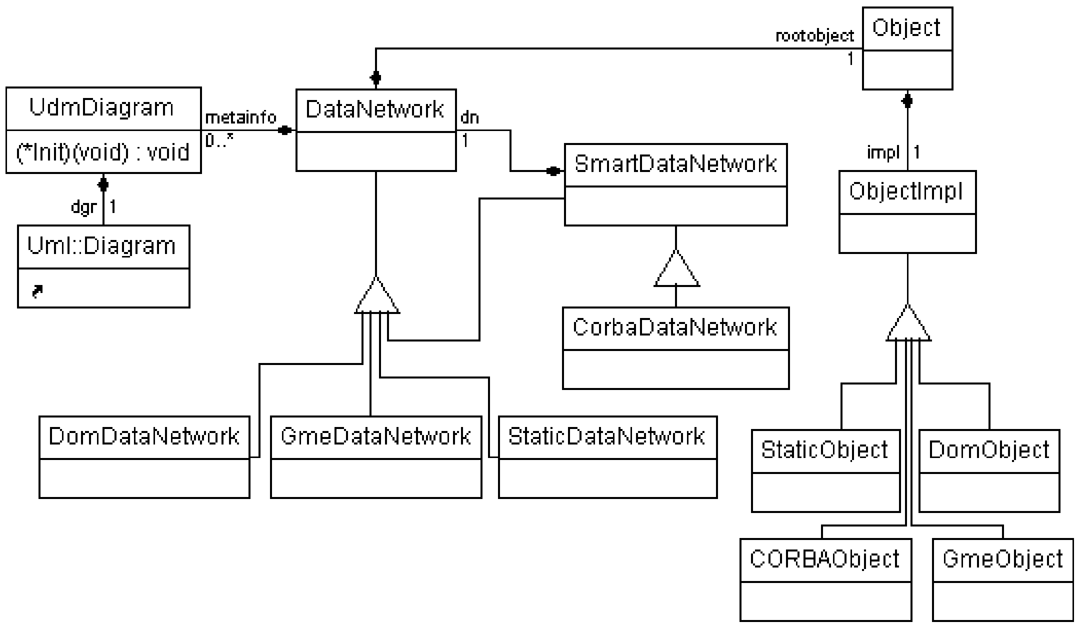


Figure - Udm Internal Arhitecture

6.1 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/.xsd 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 CreateNewToString(  const string &metalocator,
                                  const Uml::Class &rootclass,
                                  enum BackendSemantics sem = CHANGES_PERSIST_ALWAYS);
```

Creates a new datanetwork into a memory string.

metalocator - The name of the corresponding .dtd/.xsd file, without the extension
rootclass - type of the root object
sem - default behavior of the backend. See earlier in this document.

Implemented only in the DOM back-end.

- 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/.xsd file.
sem - the default behavior of the backend.

Implemented in all the back-ends.

- virtual void OpenExistingFromString(string &str,
  const string &metalocator,
  enum BackendSemantics sem = CHANGES_PERSIST_ALWAYS);

Opens an existing data network from a string.

str - the string from which to read the data
metalocator - DOM will need it if the XML file header does not contain correct or relevant information about the location of the corresponding .dtd/.xsd file.
sem - the default behavior of the backend.

Implemented only in the DOM back-end.

- 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.

•set<Object> GetAllInstancesOf(const ::Uml::Class& meta);

Retrieves all the instances of a class. Implemented in all the back-ends.

•const string & Str();

Retrieves the string representation of a data network using a memory string. Implemented only in the DOM back-end.

```
•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.

•struct AssociationRoleInfo GetAssociationRoleInfo(const ::Uml::AssociationRole& role)

Returns information that can be used to deduce the direction of a connection in backends that support this feature (MGA). The returned struct has too boolean members:

has_direction - the backend supports the notion of source and destination of an association is_src - the target of this AssociationRole is the source of the association

6.2 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 thecontainment 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.

•vector<string> getStringAttrArr(const Uml::Attribute &meta) const
•void setStringAttr(const Uml::Attribute &meta, const vector<string> &a)
•vector<bool> getBooleanAttr(const Uml::Attribute &meta) const
•void setBooleanAttr(const Uml::Attribute &meta, const vector<bool> &a)
•void<long> getIntegerAttr(const Uml::Attribute &meta) const
•void setIntegerAttr(const Uml::Attribute &meta, const vector<long> &a)
•vector<double> getRealAttr(const Uml::Attribute &meta) const
•void setRealAttr(const Uml::Attribute &meta, const vector<double> &a)

Attribute set'ers and get'ers for array type values. Meta mandatory parameter specifies which attribute to set or to get.

•long getAttrStatus(const ::Uml::Attribute &meta)

Get attribute “desynched” status. The returned value is one of the following values:

– Udm::ATTSTATUS_HERE: the value is defined in the current instance, i.e. it is “desynched” from archetype
– Udm::ATTSTATUS_METADEFAULT: the value is a default value defined by the meta
– value > 0: the attribute value is inherited from archetype.

•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;

•void connectTo(const ::Uml::AssociationRole &meta, const Object &target, const vector<Object> &refs)

Connects an association with association class to one of its ends using the vector of refs as a chain of references to use to reach the end. The vector may be empty, in which case Udm will try to find a chain by itself, or it may be incomplete, in which case Udm will check if it can reach the target by continuing the given chain.

•void disconnectFrom(const ::Uml::AssociationRole &meta, const Object &peer)

Disconnect an end of an association with association class by setting to NULL the first reference from the chain of references connecting the association to its end. Can be also called reversed, on the end object.

•vector<Object> getConnectingChain(const ::Uml::AssociationRole &meta, const Object &peer) const

Returns the chain of references connecting an association with association class to the given end.

•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 instances 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 each 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 isInstance() const;

Returns true if the object is an instance of an object. Returns false if is not.

•bool isSubtype() const;

Returns true if the object is derived from an object. Returns false if is not.

- 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.

- Object AttachLibrary(const Object &lib_src, const string &lib_name);

Imports a library into the current data-network. A child is created for the current object and the root object of the library, lib_src, is copied into it, together with all its children. The library name is set to lib_name. Returns the child into which lib_src has been copied.

- bool isLibObject() const;

Returns true if the object is part of a library, false otherwise.

- bool isLibRoot() const;

Returns true if the object is the root of a library, false otherwise.

- bool getLibraryName(string &name) const;

For objects that are library roots returns the library name into argument name and returns true. If the object is not a library root, the function returns false.

- void setLibraryName(const char *name);

If name is not null, sets the name of the attached library to the given value. If name is null, the the library is disconnected from this data-network.

- void GetChildRole(const Object &child, Uml::CompositionChildRole &ret) const

Get the composition child role the given child has in this parent. Sets the ret parameter to NULL if the given child is not actually a child.

6.3 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.

•vector<string> getStringAttrArr(const Uml::Attribute &meta) const
•void setStringAttr(const Uml::Attribute &meta, const vector<string> &a)
•vector<bool> getBooleanAttr(const Uml::Attribute &meta) const
•void setBooleanAttr(const Uml::Attribute &meta, const vector<bool> &a)
•void<long> getIntegerAttr(const Uml::Attribute &meta) const
•void setIntegerAttr(const Uml::Attribute &meta, const vector<long> &a)
•vector<double> getRealAttr(const Uml::Attribute &meta) const
•void setRealAttr(const Uml::Attribute &meta, const vector<double> &a)

Attribute setters and getters for array type values. Meta always specifies the attribute to operate.

•long getAttrStatus(const ::Uml::Attribute &meta)

Get attribute “desynched” status. The returned value is one of the following values:

– Udm::ATTSTATUS_HERE: the value is defined in the current instance, i.e. it is “desynched” from archetype
– Udm::ATTSTATUS_METADEFAULT: the value is a default value defined by the meta
– value > 0: the attribute value is inherited from archetype.

•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 void getChildRole(ObjectImpl *c, Uml::CompositionChildRole &ret) const

  If c is a child of this object, then set ret to be the role taken by the child, otherwise set ret to null.

- 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 instances 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.

•void connectTo(const ::Uml::AssociationRole &meta, ObjectImpl *target, const vector<ObjectImpl *> &refs)

Connects an association with association class to one of its ends using the vector of refs as a chain of references to use to reach the end. The vector may be empty, in which case Udm will try to find a chain by itself, or it may be incomplete, in which case Udm will check if it can reach the target by continuing the given chain.

•void disconnectFrom(const ::Uml::AssociationRole &meta, ObjectImpl *peer)

Disconnect an end of an association with association class by setting to NULL the first reference from the chain of references connecting the association to its end. Can be also called reversed, on the end object.

•vector<ObjectImpl *> getConnectingChain(const ::Uml::AssociationRole &meta, const ObjectImpl *peer) const

Returns the chain of references connecting an association with association class to the given end.

•ObjectImpl *AttachLibrary(ObjectImpl *lib_src, const string &lib_name);

Imports a library into the current data-network. A child is created for the current object and the root object of the library, lib_src, is copied into it, together with all its children. The library name is set to lib_name. Returns the child into which lib_src has been copied.

•bool isLibObject() const;

Returns true if the object is part of a library, false otherwise.

•bool isLibRoot() const;

Returns true if the object is the root of a library, false otherwise.

•bool getLibraryName(string &name) const;

For objects that are library roots returns the library name into argument name and returns true. If the object is not a library root, the function returns false.

•void setLibraryName(const char *name);

If name is not null, sets the name of the attached library to the given value. If name is null, the the library is disconnected from this data-network.

6.4 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);
•Class classByName(const Namespace &ns, const string &name);
•Class assocClassByName(const Diagram &d, const string &name);

- Class assocClassByName(const Namespace &ns, const string &name);
- Association associationByName(const Diagram &d, const string &name);
- Association associationByName(const Namespace &ns, const string &name);
- Composition compositionByName(const Diagram &d, const string &name);
- Composition compositionByName(const Namespace &ns, const string &name);
- Diagram diagramByName(const Diagram &d, const string &name);
- Namespace namespaceByName(const Diagram &d, const string &name);
- Namespace namespaceByName(const Namespace &ns, const string &name);

Finds a class/association class/association/composition/diagram/namespace by its name in a diagram/namespace.

- Class classByPath(const Diagram &d, const string &path, const string &delim);
- Namespace namespaceByPath(const Diagram &d, const string &path, const string &delim);

Finds a class/namespace by its namespace-based hierarchical path. The path is the place of the searched class/namespace in the namespace hierarchy. The nodes of the tree are separated in the path by the specified delimiter string.

- 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.

- Composition matchChildToParent(Class c, Class p, const char * crole, const char * prole = NULL);

  Same as the previous function, but additionally constraint the possible compositions by rolenames.

- AssociationRole matchPeerToPeer(Class c, Class target_class, Class target_aclass, const char * role = NULL);

  Finds the only suitable AssociationRole to reach 'target_class' (in case of associations without association class) or 'target_aclass' (in case of associations with assoc.class) from c. Results may be constrained to AssociationRole-s with name 'role' on c's side.
  if target_aclass is provided than target_class is ignored, and only those association roles are considered which are in an association class based association.
  role can be NULL. if not null, the results will be filtered against the rolename as well
  multiple or no result will return null object. The returned value is always the other AssociationRole, pointing to c's peer.

- 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.

- bool GetChildRoleChain(const Class &origin, const Class &what, vector<ChildRoleChain> &chains);

  Fills the vector chains with the possible containments paths of class what in class origin, at any level; returns true if successful, false if a loop was detected.

6.4.1 UML (Meta) iterable collections

- DiagramAssociations; constructor DiagramAssociations(const Diagram &dgr)
- DiagramClasses; constructor DiagramClasses(const Diagram &dgr)
- DiagramCompositions; constructor DiagramCompositions(const Diagram &dgr)

  Collections that can be used to iterate over all associations/classes/compositions from a diagram.

- NamespaceAssociations; constructor NamespaceAssociations(const Namespace &ns)
- NamespaceClasses; constructor NamespaceClasses(const Namespace &ns)

•NamespaceCompositions; constructor NamespaceCompositions(const Namespace &ns)

Collections that can be used to iterate over all associations/classes/compositions from a namespace.

•DiagramNamespaces; constructor DiagramNamespaces(const Diagram &dgr)
•NamespaceNamespaces; constructor NamespaceNamespaces(const Namespace &ns)

Collections that can be used to iterate over all namespaces from a diagram/namespace.

6.5 Miscellaneous calls

•int 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 UdmUtil::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 if found, it will extract its value.

•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.

6.6 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.

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

- 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.

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

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

- Retrieves the adjacent objects, together with the association class, 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. Associations without an association class are ignored.
  Parameter clsType can be null.

```
multiset< pair<Object, Object> > GetAdjacentObjectsWithAssocClasses(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, __int64& value);
bool GetIntValue(string strAttrName, string& value);
bool GetStrValue( string strAttrName, string& value);
bool GetStrValue( string strAttrName, __int64& value);
bool GetRealValue( string strAttrName, double& value);
bool GetBoolValue( string strAttrName, bool& value);
bool SetIntValue( string strAttrName, const __int64& value);
bool SetIntValue( string strAttrName, const string& value);
bool SetStrValue(string strAttrName, const string& value);
bool SetStrValue(string strAttrName, const __int64& value);
bool SetRealValue( string strAttrName, double value);
bool SetBoolValue( string strAttrName, bool value);

bool GetIntValues(string strAttrName, vector<__int64>& value);
bool GetStrValues( string strAttrName, vector<string>& value);
bool GetRealValues( string strAttrName, vector<double>& value);
bool GetBoolValues( string strAttrName, vector<bool>& value);

bool SetIntValues(string strAttrName, const vector<__int64>& value);
bool SetStrValues( string strAttrName, const vector<string>& value);
bool SetRealValues( string strAttrName, const vector<double>& value);
bool SetBoolValues( string strAttrName, const vector<bool>& value);
```

- Returns the parent object.

```
Object GetParent();
Object container();
```

- 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);
```

- Tests if the object is in the tree rooted at where.

```
bool IsNodeOfTree(const Object &where);
```

- Recursively obtains objects of a certain type down in the hierarchy rooted at this. The function uses a CompositionChildRole chain which was previously obtained by Uml::GetChildRoleChain() function.

```
set<Object> getChildrenByChildRoleChain(const ::Uml::Class& meta, vector<
::Uml::CompositionChildRole> chain);
```

- 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();
```

- Gets the source and, respectively, the destination, of this association class instance. Only the MGA backend supports directed connections and can return the source and destination. For the other backends null is returned.

```
Object getSrcObject();
Object getDstObject();
```

- Creates an object of clsType

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

- Creates an object of clsType using the given composition role names.

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

- 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 a link of a simple association or an association with association class. If AscType.clsAssociation is not valid, a simple association will be attempted. On error or when there is no such link returns false, otherwise true.

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

- Returns one specific end of an association class object.

```
Object GetAssociationEnd(string roleName);
```

- Checks if the object is non-empty reference object and returns the referred object if true.

```
Object getReferencedObject();
```

- Removes an object from the persistent storage.

```
void DeleteObject();
```

- Finds the child (only first-level children) in where which is an inherited subtype of this. Throws exception if not found.

```
Object FindCorrespondingObjectInSubtypes(const Object & where);
```

- Finds the child (only first-level children) in where which is an inherited instance of this. Throws exception if not found.

```
Object FindCorrespondingObjectInInstances(const Object & where);
```

- Finds the child (recursive, whole tree) in where which is an inherited subtype of this. Throws exception if not found.

```
Object FindCorrespondingObjectInSubtypesTree(const Object & where);
```

- Finds the child (recursive, whole tree) in where which is an inherited instance of this. Throws exception if not found.

```
Object FindCorrespondingObjectInInstancesTree(const Object & where);
```

- Gets the set of my parent's subtypes/instances.

```
set<Object> GetMyParentsSubtypes();
set<Object> GetMyParentsInstances();
```

- Checks if the object has derived or instantiated objects.

```
bool HasObjectSubtypes();
bool HasObjectInstances();
```

- Finds the closest parent which has an archetype.

```
Object FindClosestPrimarilyDerivedParent();
```

- Finds a derived/instance object of this in where.

```
Object getDerivedObjectInTree(const Udm::Object &where);
Object getInstantiatedObjectInTree(const Udm::Object &where);
```

- Checks if where is within the same derived/instantiated block as this object.

```
Object getDerivedObjectInPrimarilyDerivedBlock(const Udm::Object &where);
Object getInstantiatedObjectInPrimarilyDerivedBlock(const Udm::Object &where);
```

•Finds the closest parent which has an archetype.

```
Object FindClosestPrimarilyDerivedParent();
```

•Returns the path in the node tree from the root object to this object.
The name of the nodes is the value of the string attribute att_name (default value is “name”).
The nodes are separated in the result with delimiter strDelimiter (default value is “./.”).
If bReverseOrder is false (the default), then the path is from the root object to this object, otherwise the path is reversed, from this object to the root object.
If bNeedRootFolder is false (the default), then the root object is not added to the path.
If omit_lead_delim is false (the default), then the delimiter is not added at the beginning of the path. Otherwise, the path starts with the delimiter.

```
string getPath( const std::string& strDelimiter, bool bReverseOrder, bool bNeedRootFolder, const string &att_name, bool omit_lead_delim);
```

•Returns the path in the node tree from the root object to this object.
The nodes are separated in the result with delimiter strDelimiter (default value is “./.”).
If bNeedRootFolder is true (the default), then the root object is added to the path.

```
string getPath2( const std::string& strDelimiter , bool bNeedRootFolder);
```

•Returns the depth level of this object in the node tree. The root object is at level zero.

```
int depth_level();
```

6.7 UDM Project classes and members

These calls are defined as member functions of the class Udm::UdmProject and Udm::StaticUdmProject.

