Software through Pictures®

Millennium Edition

Release 8

Modelling and generating COM components with StP/UML and ACD

StP.README-10147/001

White Paper
March 2000

Oliver Maus Aonix GmbH

Software through Pictures

Modelling and generating COM components with StP/UML and ACD
Millennium Edition (NT and Windows 2000)
April 2001

Executive Summary

Code generation from application development tools promised the earth and actually delivered just a few stones. With a new approach and proven technology, it is now possible to leverage modelling tools to get meaningful amounts of high quality code automatically generated.

This white paper describes the application of this technology in the modelling of COM components with Software through Pictures (StP) for the UML and the generation of code using the Architecture Centric Development technology (ACD).

The modelling of components is thus taken beyond the traditional role of building models for documentation purposes, and enriching these, in accordance with UML to achieve compact, re-usable models which can be used for application generation.

More Compact Models, and more Generated Code - How?

Demonstrably larger amounts of much more meaningful source code can be generated from UML models by capturing the architectural and technical aspects of an application in a set of templates, which are automatically applied when generating code. Putting this detail into templates rather than UML results in more compact, easier to understand and maintain models, that more accurately reflect the business and application requirements.

This approach helps to achieve a level of re-use of these templates and models on subsequent projects. Ultimately this can mean increased quality, better usage of people resources, and a greater likelihood of meeting project cost and schedules.

The starting point is to get a clear understanding of what needs to be generated - using the target language and the way in which it will be used. The next step is to make clear what elements can be modelled with UML, and how these are to be represented in the models. The third and last step is to write templates that generate the source code from the model elements using these strategies.

Mapping UML to COM

COM (Component Object Model) is a binary standard for remote object invocation between clients and servers. It is based on interfaces that are separated from the implementations, and its components are binary, reusable components that are used via standardised mechanisms.

The interfaces mentioned above are described in an “Interface Description Language” (IDL) and stored in one or several files with an extension of .idl.

The Interface Description Language

This is an example of an IDL file describing the interface of a simple timeserver:

```
[  object,
   uuid(1a658790-982e-11d3-b844-00008606a51e),
   helpstring("ITime Interface"),
   pointer_default(unique)
]
interface ITime : IUnknown
{
        import "unknown.idl";
        HRESULT setSwitchTime( [in] int hour, [in] int
minute);
        HRESULT getSwitchTime( [out] int* hour, [out] int*
minute);

};

[  object,
   uuid(1a6ba450-982e-11d3-b844-00008606a51e),
   helpstring("ITimeString Interface"),
   pointer_default(unique)
]
interface ITimeFormat : IUnknown
{
        import "unknown.idl";
        HRESULT getEuropeanTime( [out] int* hour, [out]
int*  minute);
        HRESULT getAmericanTime( [out] int* hour, [out]
int*  minute, [out] int*  am);
};

[  uuid(1a71c110-982e-11d3-b844-00008606a51e),
   version(1.0),
   helpstring("Library for time server")
]
library TimeLib
{
    importlib("stdole32.tlb");
    [ uuid(1a7656a0-982e-11d3-b844-00008606a51e),
      helpstring("TimeServer Class")
    ]
    coclass TimeServer
    {
        [default] interface ITime;
        interface ITimeFormat;
    };

};
```

Two interfaces **ITime** and **ITimeFormat** are specified. Both interfaces have two methods specified, for example interface ITime has the methods *setSwitchTime* and *getSwitchTime*. Interfaces can be seen as abstract classes that have no attributes. Interfaces are directly or indirectly inherited from the base interface **IUnknown**. IUnknown itself has three methods: *QueryInterface*, *AddRef* and *Release*, but more about this later.