•Constructs an empty UdmProject. No datanetworks are created yet.

```
UdmProject(bool static_pr = false);
```

•Opens an existing UDM Project file

```
void OpenExisting(const string & project_file, enum BackendSemantics = Udm::CHANGES_PERSIST_ALWAYS);
```

•Creates a UDM Project file. The Datanetworks which are to be created as part of the project are specified through a DataNetworkSpecifiers:

```
class DataNetworkSpecifier
{
public:
        const string filename();
        const string metalocator();
        const Uml::Class &rootclass();
        DataNetworkSpecifier(const string& fn, const string& ml, const Uml::Class& rc)
;
}
```

```
void CreateNew(const string & project_file, vector<DataNetworkSpecifier>, const Udm::UdmDiagram& cross_diag , enum BackendSemantics = Udm::CHANGES_PERSIST_ALWAYS);
```

•Creates a UDM Meta Project file.

```
void CreateNewMeta(const string & project_name, const string & project_file, vector<DataNetworkSpecifier>, enum BackendSemantics = Udm::CHANGES_PERSIST_ALWAYS);
```

•Get pointers to the datanetworks inside a project

```
vector<DataNetwork*> GetDataNetworks();
```

•Get a reference to a specific datanetwork (located by it’s systemname)

```
Udm::DataNetwork&  GetDataNetwork(const string &which);
```

•Get all the meta diagrams for the datanetworks in the Project.

```
vector<Uml::Diagram> GetMetaDiagrams() const;
```

•Check if the project is a meta UDM project. If so, this returns true.

```
bool IsCrossMetaSpecifier() const;
```

•Check if the project is an instance UDM project. If so, this returns true.

```
bool HasCrossMeta() const;
```

•Valid only in case of meta UDM projects, returns the DataNetwork containing the metainformation (UML classdiagram) describing the cross-package associations.

```
DataNetwork& GetCrossMetaNetwork();
```

•Valid only in case of instance UDM projects, returns the UdmDiagram holding the Initialize() function and the metainformation for the datanetwork containing the cross-datanetwork links.

```
UdmDiagram& GetCrossMeta() const;
```

•Add dynamically metainformation which was not compiled-in the application.

```
virtual void AddDynamicMeta(const DynamicMetaSpecifier&);
```

•Remove the dynamically added datanetworks.

```
virtual void ResetDynamicMetas();
```

6.8 UDM MetaDepository members

•Locate a Uml::Diagram by it’s name

```
static const  UdmDiagram & LocateDiagram(const string& DgrName);
```

7. Namespace support

Since version UDM 3.0.0, namespace support has been added to UDM.

7.1 Modelling environment

The GME UML environment has been modified, a new element called "namespace" was introduced. A namespace can be contained in a package, and a namespace can contain classes, associations, compositions. The modelling of a namespace is not mandatory, classes can be contained directly in package, as it was the case in previous UDM versions.

7.2 Mapping to C++ code, code generator

Modelled namespaces are reflected as C++ namespaces in generated code that are nested in the C++ namespace which corresponds to the UML package in the model:

UML modelling element	C++ mapping	Remarks
UML Package	C++ namespace (outter)	mandatory; the name of the C++ namespace is the 'alias' attribute of the package. If that does not exist then it's name
UML Namespace	C++ namespace (inner, if any)	It may not exist
UML Class	C++ Class	Can be generated either directly in the namespace that corresponds to the UML package or in a namespace that corresponds to a UML Namespace

The code generator can create a .h/.cpp pair of files for each UML class, for each UML namespace or for each UML package. The desired mode can be selected by passing the appropiate option to Udm.exe, see -w switch.

7.3 Mapping to xsd/xml files in DOM backend

For each namespace the code generator will generate an XSD file, with type definitions in the targetnamespace.
For the classes placed directly in the UML package a separate XSD file will be generated for type definitions in the default namespace.

In the case of datanetworks, a single XML file is maintained which references all the XSD files. The XSD files may also contain cross-references when for instance a cross-namespace containment is defined.

For UML namespace an URI namespace is generated automatically, with the http://www.isis.vanderbilt.edu/2004/schemas/ prefix; however this can be customized with the –u switch of udm.exe

In order to be able to parse almost any XML input as a UDM datanetwork, UML/URI namespaces can be ignored during parsing (see –i switch), or some attributes may be parsed in with namespace qualifiers (see –q switch)

8. The UdmPseudoInterface API:

The UdmPseudoInterface API is a C++ wrapper library around core Udm::Object and Udm::DataNetwork APIs. The Java API uses this wrapper library to access the core UDM API, thus this section is dedicated mostly to the developers (and not to the users) of the Java 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.

<u>class UdmPseudoObject</u>

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.

9. C++ template metaprogramming with UDM API

When generating code, Udm can generate additional types and methods that provide support for using template metaprogramming techniques in the UDM applications.

Template metaprogramming is a metaprogramming technique in which C++ templates are used to implement generic programming. Generic programming allows a programmer to write abstract algorithms and data structures that can be later reused on a diverse variety of specific types.

One example of a framework using the Udm support for template metaprogramming is LEESA (Language for Embedded quEry and traverSAl), a domain-specific embedded language in C++ that allow the programmer to write in a novel way traversals over the typed object structures of the domain modeled by the UDM.

The extended API generated by Udm defines additional C++ types that convey the relationships between UML/C++ classes to C++ generic algorithms. With the regular API only the inheritance relationship is available to generic algorithms. The extended API allow generic algorithms to also use the composition and association relationships.

As a building block for its support for template metaprogramming, Udm uses the Boost MPL library. The type lists mentioned below are Boost MPL vectors.

The support for template metaprogramming consists of the following:

- each generated C++ class representing an UML class has a nested type, MetaKind, which is a tag for its stereotype; the tags are defined in UdmMPL.h and they are the following types:
  - Udm::AtomMetaTag
  - Udm::ConnectionMetaTag
  - Udm::FCOMetaTag
  - Udm::FolderMetaTag
  - Udm::ModelMetaTag
  - Udm::ReferenceMetaTag
  - Udm::SetMetaTag
  - Udm::UnknownMetaTag
- a set of concept checking classes that can be used with the old and deprecated boost::function_requires function to perform concept checking; these classes are defined in UdmMPL.h
- generateded C++ classes have nested types that represent their parent composition roles, the child composition roles, the association roles; the nested types have the following name:
  - parent composition roles: PR_<relation name>
  - child composition roles: CR_<relation name>
  - association roles: AR_<relation name>

    - association roles for an association class: ACE_<relation name>
- generated C++ classes have nested types that represent their composition and association relationships with other C++ classes; these nested types are Boost MPL vectors and they are the following:
    - // typelist for parent by returned type and role relations

Parents

    - // typelist for parent by returned type relations;

ParentKinds

    - // typelists for children by returned type and role relations

ChildrenSingle // for single child relationships
ChildrenMulti // for more than one child relationships

    - // typelist for children by returned type relations

ChildrenKinds

    - // typelists for associations by returned type and role relations

AssociationsSingle // if maximum cardinality is 1
AssociationsMulti // if maximum cardinality is greater than 1

    - // typelists for associations by returned type, association class and role relations

AssociationsWAClassSingle
AssociationsWAClassMulti

    - // typelists for cross associations by returned type and role relations

CrossAssociationsSingle
CrossAssociationsMulti

    - // typelists for cross associations by returned type, association class and role relations

CrossAssociationsWAClassSingle
CrossAssociationsWAClassMulti

    - // typelist for association class ends by returned type and role relations

AClassEnds

    - // typelist for cross association class by returned type and role relations

CrossAClassEnds

- the generated C++ classes have additional member function templates to retrieve the parent and the child(ren) of the class:
    - template <class ChildType, class RoleType>

Udm::ChildAttr<ChildType> child() const;

    - template <class ChildrenType, class RoleType, class Pred>
      Udm::ChildrenAttr<ChildrenType, Pred> children_sorted() const;

– template <class ChildrenType>
Udm::ChildrenAttr<ChildrenType> children_kind() const;
– template <class ChildrenType, class Pred>
Udm::ChildrenAttr<ChildrenType, Pred> children_kind_sorted() const;
– template <class ParentType, class RoleType>
Udm::ParentAttr<ParentType> parent() const;
– template <class ParentType>
Udm::ParentAttr<ParentType> parent_kind() const;

- the generated C++ classes have additional member function templates to retrieve the targets of associations:
  - template <class PeerType, class RoleType>
Udm::PointerAttr<PeerT
ype> peer() const;
  - template <class PeersType, class RoleType>
Udm::AssocAttr<PeersT
ype> peers() const;
  - template <class PeersType, class RoleType, class Pred>
Udm::Asso
cAttr<PeersType, Pred> peers_sorted() const;
  - template <class ConnectorType, class PeerType, class RoleType>
U
dm::AClassPointerAttr<ConnectorType, PeerType> connector() const;
  - template <class ConnectorsType, class PeerType, class RoleType>
Udm::AClassAssocAttr<ConnectorsType, PeerType> connectors() const;
  - template <class ConnectorsType, class PeerType, class RoleType, class Pred> Udm::AClassAssocAttr<ConnectorsType, PeerType, Pred> connectors_sorted() const;
  - template <class PeerType, class RoleType> Udm::CrossPointerAttr<PeerType> cross_peer() const;
  - template <class PeersType, class RoleType> Udm::CrossAssocAttr<PeersType> cross_peers() const;
  - template <class PeersType, class RoleType, class Pred> Udm::CrossAssocAttr<PeersType, Pred> cross_peers_sorted() const;
  - template <class ConnectorType, class PeerType, class RoleType> Udm::AClassCrossPointerAttr<ConnectorType, PeerType> cross_connector() const;
  - template <class ConnectorsType, class PeerType, class RoleType> Udm::AClassCrossAssocAttr<ConnectorsType, PeerType> cross_connectors() const;
  - template <class ConnectorsType, class PeerType, class RoleType, class Pred> Udm::AClassCrossAssocAttr<ConnectorsType, PeerType, Pred> cross_connectors_sorted() const;
  - template <class EndType, class RoleType>
Udm::AssocEndAttr<EndType> end() const;

- template <class EndType, class RoleType> Udm::CrossAssocEndAttr<EndType> cross_end() const;

– in the diagram C++ namespace, there is a type IsDescendant<T, U> that can be used to check at compile time if type U is a descendant (in the parent-child sense) of type T.

To generate the extended API, Udm must be invoked with parameter “--leesa”.

Since the compile time can increase very much in case of large UML diagrams, the extra generated code is enclosed by macro guards that can be used to ignore the unneeded parts:

– if preprocessor macro MPL_NO_COMPOSITIONS is defined, then the types, typelists and member function templates for composition relationships are not compiled
– if preprocessor macro MPL_NO_ASSOCIATIONS is defined, then the types, typelists and member function templates for association relationships are not compiled
– if preprocessor macro PARADIGM_HAS_DESCENDANT_PAIRS is defined, then the IsDescendant type and its specializations are visible to the compiler

`set<Bulb> s_bulb =`
` lamp.Bulb_kind_children();`
	`set<Bulb> s_bulb =`
	` lamp.children_kind<Bulb>();`

`lamp.ControlLink_children() = cl_test_set;`	`lamp.children<ControlLink,`
	`Lamp::CR_ControlLink_children>() =`
	` cl_test_set;`

`Lamp lamp1 =`
` switch1.MainSwitch_Lamp_parent();`
	`Lamp lamp1 = switch1.parent<Lamp,`
	` Switch::PR_MainSwitch_Lamp_parent>();`

`wire1.End1_end() = plug_terminal1;`	`wire1.end<ElectricTerminal,`
	` Wire::ACE_End1>() = plug_terminal1;`

`control_link3.src_end() = bulb3;`	`control_link3.end<Bulb,`
	` ControlLink::ACE_src>() = bulb3;`

Regular UDM API examples	Extended UDM API examples

Figure 1-examples of extended UDP API vs. regular UDM API constructs

10. jUDM

The jUDM is a java wrapper for the UDM core technology. It generates Domain-Specific API in java language.

10.1 Usage of jUDM

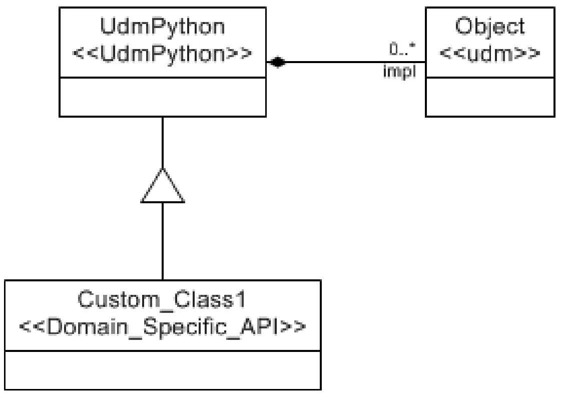
The java API generation is supported by *UDM.exe* with the *[–j]* option. The generated classes provide all necessary classes (and factories) to manipulate the data networks. The *FactoryRepository* class is a collection of namespace and root object specific data network factories and contains static method to access these factories. The *UDM* code generator tool generates two different types of data network factories for each root object in the data network as their name show: one for manipulating data networks via file and stream operations and one for manipulating data networks via string and stream operations. The factories provide common interfaces to manipulate the data network, such as *open()*, *create()*, *save()*, *close()*, and *checkConstraints()*.

11. Udm Python Interface

The Udm Python interface consists of:

1. Python module for interfacing with Udm
2. Python module providing the base class and domain-independent API for the Domain specific API
3. Python Domains Specific API   - and the generator

Basic structure is shown on the below figure:



At this moment the python API is read-only for structure (childrens, parents and associations) and read-write for attributes.

<u>11.2 UDM Python module</u>

The UDM Python module is a binary module written in C++ linked against the udm shared library.
The main function of this module is to forward calls from python Object to underlying C++ Udm Objects.
The scope of this module is:
-to provide a Python interface to underlying Udm::Obect and Udm::ObjectImpl C++ APIs
-to provide the required functionality to higher level APIs on descendant classes.
-to provide the necessary functions for initializing the meta-objects in Python Domain-Specific API

It is imported in python by :

```
import udm;
```

The following python language elements are exposed by this module:

1. class udm.Object – this class implements the most important access functions for structure and attributes at generic level: obtaining children, adjacent and parent, archetype, instance and derived objects, setting/getting attributes. It is not intended to use this class directly, this class is used mainly by the container class UdmPython and it‘s descendants in domain-specific API.

2. class udm.SmartDataNetwork – this class implements all the necessary calls to open, close and save udm Data networks of any type.
3. several constants, like uml_diagram, null, TARGETFROMCLASS, CLASSFROMTARGET, TARGETFROMPEER
4. utility and meta object initializer functions: CopyObjectHierarchy, InitChildRole, InitParentRole, GetUmlAssocRole

11.3 UdmPython - base class module

The UdmPython module is written in Python and it is main scope is to define the UdmPython base class. This base class is the Python equivalent of the Udm::Object C++ class and serves as a base class for all the domain specific API classes.

In case of non-array attributes the UdmPython base class handles the access to an object’s attributes in a generic way, through the python-specific __getattr__ and __setattr__ special functions are used to set/get any attributes. In case of array attributes, special helper classes are used.
It also defines helper classes for array attributes wrapper functions for the meta initializer functions

It is imported in python by:

```
import UdmPython
```

The UdmPython module exposes the following classes:

```
class UdmPython(object)
```

This is the base class of all the classes from DS APIs.
The most important functions are:

```
	def _get_parent(self, parent_role=None):
```

get the parent object or null if object has no parent
/ if role is not None, return parent only if the role actually applies
parent_role can be provided either as an udm.Object or UdmPython.UdmPython (or any descendants). Usually, one will provide Uml.CompositionChildRole as parent_role.

```
def _get_children(self,child_role=None, child_type=None):
```

if child_role is given, return only those children that have that role (ignore kind)
else if child_type is not null, return all children which are compatible with child_type
else if child_type is null, return all children
child_role and child_type can be provided either as an udm.Object or UdmPython.UdmPython (or any descendants). Usually, one will an object of Uml.CompositionChildRole as child_role and an objct of type Uml.Class as child_type.

```
def _get_associations(self, assoc_role, mode):
```

assoc_role must be non-null
return the adjacent objects via rolename provided by assoc_role
in case of association class based associations, mode determines which object is returned
- if self is either end of an association class based association and if mode == udm.TARGETFROMPEER than the other end of the association is returned.
-if self is either and of an association class based association and if mode == udm.CLASSFROMTARGET than the other end of the association is returned
-if self is the instance of the association class, then the only accepted value for mode is udm.TARGETFROMCLASS, and the returned object is one end of the association, as specified by assoc_role. In this case, a single object is returned

the returned value's type is always a list ([]). The python-type of the objects in the list is UdmPython.UdmPython.

assoc_role can be provided either as an udm.Object or UdmPython.UdmPython (or any descendants).
Usually, one will provide an object of type Uml.AssociationRole as assoc_role.

`def _type(self):`

returns the UDM type of an object. The return value is always an instances of `Uml.Class.`

`def setIndent(self,i):`

works together with `__str__.`

the `__str__` function will insert `i` times `TAB` before each line in the returned string.

Special – python specific- methods implemented by this class:

`def __init__(self, impl):`

Object initializer.

An instance of the `UdmPython` class, or any descendant, can be initialized either from an instance of `udm.Object` or from an instance of `UdmPython.UdmPython` (or any descendant = an DSP API class)

`def __eq__(self, other):`

true if two objects (self and other) point to the same UDM object.

`def __str__(self):`

textual representation of an object. The type of the object, the UDM Object ID and all attributes are listed.

`def __getattr__(self):`
`def __setattr__(self):`

Attribute getter and setter.

These generic python functions will catch any attempt to set/get an object's attribute. If the attribute does exist in the UDM object as well, the UDM object's attribute will be returned/set.

Actually this is a powerful way to provide DS attribute getter/setter in DS API without having to define these functions in the DS API.

`def __nonzero__(self):`

true if the objects is not the null UDM object.

`class ArrayAttr`
`(and descendants: StrArrayAttr, IntArrayAttr, RealArrayAttr and BooleanArrayAttr)`

This is the base class of array attribute helper classes `StrArrayAttr, IntArrayAttr, RealArrayAttr and BooleanArrayAttr.`

Instances of these classes are returned by DS API whenever an array attribute is accessed.

These classes implement the following methods and properties:

```
def __init__(self, obj, attr):
```

Object initializer.
Any UdmPython or descendants instance is expected as obj.
A Uml.Attribute is expected as attr.

```
@property
def array(self):
@array.setter
def array(self, val):
```

This property can be used to get / assign the attribute as an array.

```
def setIndent(self,i):
```

works together with `__str__.`
the `__str__` function will insert `i` times `TAB` before each line in the returned string.

```
def __str__(self):
```

textual representation of an array attribute. all items are listed.

```
def __getitem__(self,index):
def __setitem__(self, index, value):
```

these functions are used to set/get an individual item in an array, with the [] operator

and the following functions, used for initializing UML meta pointers on python side. These functions are used to initialize the meta objects in the DS API after loading the UML Diagram from an external file.

```
def GetUmlClassByName(dgr, name):
```

Given a `Uml.Diagram`, find a class by it's name. usually, `Uml.Diagram` is loaded from an external XML file. The function will return the `Uml.Class` object from `Uml.Diagram dgr` which has the given name.

`def GetUmlAttributeByName(cl, name):`

Given a Uml.Class, find an attribute by it's name. The function will return the Uml.Attribute attribute from the attributes of Uml.Class which has the given name.

`def GetUmlChildRoleByTypesAndRolenames(child_type, parent_type, crole_name, prole_name):`

Given the type of child and parent (as `Uml.Class`), returns the `Uml.CompositionChildRole` for the matching composition. Possible compositions can be constrainted composition child role name `crole_name` and composition parent role name `prole_name`. These parameters could be `None`.
The function will throw a runtime error exception if no such composition is, or, there is more than one composition matching the criteria. The function uses `matchChildToParent()` from UmlExt API.

`def GetUmlAssocRoleByPeerAndRole(my_type, peer_type, arole_name):`

Given the type of peers (`my_type` and `peer_type` as `Uml.Class`), returns the `Uml.AssociationRole` for the matching association. Possible associations can be constrainted with association role name `arole_name`, this will filter the associationroles pointing to `my_type`. This parameter can be `None`.

The function works only for associations that are not association class based.

The function will throw a runtime error exception if no such association is, or, there is more than one association matching the criteria. The function uses `matchPeerToPerr()` from UmlExt API.

`def GetUmlAssocRoleByAClassAndRole(my_type, aclass_type, arole_name):`

Given the type of one peers (`my_type`) and an association class(`aclass_type`), both provided as `Uml.Class`, returns the `Uml.AssociationRole` for the matching association. Possible associations can be constrainted with association role name `arole_name`, this will filter the associationroles pointing to `my_type`. This parameter can be `None`.

The function works only for associations that are association class based.

The function will throw a runtime error exception if no such association is, or, there is more than one association matching the criteria. The function uses `matchPeerToPerr()` from UmlExt API.

11.3 UdmPython - Domain Specific API (DS API)

In a quite similar way as it is in C++, Java, C#, the Python DS API is generated based on the UML Diagram in XML format by the Udm utility. (by adding the –p switch to the command line).

The python DS API relies on the UdmPython and udm modules.

The DS API contains classes for each UML class, with functions for parents, children, association, attributes.

It is important to note that all the DS API functions below that return a single object or a collection of objects are always correctly typed in Python, which means that the python type of the returned object is always the most specific one.

11.3.4 Parent objects:

For each parentrole, there is a function to access the object's parent via that parentrole:

```
class Wire(UdmPython):
	…
	def ElectricDevice_parent(self)
```

If the class does not have defined a parentrole named "parent" then additionally and by-default parent function will be also generated:

```
class Wire(UdmPython):
	…
	def parent(self)
```

11.3.5 Children objects:

For each childrole, there is a function to access the object's children via that specific childrole. This functions are named with endings `_children` or `_child`, based on the cardinality of the child role.

```
class ElectricDevice(UdmPython):
	…
	def Wire_children(self)
```

There is also a function for each possible kind of object that can be contained inside a class: These functions are named with endings `_kind_children`.

```
class ElectricDevice(UdmPython):
     …
     def Wire_kind_children(self)
     def ElectricTerminal_kind_children(self)
```

<u>11.3.6 Adjacent objects:</u>

For associations, the following functions are generated:

In case of associations without an association class, there are simple functions at both classes for accessing the other end(peer) object, like in the below example from Uml.py: These functions have the same names as the respective AssociationRoles.

```
class Class(UdmPython):
     …
     def association(self):

class Association(UdmPython):
     …
     def assocClass(self):
```

In case of associations with an association class, there are generated functions in all three classes: the two legs and the in-the-middle association class.

At the legs the functions are similar, with the associations without an association class

```
class Bulb(ElectricDevice):
          …
          def dst(self)


class Switch(ElectricDevice):
          …
          def src(self)
```

The association class has functions for getting both ends:

```
class ControlLink(ElectricDevice):
          …
          def src_end(self)
```

```
                def dst_end(self)
```

11.3.7 Attributes:

For non-array attributes, they are no generated functions in classes of DS APIs. In these cases the `__getattr__` and `__setattr__` functions are used. This is very pytonish way to provide attribute setter and getter functions, it is not necessary to know the name of the attribute in advance. Any undefined function name will finally end up in `__getattr__` or `__setattr__` and an attribute named in the same way exists for that object, than the operation will be carried out successfully, otherwise an exception will be raised.

For array attributes, the situation is complicated and therefore helper classes are used, which are defined in UdmPython: `StrArrayAttr`, `IntArrayAttr`, `RealArrayAttr` and `BooleanArrayAttr`.,

These classes make Udm array attributes look like Python arrays, with `__getitem__`, `__setitem__` functions implemented. These helper classes also encode/decode array attributes to ; delimited strings, this is how they are stored in UDM data structures.

In case of array attributes, there will be generated functions:

```
      class Lamp(ElectricDevice)
            …
            def annotations(self)# this will return an
StrArrayAttr
```

11.3.8 Object factory:

Each class has a

```
      @staticmethod:
      def cast(obj)
            …
```

which will create an object of a given type either from an UdmPython object or directly an udm.Object.

11.3.10 Inheritence:

The inheritance relationships in the Uml Diagram are fully represented in the generated python DS-API. In our Lamp example we have:

```
      class UdmPython(object)             #in module UdmPython.py
      class ElectricDevice(UdmPython)     #in module LampDiagram.py
      class Bulb(ElectricDevice)          #in module LampDiagram.py
```

```
class HalogenBulb(Bulb)            #in module LampDiagram.py
```

11.3.10 Meta objects:

The generated DS API holds static variables to metaobjects of the DS Uml Diagram. Therefore, all the metaobjects are accessible from Python by navigation from certain "anchor-objects" which are pointed by static variables in the DS API.

The Meta objects defined in a Python DS API class are:

`Meta` – instance of `Uml.Class`, points to the type of the object
`meta_<name_of_attribute>` - instance of `Uml.Attribute`, points to the respective `Uml::Attribute` meta-object
`meta_<childrole_name>_children` – instance `of Uml.CompositionChildRole,` points to the respective `Uml::CompositionChildRole` meta-object
meta_<parentrole_name>_parent, instance of `Uml.CompositionParentRole`, points to the respective Uml::CompositionParentRole meta-object
`meta_<assoc_role>`, and `meta_<assoc_role>_rev` instances of `Uml.AssociationRole`, point to the respective `Uml::AssociationRole` meta-object and it's pair

The association classes also have static variables that point to the metaobject for both ends:

`meta_<assoc_role>_end_`, instance of `Uml.AssociationRole`, points to the respective `Uml::AssociationRole` meta-object

`meta_<assoc_role>_end_`, instance of `Uml.AssociationRole`, points to the respective `Uml::AssociationRole` meta-object

11.3.10 Typical application

The first step in creating a python script using Udm is to generate uml.py and <diagram>.py from the Uml Diagrams.

```
Udm –p Uml_udm.xml
Udm –p LampDiagram.xml
```

Then, a typical script would begin with these import lines:

```
import udm        #C++ Python modules included in Udm
import UdmPython  #Python module included in Udm
import Uml        #generated
import LampDiagram      #generated
```

Then, the next step is to open the meta-data network. This is needed because – in contrary to the C++ version – the domain specific meta objects are not compiled in, they have to be loaded runtime.

```
pdn = udm.SmartDataNetwork(udm.uml_diagram())
pdn.open(r"LampDiagram_uml.xml")
LampDiagram.Initialize(pdn.root)
```

Now we have a meta network in place, so we can open an instance network.

```
dn= udm.SmartDataNetwork(pdn.root)
dn.open(r"Lamp.mem")
```

At this point we have successfully opened a data network and we have a pointer to the root object. We can navigate through the datanetwork by the access functions to children, parent and adjacent objects:

Children:

```
Rf = Lampdiagram.RootFolder(dn.root)
l_children = rf.Lamp_kind_children()
if l_children:
      print "I have found the following lamps:"
      print lamp.ModelName
      print lamp.name
```

parent: In this example, rf1 is the same object as rf, and rf1 is also correctly type, it's python type will be LampDiagram.RootFolder

```
l_children = rf.Lamp_kind_children()
lamp_1 = l_children[0]
rf1 = lamp_1.parent
print type(rf1)
```

Associations: In this example, Bulb, Switch and ControlLink compose an association class based association, whete ControlLink is the association class.

```
Bulbs = lamp_1.Bulb_kind_children()
For bulb in bulbs:
      cl = bulb.dst() #  this is a collection of ControlLinks
      if len(cl):
            sw = cl[0].dst_end() # this will be a single switch
            cl1 = sw.src() #again, a collection of ControlLinks,
obtained from the other endpoint
```

```
                # at this point, cl1[0] is the same object as cl[0]
                bulb1 = cl1[0].src_end() # this will be  bulb, and
      points to the same object as bulb
```

Attributes:

Accessing non-array attributes is as simple as possible:

```
      print lamp_1.ModelName              #read
      lamp_1.ModelName = “test”           #write
```

Array attributes work like python arrays:

```
      o = lamp_1.ArrayStr()
      # list all items using __str__ function of class ArrayAttr
      print o
      # access a single item (the third one)
      print o[2]
      o[2]  = “hello python”
      # access as an array
      print o.array
      o.array = [“alma”, “korte”, “szilva”]
      # checking the length
      print len(o.array)
```

Working with UdmPython API: The function below will traverse recursively all the hierarchy and will print all the objects found. The children are obtained with the `_get_children()` of UdmPython class. The `print obj` call will use the `__str__` function in UdmPython to provide a formatted output with all information from object. This works together with `setIndent`, which sets the number of tab-s to be printed before the actual text.

```
def walk_hierarchy_udmp(obj, indent_tabs = 0):
    #obj is expected to be UdmPython.UdmPython or any DS class
    #walks recursively the hierarchy

    if obj:

        obj.setIndent(indent_tabs)
        print obj
        generic_children = obj._get_children()
        if generic_children:
            print UdmPython.indent(indent_tabs) + "I have found the
following children at this level: %d" % (len(generic_children))
            for child in generic_children:
                walk_hierarchy_udmp(child, indent_tabs+1)
    else:
        return
```

Working with metaobjects:

Accessing the names of association roles in the Switch – ControlLink – Bulb association:

```
print "Switch.meta_src: "  + LampDiagram.Switch.meta_src.name  #this
should be src
print "Bulb.meta_dst: "  + LampDiagram.Bulb.meta_dst.name      #this
should be dst
```

Checking the inheritances relationship between ElectricDevice and it’s descendants:

```
print "descendants of ElectricDevice are: "
classes = LampDiagram.ElectricDevice.Meta.subTypes()
for cl in classes:
    print cl.name
    print cl.baseTypes()[0] == LampDiagram.ElectricDevice.Meta #true!
```

Appendix A - UDM tools documentation

Udm

```
NAME
      Udm - generate API and XSD/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 <diagramfilename> [<genfilesnamebase>]
            [-d <Uml.XSD searchpath>] [-m|c] [-v] [-t] [-l <macro>] [-
x]

      <diagramfilename >            The name of the input XML document
                        (Conformant to the Uml.xsd file)
      <genfilesnamebase >           The basename of the output
documents
                        (.h, .cpp, .xsd/.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.
      -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.
      -j                Switch Udm into generate Java code mode.
      --leesa           Generate C++ with support for generic
                        programming.

      -t                Generate DTD instead of XSD.
      -x                The generated XSD file will correctly reflect
                        the inheritance relationships and the
                        abstractness of types, as described in the UML
                        Diagram. However the generated XSD file is not
                        suitable for validation of UDM DOM
                        Datanetworks.
      -e                The XSD elements are generated like they would
                        have text attributes.
      -g                Integrate the generated XSD file into a C++
                        header file. The XSD will be loaded
                        automatically when the diagram is initialized.

      -d                the path where the DTD file are searched. If
                        not provided, the PATH is searched if a DTD is
                        needed.
      -u UML name=URI   Map UML Namespace name to URI.
      -i name           All the contained elements in a namespace given
                        here will be ignored during parsing.
      -q name           All the contained attributes in a namespace
                        given here will be prefixed with the UML
                        namespace.

      -T                Don't record the timestamp when code was
                        generated.
      -w[c|d|n]         How to group classes and namespaces into files:
                        -wc: One .cpp file per class
                        -wn: All classes from a namespace into a single
                        .cpp file
                        -wd: All classes from a diagram and its sub-
                        namespaces into a single .cpp file
      -o                An output directory where to save the generated
                        sources.


      -v                Generate code to support the visitor design
                        pattern. A Visitor skeleton class is generated
                        in each namespace and in the diagram too.
      -l macro_name     A custom macro name is included before class
                        declarations. This makes possible to export all
                        classes in a DLL for example.
```

DESCRIPTION

Udm generates API and .xsd files from an XML representation of a UML diagram. These XML-s are usually extracted from 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.

Udm.exe also contains Uml.xsd as a resource, so it is not necessary to have it available upon loading and XML representation of a UML diagram.

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|mem|xml]>
<backend_meta_file.xml>

      <backend_file.[mga|mem|xml]>  The name of the GME|MEM|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.
```

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|mem|xml]> <backend_meta_file.xml> <flags>

      <backend_file.[mga|mem|xml]> The name of the GME|MEM|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
```

UdmJson

```
NAME
      UdmJson – serializes a data backend to Json data and its
validating SCHEMA


AUTHOR
      Magyari Endre - endre@isis.vanderbilt.edu

SYNOPSIS

Usage: UdmJson [options] <InputDataNetwork> <UmlMetaDiagram>
<OutputJson> [<OutputSchema>]

Valid options are :
--always_escape_nonascii:All unicode wide characters are escaped, i.e.
outputed
as "\uXXXX", even if they are printable under the current locale, ascii
printabl
e chars are not escaped
--help:                 Produce help message
--pretty_print:         Add whitespace to format the output nicely
--remove_trailing_zeros: Outputs e.g. "1.200000000000000" as "1.2"
--single_line_arrays:    Pretty printing arrays on single lines unless
they contain composite elements, i.e. objects or arrays


InputDataNetwork is the datanetwork to be serialized
UmlMetaDiagram is the UML Meta of InputDataNetwork
OutputJson is the file where the serialized objects will be written.
OutputSchema – optional – if provided, a schema will be generated which
validates OutputJson.
```

UdmOclPat

NAME

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

AUTHOR

    Anantha Narayanan   - Ananth@isis.vanderbilt.edu

SYNOPSIS

```
      UdmOclPat <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

UdmOclPat 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 that are evaluated, and their result is inserted into the output. These instructions are in a modified form of OCL. The special instructions are separated from the plain text using the special tags "<:" and ":>".

The input Udm data can be navigated using standard OCL terms (eg. object.name). These may return primitive types such as strings or numbers, or return an object or a collection of objects. Operations can be performed on the returned values, such as print a string or number to the output, or iterate through a collection of objects. All script statements must end in a semicolon. Script blocks must be enclosed in curly braces. Some additional commands such as "print" and "open" have been added to provide functionality not available in standard OCL, such as print output or open files. These commands will be discussed below.

UdmOclPat always accesses Udm data in read-only mode.

UdmOclPat 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 OCL instructions are always evaluated in the context of a UDM object. The default context is the root object. This context may be changed, such as inside an iteration.

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

open(<filename>, <mode>, <handle>);

This command is used to open files for output.

Filename: This is a string that represents a file name. This can be a literal string, a string attribute or a concatenation of the two.

Mode: This specifies the mode to open the file in – "o" opens for output, and deletes any existing file with the same name. "a" opens in append mode, and all output is appended at the end of any existing data in the file. "f" indicates that the program must fail if a file with the same name already exists.

Handle: The handle is used to identify a file for directing output. Multiple files can be opened at the same time, and the output directed to teh appropriate handle.

e.g.:

```
<: open(obj.name + ".xml", "o", configFile); :>
```

switch(<handle>);

The open command only opens a file for output, but does not direct output to that file. The "switch" command is used to do that. The argument must be a valid "handle" that was defined using a previous "open" statement. Once a switch command is executed, all further output is sent to that file.

e.g.:

```
<: switch( configFile); :>
```

print(<arg>)

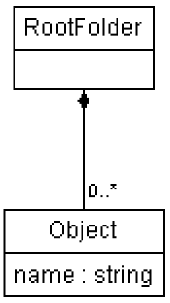
Print the value of <arg> to the output. <arg> can be a literal, an attribute with a printable value, or a concatenation of the two ("[undefined]" is printed in case the attribute or value is not printable).

e.g.:

```
<: print("The name of the object is: " + obj.name); :>
```

In addition to these three basic commands, several OCL statements can be used for navigating the data. Some of them are discussed below. You can find the details in the OCL section of the GME User's Manual, or in any book on OCL.

The "self" expression identifies the initial context, which is usually the root folder. All objects can be navigated from this context. The navigation can be done according to the UML meta-model which you provide to the pattern processor. Consider the simplified meta shown below:



The root folder can contain any number of “object” classes. Since “self” denotes the root folder, we can use “self.object” to navigate these objects. Note that “object” is used with a lowercase “o”. The UDM standard is to change the first letter of the classname to lowercase, in order to navigate contained objects.

In this case, the expression “self.object” returns a collection of classes. These can in turn be iterated using the standard OCL iterators. “forAll” is one such iterator. The syntax for its usage in the pattern processor (which is slightly different from the standard syntax) is:

Collection->forAll(*iter* | { <ocl expression block> });

Where “collection” is the OCL collection which is being iterated and “iter” is the iteration variable or loop variable (it refers to the instance from the collection for each iteration). The “ocl expression block” can be any valid set of pattern instructions or plain text with its own appropriate syntax. This has to be enclosed in curly braces, even if it is a single OCL expression. Also notice that the forAll statement itself must be ended with a semicolon.

The usage of the “forAll” iterator is shown below:

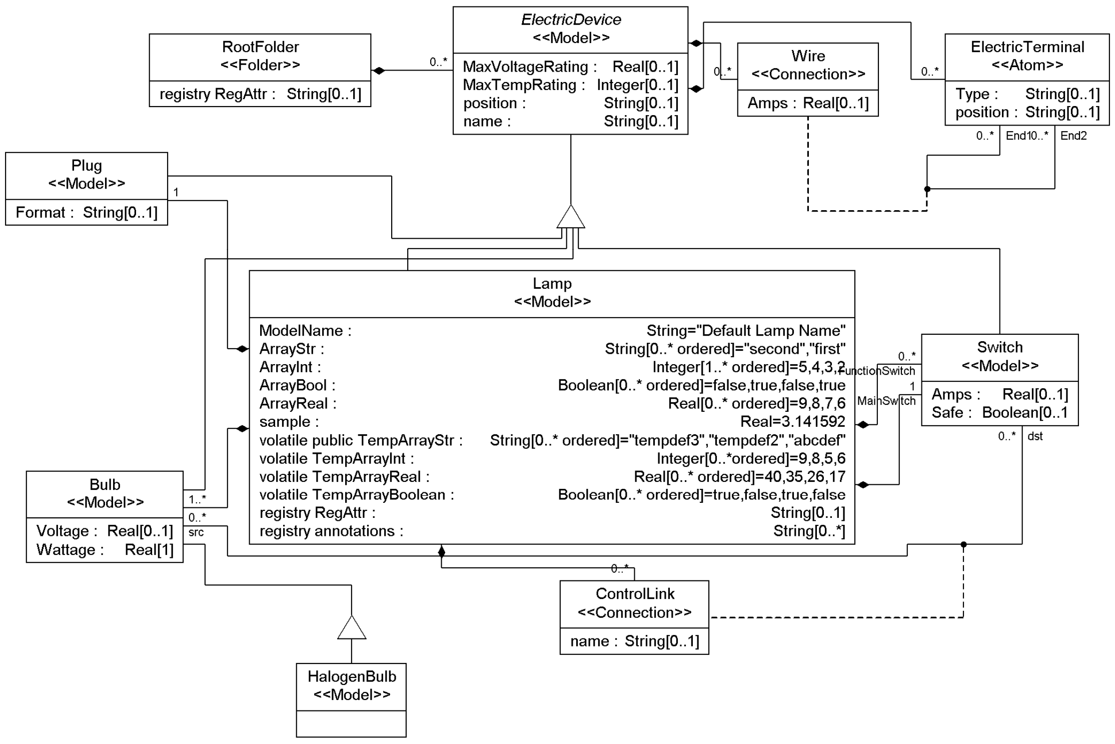
```
<:
      self.object->forAll( obj | { print(“Obj name: “ + obj.name); } );
:>
```

This iterates through the collection of all “object” classes in the root folder, and prints the value of the “name” attribute of each instance.

UdmCopy

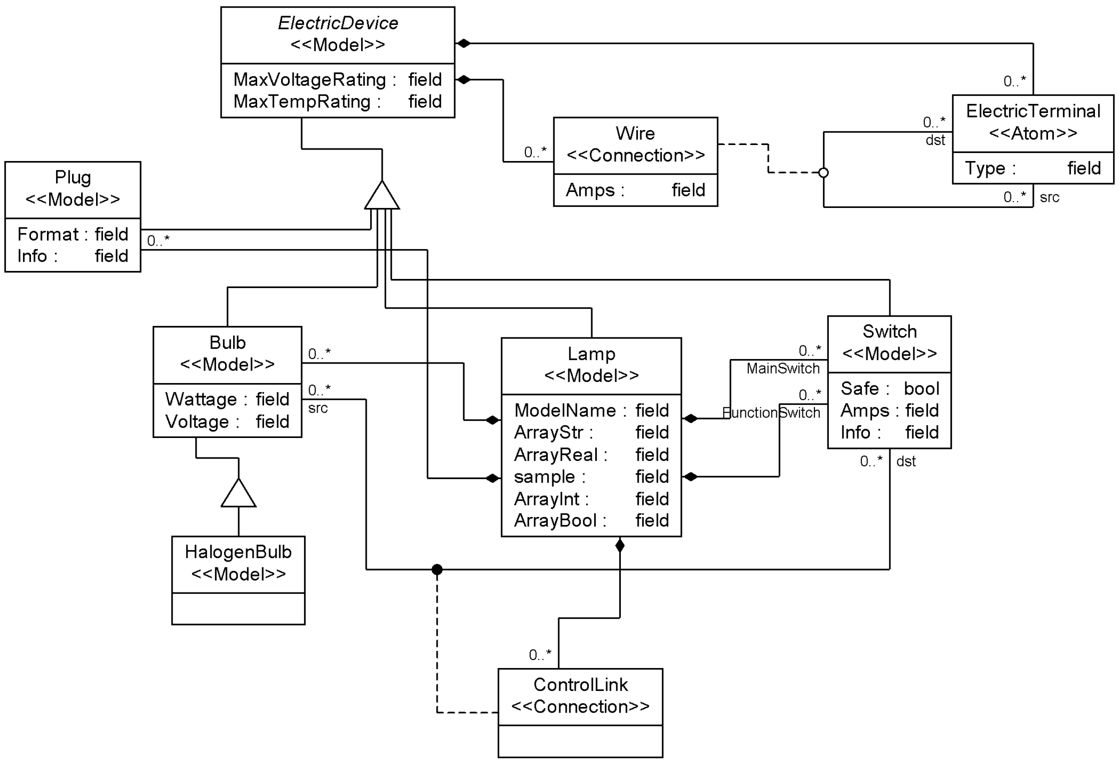
```
NAME
      UdmCopy - replicate a data network in another persistence engine.

AUTHOR
      Arpad Bakay    - arpad.bakay@vanderbilt.edu

SYNOPSIS
      UdmCopy <indataname> <outdataname> <diagramname> [<metalocator>]

      <indataname>  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.
      <outdataname> 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.
      <diagramname> the XML representation of the UML diagram that
                    describes the data structure to be copied.


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

FILES
uml.xsd :                the <diagramname> 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.
Example:

            UdmCopy l1.xml l2.xml   LampDiagram.xml

      This copies the existing datanetwork (l1.xml), which is an
instance of the UML diagram described in LampDiagram.xml. The copy
datanetwork resides in l2.xml.
```

Appendix B - Simple UML class diagram, created in GME



ElectricDevice
<<Model>>
MaxVoltageRating : Real[0..1]
MaxTempRating : Integer[0..1]
position : String[0..1]
name : String[0..1]
RootFolder
<<Folder>>
registry RegAttr : String[0..1]
0..*
Wire
<<Connection>>
Amps : Real[0..1]
0..*
ElectricTerminal
<<Atom>>
Type : String[0..1]
position : String[0..1]
0..*
0..* End1
0..* End2
Plug
<<Model>>
Format : String[0..1]
1
Lamp
<<Model>>
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
sample : Real=3.141592
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
registry RegAttr : String[0..1]
registry annotations : String[0..*]
0..*
FunctionSwitch
1
MainSwitch
Switch
<<Model>>
Amps : Real[0..1]
Safe : Boolean[0..1
0..*
dst
Bulb
<<Model>>
Voltage : Real[0..1]
Wattage : Real[1]
1..*
0..*
src
0..*
ControlLink
<<Connection>>
name : String[0..1]
HalogenBulb
<<Model>>

Appendix C - Matching GMEMeta paradigm of the UML ClassDiagram



Appendix D - XML representation of UML information.
(Created with UML2XML GME Interpretter from the Diagram in Appendix B)

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<Diagram name="LampDiagram" version="1.00" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Uml.xsd">


  <Association _id="id1D" assocClass="id16" name="ControlLink" nonpersistent="false">

    <AssociationRole _id="id1e" isNavigable="true" isPrimary="false" max="-1" min="0" name="src"
target="id5"/>

    <AssociationRole _id="id1F" isNavigable="true" isPrimary="false" max="-1" min="0" name="dst"
target="id7"/>

  </Association>
  <Association _id="id20" assocClass="idC" name="Wire" nonpersistent="false">

    <AssociationRole _id="id21" isNavigable="true" isPrimary="false" max="-1" min="0" name="End1" 
target="id9"/>

    <AssociationRole _id="id22" isNavigable="true" isPrimary="false" max="-1" min="0" name="End2" 
target="id9"/>

  </Association>



  <Class _id="id3" baseTypes="id4" childRoles="id14" isAbstract="false" name="Plug" 
stereotype="Model">

    <Attribute max="1" min="0" name="Format" nonpersistent="false" ordered="false" registry="false" 
type="String" visibility="public"/>

  </Class>



  <Class _id="id5" associationRoles="id1e" baseTypes="id4" childRoles="id19" isAbstract="false" 
name="Bulb" stereotype="Model" subTypes="id6">

    <Attribute max="1" min="0" name="Voltage" nonpersistent="false" ordered="false" registry="false" 
type="Real" visibility="public"/>

    <Attribute max="1" min="1" name="Wattage" nonpersistent="false" ordered="false" registry="false" 
type="Real" visibility="public"/>

  </Class>



  <Class _id="id7" associationRoles="id1F" baseTypes="id4" childRoles="id12 id1B" isAbstract="false" 
name="Switch" stereotype="Model">

    <Attribute max="1" min="0" name="Amps" nonpersistent="false" ordered="false" registry="false" 
type="Real" visibility="public"/>

    <Attribute max="1" min="0" name="Safe" nonpersistent="false" ordered="false" registry="false" 
type="Boolean" visibility="public"/>

  </Class>



  <Class _id="idC" association="id20" childRoles="idD" isAbstract="false" name="Wire" 
stereotype="Connection">

    <Attribute max="1" min="0" name="Amps" nonpersistent="false" ordered="false" registry="false" 
type="Real" visibility="public"/>

  </Class>



  <Class _id="id4" childRoles="idF" isAbstract="true" name="ElectricDevice" parentRoles="idB ide" 
stereotype="Model" subTypes="id3 id5 id7 id8">

    <Attribute max="1" min="0" name="MaxVoltageRating" nonpersistent="false" ordered="false" 
registry="false" type="Real" visibility="public"/>

    <Attribute max="1" min="0" name="MaxTempRating" nonpersistent="false" ordered="false" 
registry="false" type="Integer" visibility="public"/>

    <Attribute max="1" min="0" name="position" nonpersistent="false" ordered="false" registry="false" 
type="String" visibility="public"/>

    <Attribute max="1" min="0" name="name" nonpersistent="false" ordered="false" registry="false" 
type="String" visibility="public"/>

  </Class>



  <Class _id="id10" isAbstract="false" name="RootFolder" parentRoles="id11" stereotype="Folder">
```

```
    <Attribute max="1" min="0" name="RegAttr" nonpersistent="false" ordered="false" registry="true"
type="String" visibility="public"/>

  </Class>


  <Class _id="id9" associationRoles="id21 id22" childRoles="idA" isAbstract="false"
name="ElectricTerminal" stereotype="Atom">

    <Attribute max="1" min="0" name="Type" nonpersistent="false" ordered="false" registry="false"
type="String" visibility="public"/>

    <Attribute max="1" min="0" name="position" nonpersistent="false" ordered="false" registry="false"
type="String" visibility="public"/>

  </Class>


  <Class _id="id6" baseTypes="id5" isAbstract="false" name="HalogenBulb" stereotype="Model"/>


  <Class _id="id8" baseTypes="id4" isAbstract="false" name="Lamp" parentRoles="id13 id15 id18 id1A
id1C" stereotype="Model">

    <Attribute defvalue="Default Lamp Name;" max="1" min="1" name="ModelName" nonpersistent="false"
ordered="false" registry="false" type="String" visibility="public"/>

    <Attribute defvalue="second;first;" max="-1" min="0" name="ArrayStr" nonpersistent="false"
ordered="true" registry="false" type="String" visibility="public"/>

    <Attribute defvalue="5;4;3;2;" max="-1" min="1" name="ArrayInt" nonpersistent="false"
ordered="true" registry="false" type="Integer" visibility="public"/>

    <Attribute defvalue="false;true;false;true;" max="-1" min="0" name="ArrayBool"
nonpersistent="false" ordered="true" registry="false" type="Boolean" visibility="public"/>

    <Attribute defvalue="9;8;7;6;" max="-1" min="0" name="ArrayReal" nonpersistent="false"
ordered="true" registry="false" type="Real" visibility="public"/>

    <Attribute defvalue="3.141592;" max="1" min="1" name="sample" nonpersistent="false"
ordered="false" registry="false" type="Real" visibility="public"/>

    <Attribute defvalue="tempdef3;tempdef2;abcdef;" max="-1" min="0" name="TempArrayStr"
nonpersistent="true" ordered="true" registry="false" type="String" visibility="public"/>

    <Attribute defvalue="9;8;5;6;" max="-1" min="0" name="TempArrayInt" nonpersistent="true"
ordered="true" registry="false" type="Integer" visibility="public"/>

    <Attribute defvalue="40;35;26;17;" max="-1" min="0" name="TempArrayReal" nonpersistent="true"
ordered="true" registry="false" type="Real" visibility="public"/>

    <Attribute defvalue="true;false;true;false;" max="-1" min="0" name="TempArrayBoolean"
nonpersistent="true" ordered="true" registry="false" type="Boolean" visibility="public"/>

    <Attribute max="1" min="0" name="RegAttr" nonpersistent="false" ordered="false" registry="true"
type="String" visibility="public"/>

    <Attribute max="-1" min="0" name="annotations" nonpersistent="false" ordered="false"
registry="true" type="String" visibility="public"/>

  </Class>


  <Class _id="id16" association="id1D" childRoles="id17" isAbstract="false" name="ControlLink"
stereotype="Connection">

    <Attribute max="1" min="0" name="name" nonpersistent="false" ordered="false" registry="false"
type="String" visibility="public"/>

  </Class>


  <Composition name="Composition" nonpersistent="false">
```

```
    <CompositionChildRole _id="idA" isNavigable="true" max="-1" min="0" target="id9"/>

    <CompositionParentRole _id="idB" isNavigable="true" target="id4"/>

  </Composition>


  <Composition name="Composition" nonpersistent="false">

    <CompositionChildRole _id="idD" isNavigable="true" max="-1" min="0" target="idC"/>

    <CompositionParentRole _id="ide" isNavigable="true" target="id4"/>

  </Composition>


  <Composition name="Composition" nonpersistent="false">

    <CompositionChildRole _id="idF" isNavigable="true" max="-1" min="0" target="id4"/>

    <CompositionParentRole _id="id11" isNavigable="true" target="id10"/>

  </Composition>


  <Composition name="Composition" nonpersistent="false">

    <CompositionChildRole _id="id12" isNavigable="true" max="1" min="1" name="MainSwitch"
target="id7"/>

    <CompositionParentRole _id="id13" isNavigable="true" target="id8"/>

  </Composition>


  <Composition name="Composition" nonpersistent="false">

    <CompositionChildRole _id="id14" isNavigable="true" max="1" min="1" target="id3"/>

    <CompositionParentRole _id="id15" isNavigable="true" target="id8"/>

  </Composition>


  <Composition name="Composition" nonpersistent="false">

    <CompositionChildRole _id="id17" isNavigable="true" max="-1" min="0" target="id16"/>

    <CompositionParentRole _id="id18" isNavigable="true" target="id8"/>

  </Composition>


  <Composition name="Composition" nonpersistent="false">

    <CompositionChildRole _id="id19" isNavigable="true" max="-1" min="1" target="id5"/>

    <CompositionParentRole _id="id1A" isNavigable="true" target="id8"/>

  </Composition>


  <Composition name="Composition" nonpersistent="false">

    <CompositionChildRole _id="id1B" isNavigable="true" max="-1" min="0" name="FunctionSwitch"
target="id7"/>

    <CompositionParentRole _id="id1C" isNavigable="true" target="id8"/>

  </Composition>
```

```
</Diagram>
```