Interfaces must be implemented in at least one class, which inherits from the interfaces. Therefore, a class implements all methods of the interfaces it inherits from. Because interfaces can be seen as abstract classes the implementation class has to implement all methods of the interface hierarchy up to the base interface IUnknown. The keyword coclass denotes the implementation class (sometimes referred to as object class) and this term will be used throughout the document. It is therefore obvious, that the class TimeServer inherits, and therefore has to implement the methods from ITime and ITimeFormat as well as the interface IUnknown.

The declaration of the coclass and it’s related interfaces are summarised by the keyword library called *type library*.

All the type libraries, classes and interfaces must be globally unique, that’s why all elements are identified by *Universal Unique Identifier* (UUID). The COM terminology for these identifiers is *Globally Unique Identifier* (GUID). Depending on the context they are called *Class Identifier* (CLSID), *Interface Identifier* (IID) or *Library Identifier* (LIBID). An example for such an identifier is - 1a658790-982e-11d3-b844-00008606a51e.

The necessary UUIDs are part of the model, because they may not be changed. These are available for interfaces, coclass and package with "TimeLibrary" and stored in the annotation item “UniqueID” which belongs to note “object”.

Each element in an IDL file is preceded with some (COM-) attributes, included in square brackets. Therefore, the UUID‘s are attributes of the elements shown in Table 1 – COM Constructs and Mapping to UML.

If a class implements more than one interface, one of the interfaces must be the default interface denoted by *[default]* in front of that interface.

Parameters of interface methods can have various COM-attributes as well, e.g. [in], [out], [in, out], etc. This information tells COM how to marshal parameters (if marshalling is required).

Let’s start collecting requirements and map them to StP/UML model elements:

COM Construct	**Mapping to UML Model Element**
Interfaces	Either classes with stereotype "Interface" or interface symbol (bubble), depending on the context
Interface methods	Methods(operations) in class tables, with parameters and return types
Implementation class (coclass) for interfaces	Class symbol with stereotype "COMObject"
Implementation relation between coclass and interfaces	Implements link between coclass and interfaces in class diagram
IDL attributes ([...])	Tagged values, added to the appropriate modelling element
Type library	Package symbol with stereotype "TypeLibrary"

Table 1 – COM Constructs and Mapping to UML

The Interface Implementing Class

An interface represents an abstract class concept, and its methods must be implemented in a derived class that is already named in the interface file with the keyword *coclass*. Therefore, this provides the name of the interface implementing class as well as its method declarations.

The header file for coclass would look like the following:

```
#include "TimeLib.h"     // Header generated by MIDL.exe

class TimeServer : public ITime, public ITimeFormat
{
    private:
    ULONG localRefCounter;
    public:
    TimeServer() {localRefCounter=0;}

    public:
    // Root interface methods
    STDMETHODIMP QueryInterface(REFIID riid, void** ppv);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);

    // Interface ITimeFormat
    private:
    // Interface ITime
    STDMETHODIMP setSwitchTime( int hour, int  minute);
    STDMETHODIMP getSwitchTime( int* hour, int*  minute);
    // Interface ITimeFormat
    STDMETHODIMP getEuropeanTime( int* hour, int*  minute);
    STDMETHODIMP getAmericanTime( int* hour, int*  minute,
int*  am);
};
```

The parameter types of our own interface methods are in pure C++ style, without the marshalling information. Here there is a further requirement for the code generator. What is required is a single representation of the parameter types in the model that map to the appropriate target style as part of the generation process.

The implementation file comprises only the methods of interface Iunknow. These methods of IUnknow serve two purposes:

QueryInterface is needed for the client to ask the object for a special interface different to that which it is already using. The required interface is delivered in parameter *riid*, if the interface is available, a pointer to it is returned in parameter *ppv*.

AddRef and *Release* are used for reference counting. If a client has access to one of the interfaces of the object it has to call AddRef, which increases an internal counter. If the client does not use the object any longer it has to call *Release* to decrement the counter. When the counter reaches NULL, the object is no longer needed and can remove itself from memory.

Since it is known what has to happen to the methods from IUnknown, it is possible to provide a standard implementation of these functions.