Appendix E - XSD file generated from the XML in Appendix D (generated with Udm.exe)

```
<?xml version="1.0" encoding="UTF-8"?>

<?udm interface="LampDiagram" version="1.00"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"

 elementFormDefault="qualified"

>

<!-- generated on Mon Jan 31 07:39:30 2011 -->




        <xsd:complexType name="PlugType">

                <xsd:sequence>

                        <xsd:element name="ElectricTerminal" type="ElectricTerminalType"
minOccurs="0" maxOccurs="unbounded"/>

                        <xsd:element name="Wire" type="WireType" minOccurs="0"
maxOccurs="unbounded"/>

                </xsd:sequence>

                <xsd:attribute name="MaxTempRating" type="xsd:long"/>

                <xsd:attribute name="MaxVoltageRating" type="xsd:double"/>

                <xsd:attribute name="position" type="xsd:string"/>

                <xsd:attribute name="name" type="xsd:string"/>

                <xsd:attribute name="Format" type="xsd:string"/>

                <xsd:attribute name="_id" type="xsd:ID"/>

                <xsd:attribute name="_archetype" type="xsd:IDREF"/>

                <xsd:attribute name="_derived" type="xsd:IDREFS"/>

                <xsd:attribute name="_instances" type="xsd:IDREFS"/>

                <xsd:attribute name="_desynched_atts" type="xsd:string"/>

                <xsd:attribute name="_real_archetype" type="xsd:boolean"/>

                <xsd:attribute name="_subtype" type="xsd:boolean"/>

        </xsd:complexType>


        <xsd:complexType name="BulbType">

                <xsd:sequence>

                        <xsd:element name="ElectricTerminal" type="ElectricTerminalType"
minOccurs="0" maxOccurs="unbounded"/>

                        <xsd:element name="Wire" type="WireType" minOccurs="0"
maxOccurs="unbounded"/>

                </xsd:sequence>

                <xsd:attribute name="MaxTempRating" type="xsd:long"/>

                <xsd:attribute name="MaxVoltageRating" type="xsd:double"/>

                <xsd:attribute name="position" type="xsd:string"/>
                <xsd:attribute name="name" type="xsd:string"/>

                <xsd:attribute name="Wattage" type="xsd:double" use="required"/>

                <xsd:attribute name="Voltage" type="xsd:double"/>

                <xsd:attribute name="dst" type="xsd:IDREFS"/>

                <xsd:attribute name="_id" type="xsd:ID"/>

                <xsd:attribute name="_archetype" type="xsd:IDREF"/>

                <xsd:attribute name="_derived" type="xsd:IDREFS"/>

                <xsd:attribute name="_instances" type="xsd:IDREFS"/>

                <xsd:attribute name="_desynched_atts" type="xsd:string"/>

                <xsd:attribute name="_real_archetype" type="xsd:boolean"/>

                <xsd:attribute name="_subtype" type="xsd:boolean"/>

        </xsd:complexType>


        <xsd:complexType name="HalogenBulbType">

                <xsd:sequence>

                        <xsd:element name="ElectricTerminal" type="ElectricTerminalType"
minOccurs="0" maxOccurs="unbounded"/>

                        <xsd:element name="Wire" type="WireType" minOccurs="0"
maxOccurs="unbounded"/>

                </xsd:sequence>

                <xsd:attribute name="MaxTempRating" type="xsd:long"/>

                <xsd:attribute name="MaxVoltageRating" type="xsd:double"/>

                <xsd:attribute name="position" type="xsd:string"/>

                <xsd:attribute name="name" type="xsd:string"/>

                <xsd:attribute name="Wattage" type="xsd:double" use="required"/>

                <xsd:attribute name="Voltage" type="xsd:double"/>

                <xsd:attribute name="dst" type="xsd:IDREFS"/>

                <xsd:attribute name="_id" type="xsd:ID"/>

                <xsd:attribute name="_archetype" type="xsd:IDREF"/>

                <xsd:attribute name="_derived" type="xsd:IDREFS"/>

                <xsd:attribute name="_instances" type="xsd:IDREFS"/>

                <xsd:attribute name="_desynched_atts" type="xsd:string"/>

                <xsd:attribute name="_real_archetype" type="xsd:boolean"/>

                <xsd:attribute name="_subtype" type="xsd:boolean"/>

        </xsd:complexType>


        <xsd:complexType name="SwitchType">

                <xsd:sequence>

                        <xsd:element name="ElectricTerminal" type="ElectricTerminalType"
minOccurs="0" maxOccurs="unbounded"/>

                        <xsd:element name="Wire" type="WireType" minOccurs="0"
maxOccurs="unbounded"/>
```

```
		</xsd:sequence>
		<xsd:attribute name="__child_as">
			<xsd:simpleType>
				<xsd:restriction base="xsd:string">
					<xsd:enumeration value="MainSwitch"/>
					<xsd:enumeration value="FunctionSwitch"/>
				</xsd:restriction>
			</xsd:simpleType>
		</xsd:attribute>
		<xsd:attribute name="MaxTempRating" type="xsd:long"/>
		<xsd:attribute name="MaxVoltageRating" type="xsd:double"/>
		<xsd:attribute name="position" type="xsd:string"/>
		<xsd:attribute name="name" type="xsd:string"/>
		<xsd:attribute name="Safe" type="xsd:boolean"/>
		<xsd:attribute name="Amps" type="xsd:double"/>
		<xsd:attribute name="src" type="xsd:IDREFS"/>
		<xsd:attribute name="_id" type="xsd:ID"/>
		<xsd:attribute name="_archetype" type="xsd:IDREF"/>
		<xsd:attribute name="_derived" type="xsd:IDREFS"/>
		<xsd:attribute name="_instances" type="xsd:IDREFS"/>
		<xsd:attribute name="_desynched_atts" type="xsd:string"/>
		<xsd:attribute name="_real_archetype" type="xsd:boolean"/>
		<xsd:attribute name="_subtype" type="xsd:boolean"/>
	</xsd:complexType>


	<xsd:complexType name="LampType">
		<xsd:sequence>
			<xsd:element name="Bulb" type="BulbType" minOccurs="0" 
maxOccurs="unbounded"/>
			<xsd:element name="ControlLink" type="ControlLinkType" minOccurs="0" 
maxOccurs="unbounded"/>
			<xsd:element name="ElectricTerminal" type="ElectricTerminalType" 
minOccurs="0" maxOccurs="unbounded"/>
			<xsd:element name="HalogenBulb" type="HalogenBulbType" minOccurs="0" 
maxOccurs="unbounded"/>
			<xsd:element name="Plug" type="PlugType"/>
			<xsd:element name="Switch" type="SwitchType" minOccurs="0" 
maxOccurs="unbounded"/>
			<xsd:element name="Wire" type="WireType" minOccurs="0" 
maxOccurs="unbounded"/>
		</xsd:sequence>
		<xsd:attribute name="MaxTempRating" type="xsd:long"/>
		<xsd:attribute name="MaxVoltageRating" type="xsd:double"/>
```

```
        <xsd:attribute name="position" type="xsd:string"/>

        <xsd:attribute name="name" type="xsd:string"/>

        <xsd:attribute name="ArrayStr" type="xsd:string" default="second;first;"/>

        <xsd:attribute name="ModelName" type="xsd:string" default="Default Lamp Name"/>

        <xsd:attribute name="ArrayInt" type="xsd:string" default="5;4;3;2;"/>

        <xsd:attribute name="ArrayBool" type="xsd:string" default="false;true;false;true;"/>

        <xsd:attribute name="ArrayReal" type="xsd:string" default="9;8;7;6;"/>

        <xsd:attribute name="sample" type="xsd:double" default="3.141592"/>

        <xsd:attribute name="RegAttr" type="xsd:string"/>

        <xsd:attribute name="annotations" type="xsd:string"/>

        <xsd:attribute name="_id" type="xsd:ID"/>

        <xsd:attribute name="_archetype" type="xsd:IDREF"/>

        <xsd:attribute name="_derived" type="xsd:IDREFS"/>

        <xsd:attribute name="_instances" type="xsd:IDREFS"/>

        <xsd:attribute name="_desynched_atts" type="xsd:string"/>

        <xsd:attribute name="_real_archetype" type="xsd:boolean"/>

        <xsd:attribute name="_subtype" type="xsd:boolean"/>

</xsd:complexType>


<xsd:complexType name="ElectricTerminalType">

        <xsd:attribute name="position" type="xsd:string"/>

        <xsd:attribute name="Type" type="xsd:string"/>

        <xsd:attribute name="End1" type="xsd:IDREFS"/>

        <xsd:attribute name="End2" type="xsd:IDREFS"/>

        <xsd:attribute name="_id" type="xsd:ID"/>

        <xsd:attribute name="_archetype" type="xsd:IDREF"/>

        <xsd:attribute name="_derived" type="xsd:IDREFS"/>

        <xsd:attribute name="_instances" type="xsd:IDREFS"/>

        <xsd:attribute name="_desynched_atts" type="xsd:string"/>

        <xsd:attribute name="_real_archetype" type="xsd:boolean"/>

        <xsd:attribute name="_subtype" type="xsd:boolean"/>

</xsd:complexType>


<xsd:complexType name="WireType">

        <xsd:attribute name="Amps" type="xsd:double"/>

        <xsd:attribute name="End1_end_" type="xsd:IDREF"/>

        <xsd:attribute name="End2_end_" type="xsd:IDREF"/>

        <xsd:attribute name="_id" type="xsd:ID"/>

        <xsd:attribute name="_archetype" type="xsd:IDREF"/>

        <xsd:attribute name="_derived" type="xsd:IDREFS"/>
```

```
                <xsd:attribute name="_instances" type="xsd:IDREFS"/>
                <xsd:attribute name="_desynched_atts" type="xsd:string"/>
                <xsd:attribute name="_real_archetype" type="xsd:boolean"/>
                <xsd:attribute name="_subtype" type="xsd:boolean"/>
        </xsd:complexType>

        <xsd:complexType name="RootFolderType">
                <xsd:sequence>
                        <xsd:element name="Bulb" type="BulbType" minOccurs="0"
maxOccurs="unbounded"/>
                        <xsd:element name="HalogenBulb" type="HalogenBulbType" minOccurs="0"
maxOccurs="unbounded"/>
                        <xsd:element name="Lamp" type="LampType" minOccurs="0"
maxOccurs="unbounded"/>
                        <xsd:element name="Plug" type="PlugType" minOccurs="0"
maxOccurs="unbounded"/>
                        <xsd:element name="Switch" type="SwitchType" minOccurs="0"
maxOccurs="unbounded"/>
                        <xsd:element name="RootFolder" type="RootFolderType" minOccurs="0"
maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="RegAttr" type="xsd:string"/>
                <xsd:attribute name="_id" type="xsd:ID"/>
                <xsd:attribute name="_archetype" type="xsd:IDREF"/>
                <xsd:attribute name="_derived" type="xsd:IDREFS"/>
                <xsd:attribute name="_instances" type="xsd:IDREFS"/>
                <xsd:attribute name="_desynched_atts" type="xsd:string"/>
                <xsd:attribute name="_real_archetype" type="xsd:boolean"/>
                <xsd:attribute name="_subtype" type="xsd:boolean"/>
                <xsd:attribute name="_libname" type="xsd:string"/>
        </xsd:complexType>

        <xsd:complexType name="ControlLinkType">
                <xsd:attribute name="name" type="xsd:string"/>
                <xsd:attribute name="src_end_" type="xsd:IDREF"/>
                <xsd:attribute name="dst_end_" type="xsd:IDREF"/>
                <xsd:attribute name="_id" type="xsd:ID"/>
                <xsd:attribute name="_archetype" type="xsd:IDREF"/>
                <xsd:attribute name="_derived" type="xsd:IDREFS"/>
                <xsd:attribute name="_instances" type="xsd:IDREFS"/>
                <xsd:attribute name="_desynched_atts" type="xsd:string"/>
                <xsd:attribute name="_real_archetype" type="xsd:boolean"/>
                <xsd:attribute name="_subtype" type="xsd:boolean"/>
        </xsd:complexType>

 <xsd:element name="RootFolder" type="RootFolderType"/>

</xsd:schema>
```

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 with Udm version 3.27 on


#include <UdmBase.h>


#if !defined(UDM_VERSION_MAJOR) || !defined(UDM_VERSION_MINOR)

#    error "Udm headers too old, they do not define UDM_VERSION"

#elif UDM_VERSION_MAJOR < 3

#    error "Udm headers too old, minimum version required 3.27"

#elif UDM_VERSION_MAJOR == 3 && UDM_VERSION_MINOR < 27

#    error "Udm headers too old, minimum version required 3.27"

#endif


#include <Uml.h>




#ifdef min

#undef min

#endif


#ifdef max

#undef max

#endif


namespace LampDiagram {


        extern ::Uml::Diagram meta;

        class Plug;

        class ElectricDevice;

        class Bulb;

        class HalogenBulb;

        class Switch;

        class Lamp;

        class ElectricTerminal;
        class Wire;
        class RootFolder;
        class ControlLink;

        class Visitor : public Udm::BaseVisitor {
        public:
                virtual ~Visitor() {}

                virtual void Visit_Plug(const Plug &) {}
                virtual void Visit_Bulb(const Bulb &) {}
                virtual void Visit_HalogenBulb(const HalogenBulb &) {}
                virtual void Visit_Switch(const Switch &) {}
                virtual void Visit_Lamp(const Lamp &) {}
                virtual void Visit_ElectricTerminal(const ElectricTerminal &) {}
                virtual void Visit_Wire(const Wire &) {}
                virtual void Visit_RootFolder(const RootFolder &) {}
                virtual void Visit_ControlLink(const ControlLink &) {}
                virtual void Visit_Object(const Udm::Object &) {}

        };

        void Initialize();
        void Initialize(const ::Uml::Diagram &dgr);

        extern Udm::UdmDiagram diagram;

        class ElectricDevice : public Udm::Object {
        public:
                ElectricDevice() {}
                ElectricDevice(Udm::ObjectImpl *impl) : UDM_OBJECT(impl) {}
                ElectricDevice(const ElectricDevice &master) : UDM_OBJECT(master) {}
                #ifdef UDM_RVALUE
                ElectricDevice(ElectricDevice &&master) : UDM_OBJECT(master) {}

                static ElectricDevice Cast(Udm::Object &&a) { return __Cast(std::move(a), meta); }
                ElectricDevice& operator=(ElectricDevice &&a) { Udm::Object::operator
=(std::move(a)); return *this; }
                #endif
                static ElectricDevice Cast(const Udm::Object &a) { return __Cast(a, meta); }
                static ElectricDevice Create(const Udm::Object &parent, const
::Uml::CompositionChildRole &role = Udm::NULLCHILDROLE) { return __Create(meta, parent, role); }
```

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

		Udm::InstantiatedAttr<ElectricDevice> Instances() { return 
Udm::InstantiatedAttr<ElectricDevice>(impl); }

		template <class Pred> Udm::InstantiatedAttr<ElectricDevice, Pred> 
Instances_sorted(const Pred &) { return Udm::InstantiatedAttr<ElectricDevice, Pred>(impl); }

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

		Udm::DerivedAttr<ElectricDevice> Derived() { return 
Udm::DerivedAttr<ElectricDevice>(impl); }

		template <class Pred> Udm::DerivedAttr<ElectricDevice, Pred> Derived_sorted(const 
Pred &) { return Udm::DerivedAttr<ElectricDevice, Pred>(impl); }

		Udm::ArchetypeAttr<ElectricDevice> Archetype() const { return 
Udm::ArchetypeAttr<ElectricDevice>(impl); }

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

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

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

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

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::ElectricTerminal, Pred> 
ElectricTerminal_children_sorted(const Pred &) const { return Udm::ChildrenAttr< 
::LampDiagram::ElectricTerminal, Pred>(impl, meta_ElectricTerminal_children); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::Wire, Pred> 
Wire_children_sorted(const Pred &) const { return Udm::ChildrenAttr< ::LampDiagram::Wire, Pred>(impl, 
meta_Wire_children); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::ElectricTerminal, Pred> 
ElectricTerminal_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr< 
::LampDiagram::ElectricTerminal, Pred>(impl, Udm::NULLCHILDROLE); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::Wire, Pred> 
Wire_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr< ::LampDiagram::Wire, 
Pred>(impl, Udm::NULLCHILDROLE); }

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

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



		static ::Uml::Class meta;

		static ::Uml::Attribute meta_MaxTempRating;

		static ::Uml::Attribute meta_MaxVoltageRating;

		static ::Uml::Attribute meta_position;

		static ::Uml::Attribute meta_name;

		static ::Uml::CompositionChildRole meta_ElectricTerminal_children;
		static ::Uml::CompositionChildRole meta_Wire_children;

		static ::Uml::CompositionParentRole meta_RootFolder_parent;


	};


	class Plug :  public ElectricDevice {

	public:

		Plug() {}

		Plug(Udm::ObjectImpl *impl) : ElectricDevice(impl) {}

		Plug(const Plug &master) : ElectricDevice(master) {}

		#ifdef UDM_RVALUE

		Plug(Plug &&master) : ElectricDevice(master) {}


		static Plug Cast(Udm::Object &&a) { return __Cast(std::move(a), meta); }

		Plug& operator=(Plug &&a) { Udm::Object::operator =(std::move(a)); return *this; }

		#endif

		static Plug Cast(const Udm::Object &a) { return __Cast(a, meta); }

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

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

		Udm::InstantiatedAttr<Plug> Instances() { return Udm::InstantiatedAttr<Plug>(impl); }

		template <class Pred> Udm::InstantiatedAttr<Plug, Pred> Instances_sorted(const Pred
&) { return Udm::InstantiatedAttr<Plug, Pred>(impl); }

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

		Udm::DerivedAttr<Plug> Derived() { return Udm::DerivedAttr<Plug>(impl); }

		template <class Pred> Udm::DerivedAttr<Plug, Pred> Derived_sorted(const Pred &) {
return Udm::DerivedAttr<Plug, Pred>(impl); }

		Udm::ArchetypeAttr<Plug> Archetype() const { return Udm::ArchetypeAttr<Plug>(impl); }

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

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

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

		void Accept(Visitor &v) { v.Visit_Plug(*this); }


		static ::Uml::Class meta;

		static ::Uml::Attribute meta_Format;

		static ::Uml::CompositionParentRole meta_Lamp_parent;


	};


	class Bulb :  public ElectricDevice {
        public:

                Bulb() {}

                Bulb(Udm::ObjectImpl *impl) : ElectricDevice(impl) {}

                Bulb(const Bulb &master) : ElectricDevice(master) {}

                #ifdef UDM_RVALUE

                Bulb(Bulb &&master) : ElectricDevice(master) {}


                static Bulb Cast(Udm::Object &&a) { return __Cast(std::move(a), meta); }

                Bulb& operator=(Bulb &&a) { Udm::Object::operator =(std::move(a)); return *this; }

                #endif

                static Bulb Cast(const Udm::Object &a) { return __Cast(a, meta); }

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

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

                Udm::InstantiatedAttr<Bulb> Instances() { return Udm::InstantiatedAttr<Bulb>(impl); }

                template <class Pred> Udm::InstantiatedAttr<Bulb, Pred> Instances_sorted(const Pred
&) { return Udm::InstantiatedAttr<Bulb, Pred>(impl); }

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

                Udm::DerivedAttr<Bulb> Derived() { return Udm::DerivedAttr<Bulb>(impl); }

                template <class Pred> Udm::DerivedAttr<Bulb, Pred> Derived_sorted(const Pred &) {
return Udm::DerivedAttr<Bulb, Pred>(impl); }

                Udm::ArchetypeAttr<Bulb> Archetype() const { return Udm::ArchetypeAttr<Bulb>(impl); }

                Udm::RealAttr Wattage() const { return Udm::RealAttr(impl, meta_Wattage); }

                Udm::RealAttr Voltage() const { return Udm::RealAttr(impl, meta_Voltage); }

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

                template <class Pred> Udm::AClassAssocAttr< ControlLink, Switch, Pred>
dst_sorted(const Pred &) const { return Udm::AClassAssocAttr< ControlLink, Switch, Pred>(impl,
meta_dst, meta_dst_rev); }

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

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

                void Accept(Visitor &v) { v.Visit_Bulb(*this); }


                static ::Uml::Class meta;

                static ::Uml::Attribute meta_Wattage;

                static ::Uml::Attribute meta_Voltage;

                static ::Uml::AssociationRole meta_dst;

                static ::Uml::AssociationRole meta_dst_rev;

                static ::Uml::CompositionParentRole meta_Lamp_parent;


        };
```

```
	class HalogenBulb :  public Bulb {

	public:

		HalogenBulb() {}

		HalogenBulb(Udm::ObjectImpl *impl) : Bulb(impl) {}

		HalogenBulb(const HalogenBulb &master) : Bulb(master) {}

		#ifdef UDM_RVALUE

		HalogenBulb(HalogenBulb &&master) : Bulb(master) {}


		static HalogenBulb Cast(Udm::Object &&a) { return __Cast(std::move(a), meta); }

		HalogenBulb& operator=(HalogenBulb &&a) { Udm::Object::operator =(std::move(a));
return *this; }

		#endif

		static HalogenBulb Cast(const Udm::Object &a) { return __Cast(a, meta); }

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

		HalogenBulb CreateInstance(const Udm::Object &parent, const
::Uml::CompositionChildRole &role = Udm::NULLCHILDROLE) { return __Create(meta, parent, role, impl); }

		Udm::InstantiatedAttr<HalogenBulb> Instances() { return
Udm::InstantiatedAttr<HalogenBulb>(impl); }

		template <class Pred> Udm::InstantiatedAttr<HalogenBulb, Pred> Instances_sorted(const
Pred &) { return Udm::InstantiatedAttr<HalogenBulb, Pred>(impl); }

		HalogenBulb CreateDerived(const Udm::Object &parent, const
::Uml::CompositionChildRole &role = Udm::NULLCHILDROLE) { return __Create(meta, parent, role, impl); }

		Udm::DerivedAttr<HalogenBulb> Derived() { return Udm::DerivedAttr<HalogenBulb>(impl);
}

		template <class Pred> Udm::DerivedAttr<HalogenBulb, Pred> Derived_sorted(const Pred
&) { return Udm::DerivedAttr<HalogenBulb, Pred>(impl); }

		Udm::ArchetypeAttr<HalogenBulb> Archetype() const { return
Udm::ArchetypeAttr<HalogenBulb>(impl); }

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

		void Accept(Visitor &v) { v.Visit_HalogenBulb(*this); }


		static ::Uml::Class meta;


	};


	class Switch :  public ElectricDevice {

	public:

		Switch() {}

		Switch(Udm::ObjectImpl *impl) : ElectricDevice(impl) {}

		Switch(const Switch &master) : ElectricDevice(master) {}

		#ifdef UDM_RVALUE

		Switch(Switch &&master) : ElectricDevice(master) {}
```

```
                 static Switch Cast(Udm::Object &&a) { return __Cast(std::move(a), meta); }

                 Switch& operator=(Switch &&a) { Udm::Object::operator =(std::move(a)); return *this;
}

                 #endif

                 static Switch Cast(const Udm::Object &a) { return __Cast(a, meta); }

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

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

                 Udm::InstantiatedAttr<Switch> Instances() { return
Udm::InstantiatedAttr<Switch>(impl); }

                 template <class Pred> Udm::InstantiatedAttr<Switch, Pred> Instances_sorted(const Pred
&) { return Udm::InstantiatedAttr<Switch, Pred>(impl); }

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

                 Udm::DerivedAttr<Switch> Derived() { return Udm::DerivedAttr<Switch>(impl); }

                 template <class Pred> Udm::DerivedAttr<Switch, Pred> Derived_sorted(const Pred &) {
return Udm::DerivedAttr<Switch, Pred>(impl); }

                 Udm::ArchetypeAttr<Switch> Archetype() const { return
Udm::ArchetypeAttr<Switch>(impl); }

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

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

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

                 template <class Pred> Udm::AClassAssocAttr< ControlLink, Bulb, Pred> src_sorted(const
Pred &) const { return Udm::AClassAssocAttr< ControlLink, Bulb, Pred>(impl, meta_src, meta_src_rev); }

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

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

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

                 void Accept(Visitor &v) { v.Visit_Switch(*this); }



                 static ::Uml::Class meta;

                 static ::Uml::Attribute meta_Safe;

                 static ::Uml::Attribute meta_Amps;

                 static ::Uml::AssociationRole meta_src;

                 static ::Uml::AssociationRole meta_src_rev;

                 static ::Uml::CompositionParentRole meta_MainSwitch_Lamp_parent;

                 static ::Uml::CompositionParentRole meta_FunctionSwitch_Lamp_parent;


        };


        class Lamp :  public ElectricDevice {
```

```
        public:

                Lamp() {}

                Lamp(Udm::ObjectImpl *impl) : ElectricDevice(impl) {}

                Lamp(const Lamp &master) : ElectricDevice(master) {}

                #ifdef UDM_RVALUE

                Lamp(Lamp &&master) : ElectricDevice(master) {}


                static Lamp Cast(Udm::Object &&a) { return __Cast(std::move(a), meta); }

                Lamp& operator=(Lamp &&a) { Udm::Object::operator =(std::move(a)); return *this; }

                #endif

                static Lamp Cast(const Udm::Object &a) { return __Cast(a, meta); }

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

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

                Udm::InstantiatedAttr<Lamp> Instances() { return Udm::InstantiatedAttr<Lamp>(impl); }

                template <class Pred> Udm::InstantiatedAttr<Lamp, Pred> Instances_sorted(const Pred
&) { return Udm::InstantiatedAttr<Lamp, Pred>(impl); }

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

                Udm::DerivedAttr<Lamp> Derived() { return Udm::DerivedAttr<Lamp>(impl); }

                template <class Pred> Udm::DerivedAttr<Lamp, Pred> Derived_sorted(const Pred &) {
return Udm::DerivedAttr<Lamp, Pred>(impl); }

                Udm::ArchetypeAttr<Lamp> Archetype() const { return Udm::ArchetypeAttr<Lamp>(impl); }

                Udm::StringAttrArr ArrayStr() const { return Udm::StringAttrArr(impl, meta_ArrayStr);
}

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

                Udm::IntegerAttrArr ArrayInt() const { return Udm::IntegerAttrArr(impl,
meta_ArrayInt); }

                Udm::BooleanAttrArr ArrayBool() const { return Udm::BooleanAttrArr(impl,
meta_ArrayBool); }

                Udm::RealAttrArr ArrayReal() const { return Udm::RealAttrArr(impl, meta_ArrayReal); }

                Udm::RealAttr sample() const { return Udm::RealAttr(impl, meta_sample); }

                Udm::TempStringAttrArr TempArrayStr() const { return Udm::TempStringAttrArr(impl,
meta_TempArrayStr); }

                Udm::TempIntegerAttrArr TempArrayInt() const { return Udm::TempIntegerAttrArr(impl,
meta_TempArrayInt); }

                Udm::TempRealAttrArr TempArrayReal() const { return Udm::TempRealAttrArr(impl,
meta_TempArrayReal); }

                Udm::TempBooleanAttrArr TempArrayBoolean() const { return
Udm::TempBooleanAttrArr(impl, meta_TempArrayBoolean); }

                Udm::StringAttr RegAttr() const { return Udm::StringAttr(impl, meta_RegAttr); }

                Udm::StringAttrArr annotations() const { return Udm::StringAttrArr(impl,
meta_annotations); }

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

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

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::ControlLink, Pred> 
ControlLink_children_sorted(const Pred &) const { return Udm::ChildrenAttr< 
::LampDiagram::ControlLink, Pred>(impl, meta_ControlLink_children); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::Bulb, Pred> 
Bulb_children_sorted(const Pred &) const { return Udm::ChildrenAttr< ::LampDiagram::Bulb, Pred>(impl, 
meta_Bulb_children); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::Switch, Pred> 
FunctionSwitch_sorted(const Pred &) const { return Udm::ChildrenAttr< ::LampDiagram::Switch, 
Pred>(impl, meta_FunctionSwitch); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::Plug, Pred> 
Plug_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr< ::LampDiagram::Plug, 
Pred>(impl, Udm::NULLCHILDROLE); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::ElectricDevice, Pred> 
ElectricDevice_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr< 
::LampDiagram::ElectricDevice, Pred>(impl, Udm::NULLCHILDROLE); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::Bulb, Pred> 
Bulb_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr< ::LampDiagram::Bulb, 
Pred>(impl, Udm::NULLCHILDROLE); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::HalogenBulb, Pred> 
HalogenBulb_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr< 
::LampDiagram::HalogenBulb, Pred>(impl, Udm::NULLCHILDROLE); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::Switch, Pred> 
Switch_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr< ::LampDiagram::Switch, 
Pred>(impl, Udm::NULLCHILDROLE); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::ControlLink, Pred> 
ControlLink_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr< 
::LampDiagram::ControlLink, Pred>(impl, Udm::NULLCHILDROLE); }

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

		void Accept(Visitor &v) { v.Visit_Lamp(*this); }



		static ::Uml::Class meta;

		static ::Uml::Attribute meta_ArrayStr;

		static ::Uml::Attribute meta_ModelName;

		static ::Uml::Attribute meta_ArrayInt;

		static ::Uml::Attribute meta_ArrayBool;
		static ::Uml::Attribute meta_ArrayReal;

		static ::Uml::Attribute meta_sample;

		static ::Uml::Attribute meta_TempArrayStr;

		static ::Uml::Attribute meta_TempArrayInt;

		static ::Uml::Attribute meta_TempArrayReal;

		static ::Uml::Attribute meta_TempArrayBoolean;

		static ::Uml::Attribute meta_RegAttr;

		static ::Uml::Attribute meta_annotations;

		static ::Uml::CompositionChildRole meta_MainSwitch;

		static ::Uml::CompositionChildRole meta_Plug_child;

		static ::Uml::CompositionChildRole meta_ControlLink_children;

		static ::Uml::CompositionChildRole meta_Bulb_children;

		static ::Uml::CompositionChildRole meta_FunctionSwitch;


	};


	class ElectricTerminal : public Udm::Object {

	public:

		ElectricTerminal() {}

		ElectricTerminal(Udm::ObjectImpl *impl) : UDM_OBJECT(impl) {}

		ElectricTerminal(const ElectricTerminal &master) : UDM_OBJECT(master) {}

		#ifdef UDM_RVALUE

		ElectricTerminal(ElectricTerminal &&master) : UDM_OBJECT(master) {}


		static ElectricTerminal Cast(Udm::Object &&a) { return __Cast(std::move(a), meta); }

		ElectricTerminal& operator=(ElectricTerminal &&a) { Udm::Object::operator
=(std::move(a)); return *this; }

		#endif

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

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

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

		Udm::InstantiatedAttr<ElectricTerminal> Instances() { return
Udm::InstantiatedAttr<ElectricTerminal>(impl); }

		template <class Pred> Udm::InstantiatedAttr<ElectricTerminal, Pred>
Instances_sorted(const Pred &) { return Udm::InstantiatedAttr<ElectricTerminal, Pred>(impl); }

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

		Udm::DerivedAttr<ElectricTerminal> Derived() { return
Udm::DerivedAttr<ElectricTerminal>(impl); }

		template <class Pred> Udm::DerivedAttr<ElectricTerminal, Pred> Derived_sorted(const
Pred &) { return Udm::DerivedAttr<ElectricTerminal, Pred>(impl); }
```

```
		Udm::ArchetypeAttr<ElectricTerminal> Archetype() const { return 
Udm::ArchetypeAttr<ElectricTerminal>(impl); }

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

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

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

		template <class Pred> Udm::AClassAssocAttr< Wire, ElectricTerminal, Pred> 
End2_sorted(const Pred &) const { return Udm::AClassAssocAttr< Wire, ElectricTerminal, Pred>(impl, 
meta_End2, meta_End2_rev); }

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

		template <class Pred> Udm::AClassAssocAttr< Wire, ElectricTerminal, Pred> 
End1_sorted(const Pred &) const { return Udm::AClassAssocAttr< Wire, ElectricTerminal, Pred>(impl, 
meta_End1, meta_End1_rev); }

		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); }

		void Accept(Visitor &v) { v.Visit_ElectricTerminal(*this); }



		static ::Uml::Class meta;

		static ::Uml::Attribute meta_position;

		static ::Uml::Attribute meta_Type;

		static ::Uml::AssociationRole meta_End2;

		static ::Uml::AssociationRole meta_End2_rev;

		static ::Uml::AssociationRole meta_End1;

		static ::Uml::AssociationRole meta_End1_rev;

		static ::Uml::CompositionParentRole meta_ElectricDevice_parent;



	};


	class Wire : public Udm::Object {

	public:

		Wire() {}

		Wire(Udm::ObjectImpl *impl) : UDM_OBJECT(impl) {}

		Wire(const Wire &master) : UDM_OBJECT(master) {}

		#ifdef UDM_RVALUE

		Wire(Wire &&master) : UDM_OBJECT(master) {}


		static Wire Cast(Udm::Object &&a) { return __Cast(std::move(a), meta); }

		Wire& operator=(Wire &&a) { Udm::Object::operator =(std::move(a)); return *this; }

		#endif

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

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

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

                Udm::InstantiatedAttr<Wire> Instances() { return Udm::InstantiatedAttr<Wire>(impl); }

                template <class Pred> Udm::InstantiatedAttr<Wire, Pred> Instances_sorted(const Pred
&) { return Udm::InstantiatedAttr<Wire, Pred>(impl); }

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

                Udm::DerivedAttr<Wire> Derived() { return Udm::DerivedAttr<Wire>(impl); }

                template <class Pred> Udm::DerivedAttr<Wire, Pred> Derived_sorted(const Pred &) {
return Udm::DerivedAttr<Wire, Pred>(impl); }

                Udm::ArchetypeAttr<Wire> Archetype() const { return Udm::ArchetypeAttr<Wire>(impl); }

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

                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); }

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

                Udm::AssocEndAttr< ::LampDiagram::ElectricTerminal> End2_end() const { return
Udm::AssocEndAttr< ::LampDiagram::ElectricTerminal>(impl, meta_End2_end_); }

                void Accept(Visitor &v) { v.Visit_Wire(*this); }



                static ::Uml::Class meta;

                static ::Uml::Attribute meta_Amps;

                static ::Uml::CompositionParentRole meta_ElectricDevice_parent;

                static ::Uml::AssociationRole meta_End1_end_;

                static ::Uml::AssociationRole meta_End2_end_;



        };



        class RootFolder : public Udm::Object {

        public:

                RootFolder() {}

                RootFolder(Udm::ObjectImpl *impl) : UDM_OBJECT(impl) {}

                RootFolder(const RootFolder &master) : UDM_OBJECT(master) {}

                #ifdef UDM_RVALUE

                RootFolder(RootFolder &&master) : UDM_OBJECT(master) {}



                static RootFolder Cast(Udm::Object &&a) { return __Cast(std::move(a), meta); }

                RootFolder& operator=(RootFolder &&a) { Udm::Object::operator =(std::move(a)); return
*this; }

                #endif

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

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

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

		Udm::InstantiatedAttr<RootFolder> Instances() { return
Udm::InstantiatedAttr<RootFolder>(impl); }

		template <class Pred> Udm::InstantiatedAttr<RootFolder, Pred> Instances_sorted(const
Pred &) { return Udm::InstantiatedAttr<RootFolder, Pred>(impl); }

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

		Udm::DerivedAttr<RootFolder> Derived() { return Udm::DerivedAttr<RootFolder>(impl); }

		template <class Pred> Udm::DerivedAttr<RootFolder, Pred> Derived_sorted(const Pred &)
{ return Udm::DerivedAttr<RootFolder, Pred>(impl); }

		Udm::ArchetypeAttr<RootFolder> Archetype() const { return
Udm::ArchetypeAttr<RootFolder>(impl); }

		Udm::StringAttr RegAttr() const { return Udm::StringAttr(impl, meta_RegAttr); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::ElectricDevice, Pred>
ElectricDevice_children_sorted(const Pred &) const { return Udm::ChildrenAttr<
::LampDiagram::ElectricDevice, Pred>(impl, meta_ElectricDevice_children); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::Plug, Pred>
Plug_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr< ::LampDiagram::Plug,
Pred>(impl, Udm::NULLCHILDROLE); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::ElectricDevice, Pred>
ElectricDevice_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr<
::LampDiagram::ElectricDevice, Pred>(impl, Udm::NULLCHILDROLE); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::Bulb, Pred>
Bulb_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr< ::LampDiagram::Bulb,
Pred>(impl, Udm::NULLCHILDROLE); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::HalogenBulb, Pred>
HalogenBulb_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr<
::LampDiagram::HalogenBulb, Pred>(impl, Udm::NULLCHILDROLE); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::Switch, Pred>
Switch_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr< ::LampDiagram::Switch,
Pred>(impl, Udm::NULLCHILDROLE); }

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

		template <class Pred> Udm::ChildrenAttr< ::LampDiagram::Lamp, Pred>
Lamp_kind_children_sorted(const Pred &) const { return Udm::ChildrenAttr< ::LampDiagram::Lamp,
Pred>(impl, Udm::NULLCHILDROLE); }

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

		void Accept(Visitor &v) { v.Visit_RootFolder(*this); }



		static ::Uml::Class meta;
		static ::Uml::Attribute meta_RegAttr;

		static ::Uml::CompositionChildRole meta_ElectricDevice_children;


	};


	class ControlLink : public Udm::Object {

	public:

		ControlLink() {}

		ControlLink(Udm::ObjectImpl *impl) : UDM_OBJECT(impl) {}

		ControlLink(const ControlLink &master) : UDM_OBJECT(master) {}

		#ifdef UDM_RVALUE

		ControlLink(ControlLink &&master) : UDM_OBJECT(master) {}


		static ControlLink Cast(Udm::Object &&a) { return __Cast(std::move(a), meta); }

		ControlLink& operator=(ControlLink &&a) { Udm::Object::operator =(std::move(a));
return *this; }

		#endif

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

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

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

		Udm::InstantiatedAttr<ControlLink> Instances() { return
Udm::InstantiatedAttr<ControlLink>(impl); }

		template <class Pred> Udm::InstantiatedAttr<ControlLink, Pred> Instances_sorted(const
Pred &) { return Udm::InstantiatedAttr<ControlLink, Pred>(impl); }

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

		Udm::DerivedAttr<ControlLink> Derived() { return Udm::DerivedAttr<ControlLink>(impl);
}

		template <class Pred> Udm::DerivedAttr<ControlLink, Pred> Derived_sorted(const Pred
&) { return Udm::DerivedAttr<ControlLink, Pred>(impl); }

		Udm::ArchetypeAttr<ControlLink> Archetype() const { return
Udm::ArchetypeAttr<ControlLink>(impl); }

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

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

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

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

		Udm::AssocEndAttr< ::LampDiagram::Switch> dst_end() const { return Udm::AssocEndAttr<
::LampDiagram::Switch>(impl, meta_dst_end_); }

		void Accept(Visitor &v) { v.Visit_ControlLink(*this); }


		static ::Uml::Class meta;

		static ::Uml::Attribute meta_name;
```

```
                static ::Uml::CompositionParentRole meta_Lamp_parent;
                static ::Uml::AssociationRole meta_src_end_;
                static ::Uml::AssociationRole meta_dst_end_;

        };

}

#endif // MOBIES_LAMPDIAGRAM_H
```