The Class Factory

There are different ways in which COM components can be used. The simplest way is as *In-Process Server*, which is realised as a DLL, and is loaded into the same address space as the client application. The second way is as *Local-Serve*r, that is usually an EXE file on the same machine as the client, but clearly has it's own address space. The third possibility is as *Remote-Server*. This can be either a DLL or EXE on a remote machine. The client need not know about which way the requested server is realised (although it can specify a location for the COM server).

A COM component can exist in a number of different ways, and the client does not need to specify the location of the server, it follows that a mechanism much be present to instantiate an object with the specified interface, even if it is located in a different address space. This mechanism is realised using the *class factory* concept.

Class factories are ordinary COM classes that support a special interface called *IClassFactory*, which inherits from IUnknown. For each COM class that can be addressed by a client, a class factory must be provided. Therefore, the first step for a client is to obtain a pointer to the appropriate class factory, that can instantiate as many COM objects as needed. For this purpose the class factory provides a method called *CreateInstance*. This method expects two parameters – *riid* as identification for the interface of the COM object the client wants to address and *ppv* to deliver a pointer to that interface. The client programmer can then ask *QueryInterface* for the appropriate interface. The second method is to use *LockServer*, which has to establish the class factory in memory if it is often used.

Looking at the header of such a class factory:

```
#include "TimeLib.h"          // Header generated by
MIDL.exe
#include "TimeServer.h"

class TimeServerFactory : public IClassFactory
{
    public:
    // IUnknown
    STDMETHODIMP QueryInterface (REFIID riid, void** ppv);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);

    // IClassFactory
    STDMETHODIMP CreateInstance(LPUNKNOWN punkOuter, REFIID
iid, void **ppv);
```

```
        STDMETHODIMP LockServer(BOOL fLock);

        TimeServerFactory() {localRefCounter=0;}
        private:
        ULONG localRefCounter;

    };
```

TimeServerFactory has to implement the methods *CreateInstance* and *LockServer* from IClassFactory as well as the methods from IUnknown. All the methods that can be delivered are standard implementations in the implementation file. Therefore, class factories can be virtually 100% automatically generated.

The Type Library

The include statement "TimeLib.h" appears in the headers for both the implementation class and class factory.

As discussed earlier, the declarations in the IDL file can be collected into a *type library*. This is a binary that can be delivered together with the COM server or alternatively can be obtained by compiling the IDL file with **midl.exe**, which is shipped with Microsoft's Developer Studio. A second possibility is to generate a header file with the same name as the IDL file, containing the interfaces in a COM specific way, as well as *proxy* (client) and *stub* (server) method declarations.   Including this header is necessary for our own generated headers of implementation classes and class factories

Capturing COM components in UML Class Diagrams

Now that we have established some of the heuristics for mapping from COM constructs to UML model elements we can now turn our attention to the model representation and the role that this plays in the generation of source code.

It is important to point out that the standard UML extensibility mechanisms such as stereotypes and tagged values are used, rather than proprietary notations and symbols. This is a key part of adapting the UML to support particular architectures or target languages, and as such is fully supported in the Software through Pictures tool suite.

Modelling the Interfaces and Mapping these to Code

In the class diagram below, the inheritance hierarchy for the interfaces is modelled using class symbols with the stereotype "Interface". This stereotype is necessary, so that code generator knows that it is dealing with an interface and not a class, and can thus generate code accordingly.