Appendix G. - Sample UDM-based program

```
// CreateLampModel.cpp : Create and test UDM data
//

#include "LampDiagram.h"


#ifdef _WIN32
UDM_USE_MGA
#endif

//UDM_USE_DOM


#include <Uml.h>
#include <UdmStatic.h>
#include <UmlExt.h>
#include <UdmBase.h>
#include <cint_string.h>
#include <UdmDom.h>

using namespace LampDiagram;

int main_dom_string (int argc, char * argv[])
{
        string xml_xsd =
"<?xml version=\"1.0\" encoding=\"UTF-8\"?>\
<?udm interface=\"LampDiagram\" version=\"1.00\"?>\
<xsd:schema xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\">\
<!-- generated on Thu Apr 01 12:28:35 2004 -->\
 <xsd:complexType name=\"BulbType\">\
  <xsd:sequence>\
   <xsd:element name=\"ElectricTerminal\" type=\"ElectricTerminalType\" minOccurs=\"0\"
maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"Wire\" type=\"WireType\" minOccurs=\"0\" maxOccurs=\"unbounded\"/>\
  </xsd:sequence>\
  <xsd:attribute name=\"_id\" type=\"xsd:ID\"/>\
  <xsd:attribute name=\"Wattage\" type=\"xsd:double\" use=\"required\"/>\
  <xsd:attribute name=\"Voltage\" type=\"xsd:double\"/>\
  <xsd:attribute name=\"MaxTempRating\" type=\"xsd:long\"/>\
  <xsd:attribute name=\"MaxVoltageRating\" type=\"xsd:double\"/>\
  <xsd:attribute name=\"position\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"name\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"dst\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_archetype\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"_derived\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_instances\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_desynched_atts\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"_real_archetype\" type=\"xsd:boolean\"/>\
  <xsd:attribute name=\"_subtype\" type=\"xsd:boolean\"/>\
 </xsd:complexType>\
 <xsd:complexType name=\"WireType\">\
  <xsd:attribute name=\"_id\" type=\"xsd:ID\"/>\
  <xsd:attribute name=\"Amps\" type=\"xsd:double\"/>\
  <xsd:attribute name=\"End2_end_\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"End1_end_\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"_archetype\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"_derived\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_instances\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_desynched_atts\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"_real_archetype\" type=\"xsd:boolean\"/>\
  <xsd:attribute name=\"_subtype\" type=\"xsd:boolean\"/>\
 </xsd:complexType>\
 <xsd:complexType name=\"PlugType\">\
  <xsd:sequence>\
   <xsd:element name=\"ElectricTerminal\" type=\"ElectricTerminalType\" minOccurs=\"0\"
maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"Wire\" type=\"WireType\" minOccurs=\"0\" maxOccurs=\"unbounded\"/>\
  </xsd:sequence>";

xml_xsd+= " <xsd:attribute name=\"_id\" type=\"xsd:ID\"/>\
  <xsd:attribute name=\"MaxTempRating\" type=\"xsd:long\"/>\
  <xsd:attribute name=\"MaxVoltageRating\" type=\"xsd:double\"/>\
  <xsd:attribute name=\"position\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"name\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"Format\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"_archetype\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"_derived\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_instances\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_desynched_atts\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"_real_archetype\" type=\"xsd:boolean\"/>\
  <xsd:attribute name=\"_subtype\" type=\"xsd:boolean\"/>\
 </xsd:complexType>\
 <xsd:complexType name=\"LampType\">\
  <xsd:sequence>\
   <xsd:element name=\"Bulb\" type=\"BulbType\" minOccurs=\"0\" maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"ControlLink\" type=\"ControlLinkType\" minOccurs=\"0\"
maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"ElectricTerminal\" type=\"ElectricTerminalType\" minOccurs=\"0\"
maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"HalogenBulb\" type=\"HalogenBulbType\" minOccurs=\"0\"
maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"Plug\" type=\"PlugType\" minOccurs=\"0\" maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"Switch\" type=\"SwitchType\" minOccurs=\"0\" maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"Wire\" type=\"WireType\" minOccurs=\"0\" maxOccurs=\"unbounded\"/>\
  </xsd:sequence>\
  <xsd:attribute name=\"_id\" type=\"xsd:ID\"/>\
  <xsd:attribute name=\"MaxTempRating\" type=\"xsd:long\"/>\
  <xsd:attribute name=\"MaxVoltageRating\" type=\"xsd:double\"/>\
  <xsd:attribute name=\"position\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"name\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"ArrayStr\" type=\"xsd:string\" default=\"second;first;\"/>\
  <xsd:attribute name=\"ModelName\" type=\"xsd:string\" default=\"Default Lamp Name\"/>\
  <xsd:attribute name=\"ArrayInt\" type=\"xsd:string\" default=\"5;4;3;2;\"/>\
  <xsd:attribute name=\"ArrayBool\" type=\"xsd:string\" default=\"false;true;false;true;\"/>\
  <xsd:attribute name=\"ArrayReal\" type=\"xsd:string\" default=\"9;8;7;6;\"/>\
  <xsd:attribute name=\"RegAttr\" type=\"xsd:string\"/>";

xml_xsd += "<xsd:attribute name=\"_archetype\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"_derived\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_instances\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_desynched_atts\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"_real_archetype\" type=\"xsd:boolean\"/>\
  <xsd:attribute name=\"_subtype\" type=\"xsd:boolean\"/>\
 </xsd:complexType>\
 <xsd:complexType name=\"HalogenBulbType\">\
  <xsd:sequence>\
   <xsd:element name=\"ElectricTerminal\" type=\"ElectricTerminalType\" minOccurs=\"0\"
maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"Wire\" type=\"WireType\" minOccurs=\"0\" maxOccurs=\"unbounded\"/>\
  </xsd:sequence>\
  <xsd:attribute name=\"_id\" type=\"xsd:ID\"/>\
  <xsd:attribute name=\"Wattage\" type=\"xsd:double\" use=\"required\"/>\
  <xsd:attribute name=\"Voltage\" type=\"xsd:double\"/>\
  <xsd:attribute name=\"MaxTempRating\" type=\"xsd:long\"/>\
  <xsd:attribute name=\"MaxVoltageRating\" type=\"xsd:double\"/>\
  <xsd:attribute name=\"position\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"name\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"dst\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_archetype\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"_derived\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_instances\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_desynched_atts\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"_real_archetype\" type=\"xsd:boolean\"/>\
  <xsd:attribute name=\"_subtype\" type=\"xsd:boolean\"/>\
 </xsd:complexType>\
 <xsd:complexType name=\"SwitchType\">\
  <xsd:sequence>\
   <xsd:element name=\"ElectricTerminal\" type=\"ElectricTerminalType\" minOccurs=\"0\"
maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"Wire\" type=\"WireType\" minOccurs=\"0\" maxOccurs=\"unbounded\"/>\
  </xsd:sequence>\
  <xsd:attribute name=\"_id\" type=\"xsd:ID\"/>\
  <xsd:attribute name=\"__child_as\">\
   <xsd:simpleType>\
    <xsd:restriction base=\"xsd:string\">\
     <xsd:enumeration value=\"FunctionSwitch\"/>\
     <xsd:enumeration value=\"MainSwitch\"/>\
     <xsd:enumeration value=\"ElectricDevice\"/>\
    </xsd:restriction>\
   </xsd:simpleType>\
  </xsd:attribute>";

  xml_xsd += "<xsd:attribute name=\"MaxTempRating\" type=\"xsd:long\"/>\
  <xsd:attribute name=\"MaxVoltageRating\" type=\"xsd:double\"/>\
  <xsd:attribute name=\"position\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"name\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"Safe\" type=\"xsd:boolean\"/>\
  <xsd:attribute name=\"Amps\" type=\"xsd:double\"/>\
  <xsd:attribute name=\"src\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_archetype\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"_derived\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_instances\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_desynched_atts\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"_real_archetype\" type=\"xsd:boolean\"/>\
  <xsd:attribute name=\"_subtype\" type=\"xsd:boolean\"/>\
 </xsd:complexType>\
 <xsd:complexType name=\"ControlLinkType\">\
  <xsd:attribute name=\"_id\" type=\"xsd:ID\"/>\
  <xsd:attribute name=\"name\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"dst_end_\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"src_end_\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"_archetype\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"_derived\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_instances\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_desynched_atts\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"_real_archetype\" type=\"xsd:boolean\"/>\
  <xsd:attribute name=\"_subtype\" type=\"xsd:boolean\"/>\
 </xsd:complexType>\
 <xsd:complexType name=\"RootFolderType\">\
  <xsd:sequence>\
   <xsd:element name=\"Bulb\" type=\"BulbType\" minOccurs=\"0\" maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"HalogenBulb\" type=\"HalogenBulbType\" minOccurs=\"0\"
maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"Lamp\" type=\"LampType\" minOccurs=\"0\" maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"Plug\" type=\"PlugType\" minOccurs=\"0\" maxOccurs=\"unbounded\"/>\
   <xsd:element name=\"Switch\" type=\"SwitchType\" minOccurs=\"0\" maxOccurs=\"unbounded\"/>\
  </xsd:sequence>\
  <xsd:attribute name=\"_id\" type=\"xsd:ID\"/>\
  <xsd:attribute name=\"RegAttr\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"_archetype\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"_derived\" type=\"xsd:IDREFS\"/>";

  xml_xsd += " <xsd:attribute name=\"_instances\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_desynched_atts\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"_real_archetype\" type=\"xsd:boolean\"/>\
  <xsd:attribute name=\"_subtype\" type=\"xsd:boolean\"/>\
 </xsd:complexType>\
 <xsd:complexType name=\"ElectricTerminalType\">\
  <xsd:attribute name=\"_id\" type=\"xsd:ID\"/>\
  <xsd:attribute name=\"position\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"Type\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"End2\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"End1\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_archetype\" type=\"xsd:IDREF\"/>\
  <xsd:attribute name=\"_derived\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_instances\" type=\"xsd:IDREFS\"/>\
  <xsd:attribute name=\"_desynched_atts\" type=\"xsd:string\"/>\
  <xsd:attribute name=\"_real_archetype\" type=\"xsd:boolean\"/>\
  <xsd:attribute name=\"_subtype\" type=\"xsd:boolean\"/>\
 </xsd:complexType>\
 <xsd:element name=\"RootFolder\" type=\"RootFolderType\"/>\
</xsd:schema>";



  string xml_str = "<RootFolder xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
xsi:noNamespaceSchemaLocation=\"LampDiagram.xsd\"><Lamp name=\"Host Lamp \" ArrayInt=\"2;3;4;5\"
ArrayStr=\"first;second;\" ArrayBool=\"false;false;true;true\"
ArrayReal=\"6.000000000000000;7.000000000000000;8.000000000000000;9.000000000000000\"
ModelName=\"Sequence tester lamp\"><Bulb name=\"Bulb 1\" Voltage=\"10.000000\"
Wattage=\"100.000000\"/><Bulb name=\"Bulb 2\" Voltage=\"20.000000\" Wattage=\"200.000000\"/><Bulb
name=\"Bulb 3\" Voltage=\"30.000000\" Wattage=\"300.000000\"/></Lamp></RootFolder>";


        UdmDom::str_xsd_storage::StoreXsd("LampDiagram.xsd", xml_xsd);

        try
        {


                UdmDom::DomDataNetwork ddn(LampDiagram::diagram);
                ddn.OpenExistingFromString(xml_str,"LampDiagram.xsd", Udm::CHANGES_PERSIST_ALWAYS);
                RootFolder rf = RootFolder::Cast(ddn.GetRootObject());
                Lamp l = *( (set<Lamp>(rf.Lamp_kind_children())).begin());
                l.name() = " name changed!";
                ddn.CloseWithUpdate();

                const string& outstr = ddn.Str();
                cout << outstr << endl;


                ddn.CreateNewToString("LampDiagram.xsd", RootFolder::meta, 
Udm::CHANGES_PERSIST_ALWAYS);
                RootFolder rf1 = RootFolder::Cast(ddn.GetRootObject());
                Lamp l1 = Lamp::Create(rf1);
                l1.name() = "cool!";
                ddn.CloseWithUpdate();

                const string & outstr1 = ddn.Str();
                cout << outstr1 << endl;



        }
        catch (udm_exception &e)
        {
                cout << e.what() << endl;
        };

        return 1;
};

int main1(int argc, char * argv[])
{

        /*
        *
        *       UDM test code, which demonstrates how to preserve the order of children
        *       in the backend. It works only with MEM and DOM backends.
        *
        *
        */
        if(argc < 2)
        {
                        cout << "Usage: CreateLampModel <UdmBackEnd file(lamp to be created in)>\n";
                        return -1;
        }
        try
        {

                Udm::SmartDataNetwork nw(diagram);

                nw.CreateNew(argv[1],"LampDiagram", RootFolder::meta, Udm::CHANGES_PERSIST_ALWAYS);

                {
                        RootFolder rrr = RootFolder::Cast(nw.GetRootObject());
                        Lamp lamp = Lamp::Create(rrr);
                        lamp.name() = "Host Lamp ";
                        lamp.ModelName() = "Sequence tester lamp";


                        Bulb bulb1  = Bulb::Create(lamp);
                        bulb1.name() = "Bulb 1";
                        bulb1.Wattage() = 100;
                        bulb1.Voltage() = 10;


                        Bulb bulb2  = Bulb::Create(lamp);
                        bulb2.name() = "Bulb 2";
                        bulb2.Wattage() = 200;
                        bulb2.Voltage() = 20;


                        Bulb bulb3  = Bulb::Create(lamp);
                        bulb3.name() = "Bulb 3";
                        bulb3.Wattage() = 300;
                        bulb3.Voltage() = 30;


                }
                nw.CloseWithUpdate();
                nw.OpenExisting(argv[1], "LampDiagram", Udm::CHANGES_PERSIST_ALWAYS);
                {
                        RootFolder rrr = RootFolder::Cast(nw.GetRootObject());
                        set<Lamp> lamps = rrr.Lamp_kind_children();
```

```
Lamp lamp = *(lamps.begin());

vector<Bulb> bulbs = lamp.Bulb_children();
vector<Bulb> new_vector;

vector<Bulb>::iterator bulb_i = bulbs.begin();
cout << " The order of bulbs should be 1, 2, 3" << endl;
cout << " The order of bulbs: " ;
while (bulb_i != bulbs.end())
{
        Bulb bulb = *bulb_i;
        cout << (string)bulb.name() << " ,";
        bulb_i++;
}


//inverting order
vector<Bulb>::reverse_iterator bulb_ri = bulbs.rbegin();
while (bulb_ri != bulbs.rend())
{


        new_vector.push_back(*bulb_ri);
        ++bulb_ri;


}

lamp.Bulb_children() = new_vector;

bulbs = lamp.Bulb_children();
bulb_i = bulbs.begin();
cout << endl << " The order of bulbs should be 3, 2, 1" << endl;
cout << " The order of bulbs: " ;
while (bulb_i != bulbs.end())
{
        Bulb bulb = *bulb_i;
        cout << (string)bulb.name() << " ,";
        bulb_i++;
}


//taking out from the middle - that's index 1
Bulb middle_b = bulbs[1];
bulbs = lamp.Bulb_children() -= middle_b;
bulb_i = bulbs.begin();
cout << endl << " Bulb 2 removed. The order of bulbs should be 3,1" << endl;
cout << " The order of bulbs: " ;
while (bulb_i != bulbs.end())
{
        Bulb bulb = *bulb_i;
        cout << (string)bulb.name() << " ,";
        bulb_i++;
}


//adding back to the  end
middle_b = Bulb::Create(lamp);
middle_b.name() = "Bulb 2 recreated";
middle_b.Wattage() =2000;
middle_b.Voltage() = 200;


bulbs = lamp.Bulb_children();


bulb_i = bulbs.begin();

cout << endl << " Bulb 2 readded to the end. The order of bulbs should be
3,1,2" << endl;
cout << " The order of bulbs: " ;
while (bulb_i != bulbs.end())
{
        Bulb bulb = *bulb_i;
        cout << (string)bulb.name() << " ,";
        bulb_i++;
}


cout << endl;
```

```
                }



        }
        catch (udm_exception e)
        {
                cout << "exception : " << e.what() << endl;

        }
        return 0;


}



int main_arch_der_sub_test(int argc, char * argv[])
{

        /*
        *
        *       UDM test code, which demonstrates how to use archetype/derived/subtypes
        *       with UDM, specially regarding attributes
        *
        *
        */
        if(argc < 2)
        {
                        cout << "Usage: CreateLampModel <UdmBackEnd file(lamp to be created in)>\n";
                        return -1;
        }
        try
        {

                Udm::SmartDataNetwork nw(diagram);

                nw.CreateNew(argv[1],"LampDiagram", RootFolder::meta, Udm::CHANGES_PERSIST_ALWAYS);

                {
                        RootFolder rrr = RootFolder::Cast(nw.GetRootObject());
                        Lamp lamp = Lamp::Create(rrr);
                        lamp.name() = "Host Lamp ";
                        lamp.ModelName() = "Sequence tester lamp";


                        Bulb bulb1  = Bulb::Create(lamp);
                        bulb1.name() = "Bulb 1";
                        bulb1.Wattage() = 100;
                        bulb1.Voltage() = 10;


                        Bulb Bulb1_instance_1 = bulb1.CreateInstance(lamp);

                        //Bulb1_instance_1.name() = "Bulb Instance 1";

                        Bulb Bulb1_instance_2 = bulb1.CreateInstance(lamp);
                        Bulb1_instance_2.name() = "Bulb Instance 2";

                        Bulb Bulb1_derived_1 = bulb1.CreateDerived(lamp);
                        //Bulb1_derived_1.name() = "Bulb derived 1";

                        Bulb Bulb1_derived_2 = bulb1.CreateDerived(lamp);

                        Bulb1_derived_2.name() = "Bulb derived 2";


                        set<Udm::Object> ders = bulb1.derived();
                        set<Udm::Object>::iterator d_i = ders.begin();
                        while (d_i != ders.end())
                        {
                                Bulb derived_bulb = Bulb::Cast(*d_i++);
                                cout << "Derived bulb name: " << (string)derived_bulb.name() <<
endl;

                                Bulb archetype = Bulb::Cast(derived_bulb.archetype());
```

```
				cout << "Archetype bulb name: " << (string)archetype.name() << endl;

			}

			set<Udm::Object> insts = bulb1.instances();
			set<Udm::Object>::iterator i_i = insts.begin();
			while (i_i != insts.end())
			{
				Bulb instance_bulb = Bulb::Cast(*i_i++);
				cout << "Instance bulb name: " << (string)instance_bulb.name() <<
endl;

				Bulb archetype = Bulb::Cast(instance_bulb.archetype());
				cout << "Archetype bulb name: " << (string)archetype.name() << endl;

			}


			//bulb kind childrens
			set<Bulb> bulbs = lamp.Bulb_kind_children();
			int i = bulbs.size();


			//Bulb1_instance_1.parent() = NULL;
			//Bulb1_derived_2.parent() = NULL;
			//bulb1.parent() = NULL;

			//bulb kind childrens
			bulbs = lamp.Bulb_kind_children();
			i = bulbs.size();

			bulb1.name() = "New Archetype name for Bulb1!";


			//Plug is required...

			Plug::Create(lamp);



		}
		nw.CloseWithUpdate();

		nw.OpenExisting(argv[1],"LampDiagram", Udm::CHANGES_PERSIST_ALWAYS);
		{
			RootFolder rf = RootFolder::Cast(nw.GetRootObject());
			vector<Lamp> rf_lamps = rf.Lamp_kind_children();
			vector<Bulb> lamp_bulbs = (*rf_lamps.begin()).Bulb_kind_children();
			vector<Bulb>::iterator lamp_bulbs_i = lamp_bulbs.begin();

			while (lamp_bulbs_i != lamp_bulbs.end())
			{
				Bulb this_bulb = *lamp_bulbs_i++;
				cout << " Bulb Name:  " << (string) this_bulb.name() << endl;
				cout << " -------------deriveds ----------- " << endl;

				set<Udm::Object> ders = this_bulb.derived();
				set<Udm::Object>::iterator d_i = ders.begin();
				while (d_i != ders.end())
				{
					Bulb derived_bulb = Bulb::Cast(*d_i++);
					cout << "\tDerived bulb name: " <<
(string)derived_bulb.name() << endl;

					Bulb archetype = Bulb::Cast(derived_bulb.archetype());
					cout << "\tArchetype bulb name: " <<
(string)archetype.name() << endl;

				}
				cout << " -------------instances ----------- " << endl;

				set<Udm::Object> insts = this_bulb.instances();
				set<Udm::Object>::iterator i_i = insts.begin();
				while (i_i != insts.end())
				{
					Bulb instance_bulb = Bulb::Cast(*i_i++);
					cout << "\tInstance bulb name: " <<
(string)instance_bulb.name() << endl;

					Bulb archetype = Bulb::Cast(instance_bulb.archetype());
```

```
                                        cout << "\tArchetype bulb name: " <<
(string)archetype.name() << endl;

                                }

                                cout << " -------------archetype ----------- " << endl;
                                if (this_bulb.archetype())
                                {
                                        Bulb archetype = Bulb::Cast(this_bulb.archetype());
                                        cout << "Archetype bulb name: " << (string)archetype.name()
<< endl;
                                }
                        }
                }


        }
        catch (udm_exception e)
        {
                cout << "exception : " << e.what() << endl;

        }
        return 0;


}

int main(int argc, char* argv[])
{

        /*
        *  This code creates a lamp, and then copies it to another Udm DataNetwork,
        *  using the Udm::DataNetwork assignment operator(=).
        *
        *  Two arguments are expected, two filenames. The first will be the datanetwork where the lamp
        *  is created, the second is a copy of the first. The filenames can have any extension of
.MEM, .MGA,
        *  .DOM. In case of MGA, the Lamp paradigm must be registered first.
        *
        *       The example shows the basic UDM operations:
        *               - creating & removing objects, associations, manipulating children
        *               - association class based associations
        *               - setting & getting simple attributes, simple non-persistent attributes
        *               - setting & getting array attributes, array non-persistent attributes,
accessing array attribute items by index
        *               - copying a datanetwork as a whole
        *               - different ways of walking the object tree
        *               - creating instances and subtypes
        *
        */
        if(argc < 3) {
                        cout << "Usage: CreateLampModel <UdmBackEnd file(lamp to be created in)>
<UdmBackEnd file(copy)>\n";
                        return -1;
        }
        try {


                Udm::SmartDataNetwork nw(diagram);

                nw.CreateNew(argv[1],"LampDiagram", RootFolder::meta, Udm::CHANGES_PERSIST_ALWAYS);

                {
                        RootFolder rrr = RootFolder::Cast(nw.GetRootObject());
                        int count = 0;
                        //for (count = 0; count < 100; count++)
                        //{
                                Lamp doubleBulbLamp = Lamp::Create(rrr);


                                vector<__int64> a;
                                a.push_back(0x1234567890abcdefLL);//64bit integer
                                a.push_back(7);
                                a.push_back(1);

                                doubleBulbLamp.ArrayInt() = a;


                                cout << " Array indexer test: integer..." << endl;
                                cout << doubleBulbLamp.ArrayInt()[0] << endl;
```

```
                          //doubleBulbLamp.ArrayInt()[4] = 3;
                          //doubleBulbLamp.ArrayInt()[4] += 2;

                          cout << (long) doubleBulbLamp.ArrayInt()[4] << endl;


                          //testing non-persistent array attributes
                          doubleBulbLamp.TempArrayInt() = a;                  //assign
                          a = doubleBulbLamp.TempArrayInt();                  //retrieve
                          //display
                          cout << "Testing non-persistent array attribute(int) :" << endl;
                          vector<__int64>::const_iterator a_i = a.begin();
                          while (a_i != a.end()) cout << *a_i++ << ",";
                          cout << endl;

                          cout << " Array indexer test: non-persistent integer..." << endl;
                          cout << doubleBulbLamp.TempArrayInt()[0] << endl;
                          //doubleBulbLamp.TempArrayInt()[4] = 3;
                          //doubleBulbLamp.TempArrayInt()[4] += 2;

                          cout << doubleBulbLamp.TempArrayInt()[4] << endl;


                          vector<double> b;
                          b.push_back(1.1);
                          b.push_back(2.4);
                          b.push_back(3.141592653589793234);
                          doubleBulbLamp.ArrayReal() = b;

                          //testing non-persistent array attributes
                          doubleBulbLamp.TempArrayReal() = b;                         //assign
                          b = doubleBulbLamp.TempArrayReal();
//retrieve
                          //display
                          cout << "Testing non-persistent array attribute(double) :" << endl;
                          vector<double>::const_iterator b_i =b.begin();
                          while (b_i != b.end()) cout << *b_i++ << ",";
                          cout << endl;


                          vector<bool> c;
                          c.push_back(true);
                          c.push_back(true);
                          c.push_back(false);
                          doubleBulbLamp.ArrayBool() = c;

                          //testing non-persistent array attributes
                          doubleBulbLamp.TempArrayBoolean() = c;                      //assign
                          c = doubleBulbLamp.TempArrayBoolean();
//retrieve
                          //display
                          cout << "Testing non-persistent array attribute(boolean) :" << endl;
                          vector<bool>::const_iterator c_i = c.begin();
                          while (c_i != c.end()) *c_i++ ? cout << "true," : cout <<"false,";
                          cout << endl;

                          vector<string> d;
                          d.push_back("apple");
                          d.push_back("orange");
                          d.push_back("hello");
                          d.push_back("semi;colon");
                          d.push_back("doublesemi;;colon");
                          d.push_back("triplesemi;;;colon");
                          d.push_back("back\\slash");
                          d.push_back("double back\\\\slash");
                          d.push_back("triple back\\\\\\slash");
                          d.push_back("triple semi-coloback\\;\\;\\;slash-1");
                          d.push_back("triple semi-coloback\\\\\\;;;slash-2");
                          doubleBulbLamp.ArrayStr() = d;


                          cout << " Array indexer test: string..." << endl;
                          cout << (string) doubleBulbLamp.ArrayStr()[4] << endl;
                          //doubleBulbLamp.ArrayStr()[4] = " array indexer assignment";
                          //doubleBulbLamp.ArrayStr()[4] += " += operator";

                          cout << (string) doubleBulbLamp.ArrayStr()[4] << endl;
```

```
				//testing non-persistent array attributes
				doubleBulbLamp.TempArrayStr() = d;			//assign
				d = doubleBulbLamp.TempArrayStr();			//retrieve
				//display
				cout << "Testing non-persistent array attribute(string) :" << endl;
				vector<string>::const_iterator d_i = d.begin();
				while (d_i != d.end()) cout << *d_i++  << ",";
				cout << endl;


				//annotations
				vector<string> annotations;
				annotations.push_back("Anno1=\\=my annotation on this Lamp\\=");
				annotations.push_back("Anno1/aspects=600,300");
				annotations.push_back("Anno1/aspects/*=");
				annotations.push_back("Anno1/hidden=0");
				annotations.push_back("Anno1/inheritable=0");
				doubleBulbLamp.annotations() = annotations;			//assign
				annotations = doubleBulbLamp.annotations();
	//retrieve
				cout << "Annotations on doubleBulbLamp:" << endl;
				vector<string>::const_iterator annotations_i = annotations.begin();
				while (annotations_i != annotations.end())
					cout << "\t" << *annotations_i++ << endl;
				cout << endl;



				char lamp_name[70];
				sprintf(lamp_name, "HighLight XL150 instance no. %d, here follows
end-of line:\n", count);
				doubleBulbLamp.name() = lamp_name;

				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() = "SW 2";
				ElectricTerminal s2t1 = ElectricTerminal::Create(switch2);
				s2t1.position() = "(20,100)";
				ElectricTerminal s2t2 = ElectricTerminal::Create(switch2);
				s2t2.position() = "(220,100)";
```

```
                                Switch switch3 = Switch::Create(doubleBulbLamp,
Lamp::meta_FunctionSwitch);
                                switch3.position() = "(300,250)";
                                switch3.name() = "Halogen switch";
                                ElectricTerminal s3t1 = ElectricTerminal::Create(switch3);
                                s3t1.position() = "(20,100)";
                                ElectricTerminal s3t2 = ElectricTerminal::Create(switch3);
                                s3t2.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);
                                b2t1.position() = "(220,100)";
                                ElectricTerminal b2t2 = ElectricTerminal::Create(bulb2);
                                b2t2.position() = "(220,200)";

        //test inherited parentroles & childroles

                                HalogenBulb bulb3 = HalogenBulb::Create(doubleBulbLamp,
Lamp::meta_Bulb_children);
                                bulb3.name() = "HalogenBulb3";
                                bulb3.position() = "(100, 150)";
                                ElectricTerminal b3t1 = ElectricTerminal::Create(bulb3,
HalogenBulb::meta_ElectricTerminal_children);
                                b3t1.position() = "(320,100)";
                                ElectricTerminal b3t2 = ElectricTerminal::Create(bulb3);
                                b3t2.position() = "(320,200)";






                // 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);
                                Wire w8 = Wire::Create(doubleBulbLamp);
                                Wire w9 = Wire::Create(doubleBulbLamp);
                                Wire w10 = Wire::Create(doubleBulbLamp);






                //connect them

                                //live onnections
                                //#1: connect plug 1 to main switch input
                                //this is one way to connect it!

                                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;

                          w10.End1_end() = mst2;
                          w10.End2_end() = s3t1;


                          //#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;

                          //#8: connect sw3 output to halogenbulb
                          w8.End1_end() = s3t2;
                          w8.End2_end() = b3t1;


                          //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;

                          w9.End1_end() = plt2;
                          w9.End2_end() = b3t2;




        // 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;
                          w8.Amps() = 14.0;
                          w9.Amps() = 16.0;
                          w10.Amps() = 17.0;



                          bulb1.Wattage() = 55;
                          bulb2.Wattage() = 155;
                          bulb3.Wattage() = 200;

                          //this should got  a default value
                          //doubleBulbLamp.ModelName() = "SuperDoubleLight 155";

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


                          //creating control links

                          ControlLink cl1 = ControlLink::Create(doubleBulbLamp);
                          cl1.name() = "Bulb 1 switcher";

                          //one way to set associations:
                          cl1.src_end() = bulb1;
                          cl1.dst_end() = switch1;

        /*

                          //the other way to set associations would be
                          set<ControlLink> s_cl;
                          s_cl.insert(cl1);
                          bulb1.dst() = s_cl;
                          switch1.src() = s_cl;
        */
```