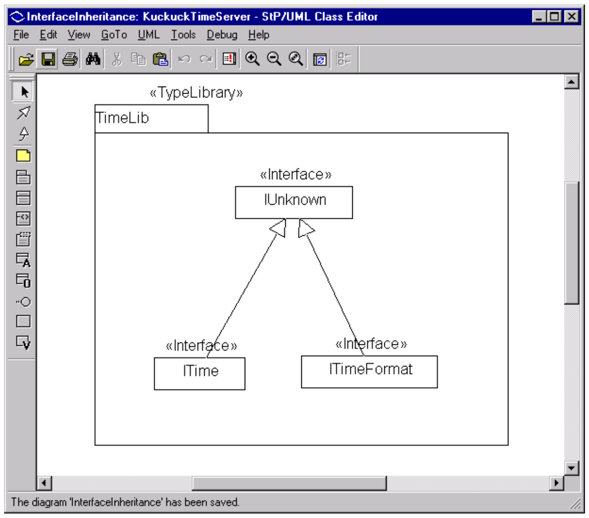
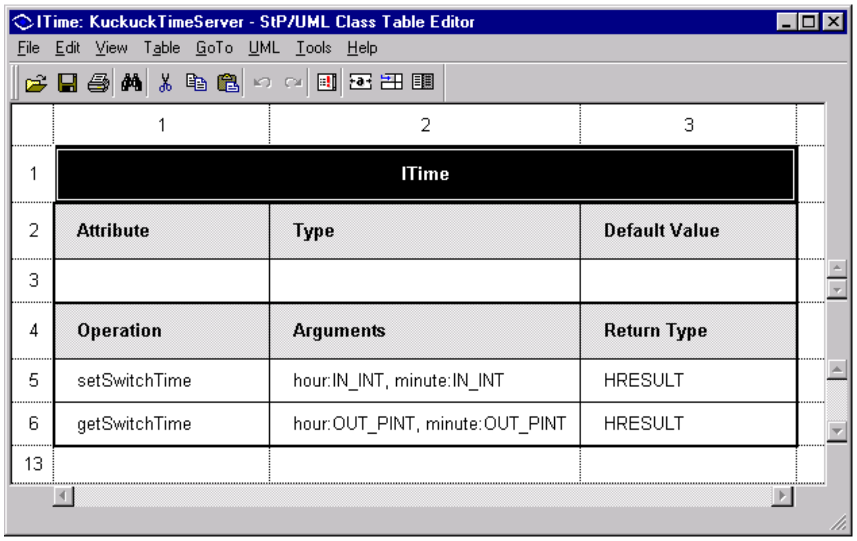

InterfaceInheritance: KuckuckTimeServer - StP/UML Class Editor
File Edit View GoTo UML Tools Debug Help
«TypeLibrary»
TimeLib
«Interface»
IUnknown
«Interface»
ITime
«Interface»
ITimeFormat
The diagram 'InterfaceInheritance' has been saved.

Diagram 1 – Modelling Interfaces in Class Diagrams

It is generally accepted good practice that class diagrams should not be used to model the internals or details of classes or interfaces and thus to specify the methods of the interfaces a class table, one for each interface, is used. The class table captures the implementation view.

The interface IUnknown is marked as external in the properties, so that the ACD generator detects that it is not necessary to generate any code. Since the methods of IUnknown are standard, there is no need to specify these methods in the class table, they can be fully generated.


ITime: KuckuckTimeServer - StP/UML Class Table Editor
File Edit View Table GoTo UML Tools Help

	1	2	3
1	ITime		
2	Attribute	Type	Default Value
3			
4	Operation	Arguments	Return Type
5	setSwitchTime	hour:IN_INT, minute:IN_INT	HRESULT
6	getSwitchTime	hour:OUT_PINT, minute:OUT_PINT	HRESULT
13			

Diagram 2 – Class Table for an Interface

Interface ITime has two methods, *setSwichTime* and *getSwichTime*, which both have two parameters and the same return type *HRESULT*.

The table below describes the stereotypes needed for modelling interfaces, to which UML model elements they are added, and how they map to source code. The code generator can extract this information from the model to generate source code correctly from the model representation.

Stereotype	**Modelling element**	**Mapping**
TypeLibrary	Package	**IDL**: Keyword library, whereby the package label represents the name of the library. The package label is used as name for the IDL file as well. **CPP**: package label is used for the automatic include statement in the top of both headers (for coclass and factory class)
Interface	Class	**IDL**: keyword interface, the label of the class symbol is used as the interface name. **CPP**: in the header of the coclass the class label of the interface is used as base class for coclass. Therefore, coclass inherits from the interface.

Table 2 – Stereotypes for Modelling Interfaces and Mapping to Source Code

Type Mapping

The parameter types are of special interest, because there is the issue of generating them with the proper type information for both the IDL file as well as the implementation file.

As mentioned earlier, the IDL file needs some extra marshalling information, so an abstract data type **IN_INT** and **OUT_PINT** are used. These abstract types will be mapped to the appropriate data type during code generation. What we now need is a clear type mapping rule. (Note: the class table for interface *ITimeFormat* looks very similiar, so there is no need to show this in a separate diagram).

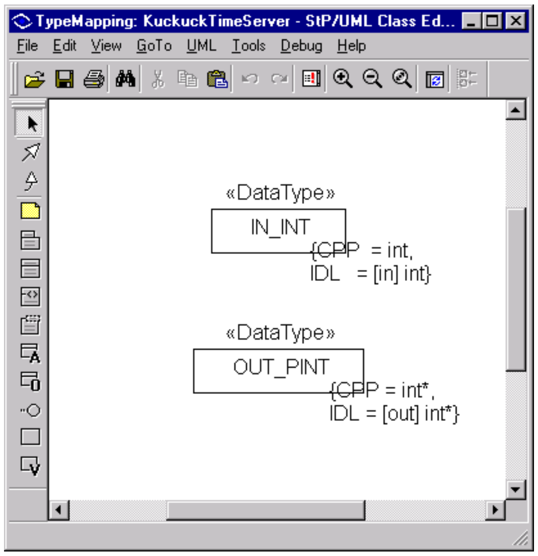

TypeMapping: KuckuckTimeServer - StP/UML Class Ed...
File Edit View GoTo UML Tools Debug Help
«DataType»
IN_INT
{CPP = int,
IDL = [in] int}
«DataType»
OUT_PINT
{CPP = int*,
IDL = [out] int*}

Diagram 3 – Modelling Data Types for Type Mapping

Abstract data types are modelled as classes with stereotype "DataType". The label of the class symbol represents the abstract data types that are used as parameter types in the class tables above, the mapping is defined with tagged values (shown in curly braces {}).

The table below describes the stereotype needed for modelling the type mapping, to which UML model elements it is added and how it maps to source code.

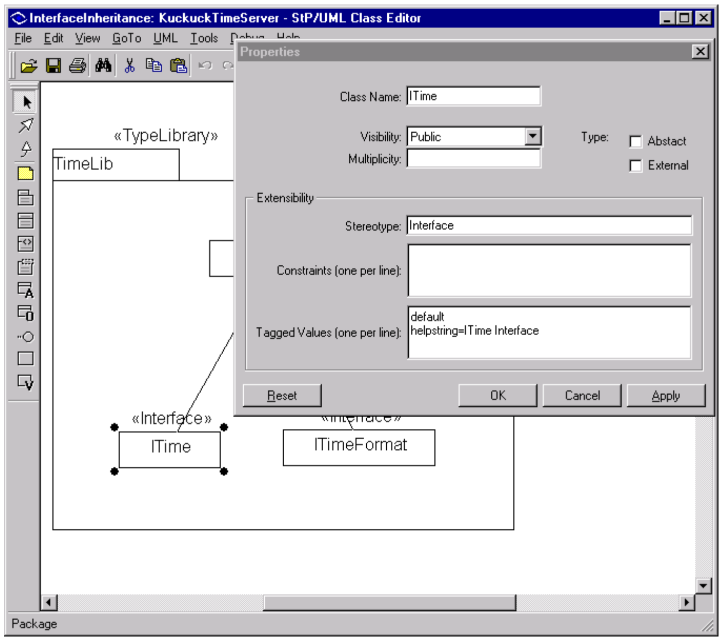
Stereotype	Modelling element	Mapping
DataType	Class	**IDL**: used for mapping to that type denoted by the value, that is associated with the tag IDL **CPP**: used for mapping to that type denoted by the value, that is associated with the tag CPP