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

                                ControlLink cl3 = ControlLink::Create(doubleBulbLamp);
                                cl3.src_end() = bulb3;
                                cl3.dst_end() = switch3;
                                cl3.name() = "Halogen Bulb  switcher";



                                set<ControlLink> lks = doubleBulbLamp.ControlLink_kind_children();
                                set<ControlLink>::iterator l1;

                                for(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;
                                }


                                set<Bulb> s_bulb = doubleBulbLamp.Bulb_kind_children();
                                for(set<Bulb>::iterator s_bulb_i = s_bulb.begin(); s_bulb_i !=
s_bulb.end(); s_bulb_i++)
                                {
                                        cout << " Name of the bulbs: " << string(s_bulb_i->name())
<< endl;
                                        set<ControlLink> s_cls = s_bulb_i->dst();
                                        cout << " number of associated controllinks: " << (long)
(s_cls.size()) << endl;
                                        ControlLink cll = *s_cls.begin();
                                        cout << " associated control link: " << (string) cll.name()
<< endl;

                                        //this is safe, because the vector will contain only one
pointer,
                                        //which is already clone()-ed,
                                        //but will be released as soon as peer_switch goes out of
context
                                        vector<Udm::ObjectImpl*> sw_ois = s_bulb_i->__impl()-
>getAssociation(LampDiagram::Bulb::meta_dst, Udm::TARGETFROMPEER);
                                        cout << " number of peers(meta_dst) :" << (long)
sw_ois.size() << endl;
                                        Switch peer_switch(*sw_ois.begin());
                                        cout << " name of peer: " << (string)peer_switch.name() <<
endl;

                                }

                        //testing object delete
                        //this is  way to delete cl1
                        //cl1.parent() = NULL;



                        //testing new set of childs
                        //this is a way to delete cl2

                        //set<ControlLink> cl_test_set;
                        //cl_test_set.insert(cl1);
                        //doubleBulbLamp.ControlLink_kind_children() = cl_test_set;


                        //testing now associations

                        Bulb b11 = cl1.src_end();
                        Bulb b12 = cl2.src_end();

                        cl1.src_end() = b12;
                        cl2.src_end() = b11;

                        lks = doubleBulbLamp.ControlLink_kind_children();
                        for(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;
                        }


                        //testing parentroles


                        Lamp lamp1 = switch1.MainSwitch_Lamp_parent();
                        if (lamp1)
                        {
                                cout << "lamp1 is not NULL" << endl;
                                long id = lamp1.uniqueId();
                                cout << id << endl;
                        }

                        Lamp lamp2 = switch1.FunctionSwitch_Lamp_parent();
                        if (lamp2)
                        {
                                cout << "lamp2 is not NULL" << endl;
                                long id = lamp2.uniqueId();
                                cout << id << endl;
                        }


                        RootFolder r1 = switch1.RootFolder_parent();
                        if (r1)
                        {
                                cout << "r1 is not NULL" << endl;
                                long id = r1.uniqueId();
                                cout << id << endl;
                        }

                        Udm::Object o1 = switch1.parent();
                        if (o1)
                        {
                                cout << "o1 is not NULL" << endl;
                                long id = o1.uniqueId();
                                cout << id << endl;

                        }

                        //testing child roles

                        set<ElectricDevice> ed_s = doubleBulbLamp.ElectricDevice_kind_children();

                        cout << ed_s.size() << " electric devices found in lamp ..." << endl;


                        //create a switch directly in the rootfolder (testing the DTD features)
                        //Switch rr_sw = Switch::Create(rrr);
                        //rr_sw.name() = "rootfolder switch";

                        //create derived object



                        //testing assignment

                        Udm::SmartDataNetwork snw(diagram);
                        snw.CreateNew(argv[2],"LampDiagram",RootFolder::meta,
Udm::CHANGES_PERSIST_ALWAYS);
                        snw = nw;                                                 //Datanetwork
copy operator
                        snw = nw;                                                 //Datanetwork
copy operator - again - should not duplicate

                        snw.CloseWithUpdate();



                        Udm::Object oo = nw.ObjectById(bulb1.uniqueId());
                        Bulb uu = Bulb::Cast(oo);

                        cout << long(uu.Wattage()) << endl;
                        cout << long(oo.uniqueId()) << endl;
```

```
//                      }
                        nw.CloseWithUpdate();

                        }


                }

                catch(const udm_exception &e)
                {
                        cout << "Exception: " << e.what() << endl;
                        return(11);

                }

                try {                                   // reload the data to see if it is valid



                        Udm::SmartDataNetwork nw(diagram);
                        nw.OpenExisting(argv[1],"", Udm::CHANGES_LOST_DEFAULT);
                        {


                                RootFolder rr = RootFolder::Cast(nw.GetRootObject());
                                set<LampDiagram::Lamp> l_s = rr.Lamp_kind_children();
                                cout << " number of lamps: " << l_s.size() << endl;
                                Lamp l = *l_s.begin();
                                cout << "lamp: " << (string) l.name() ;

                                cout << "ArrayInt: ";
                                vector<__int64> vl = l.ArrayInt();
                                vector<__int64>::const_iterator vl_i = vl.begin();

                                while (vl_i != vl.end()) cout << *vl_i++ << ",";

                                cout << endl;

                                cout << "ArrayFloat: ";
                                vector<double> vd = l.ArrayReal();
                                vector<double>::const_iterator vd_i = vd.begin();
                                while (vd_i != vd.end())
                                        cout << *vd_i++ << ",";
                                cout << endl;

                                cout << "ArrayBool: ";
                                vector<bool> vb = l.ArrayBool();
                                vector<bool>::const_iterator vb_i = vb.begin();
                                while (vb_i != vb.end())
                                        cout << (*vb_i++ ? "true" : "false") << ", ";
                                cout << endl;

                                cout << "ArrayStr: ";
                                vector<string> vs = l.ArrayStr();
                                vector<string>::const_iterator vs_i = vs.begin();
                                while (vs_i != vs.end())
                                        cout << *vs_i++ << ",";
                                cout << endl;





                                nw.CloseNoUpdate();
                        }

                        cout << "Basically, all is OK" << endl ;
                }

                catch(const udm_exception &e)   {
                        cout << "Exception: " << e.what()  << endl;
                        return(12);

                }

                cint_string a("hello");
                cint_string b("hell0");
                cint_string c("hello");
```

```
        if (a == b) cout << " a equals to b " << endl;
        if (a == c) cout << " a equals to c " << endl;

        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.