Table 3 - Stereotypes for Modelling Type Mapping and Mapping to Source Code

IDL Specific Attributes

All the elements in the IDL file could have COM attributes (enclosed in square brackets). This information is captured in the model as *tagged values*.

The diagram below shows the property dialog for interface *ITime* and each attribute added to the *Tagged Values* field. Since ITime should be the default interface, the tagged value "*default*" is added, but without the value. (This is in conformance to the UML specification, a boolean tagged value may be written without the value TRUE, if its value were FALSE, the tag would be omitted as well).


InterfaceInheritance: KuckuckTimeServer - StP/UML Class Editor
File Edit View GoTo UML Tools
Properties
«TypeLibrary»
TimeLib
«Interface»
ITime
ITimeFormat
Reset
OK
Cancel
Apply
Package

Class Name: ITime

Visibility: Public

Multiplicity:

Type: ☐ Abstact ☐ External

Extensibility

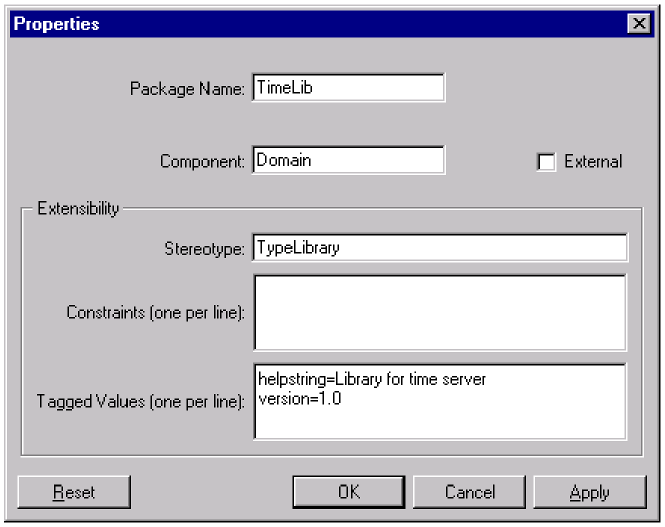
Stereotype: Interface

Constraints (one per line):

Tagged Values (one per line):
default
helpstring=ITime Interface

Diagram 4 – Dialog window for adding tagged values to class symbol

The property dialog for the package representing the type library is shown below, and contains the tagged values for the helpstring and the version attributes of the IDL file.



Package Name: TimeLib

Component: Domain

Stereotype: TypeLibrary

Constraints (one per line):

Tagged Values (one per line):
helpstring=Library for time server
version=1.0

Diagram 5 – Dialog window for adding tagged values to package symbol

In the table below you will find the tagged values needed for modelling the interfaces, to which UML model elements they are added and how they map to source code.

Tagged Value	**Modelling element**	**Mapping**
default (no value)	Interface/ Class	**IDL**: denotes the default interface in the coclass section **CPP**: in the CreateInstance method of the factory implementation it's that interface to which the new instance of coclass is cast. The QueryInterface method of this interface is called and returns a pointer to itself in parameter ppv.
helpstring=<string>	Interface	**IDL**: helpstring attribute
helpstring=<string>	package	**IDL**: helpstring attribute
version=<string>	package	**IDL**: version attribute

Table 4 – Tagged Values for adding COM Attributes to Modelling Elements and Mapping to Source Code

Modelling the coclass

Until now we specified the interfaces, introduced the package symbol for the type library and set all the necessary COM attributes for the IDL file as tagged values in the model. The type mapping for the abstract data types has been specified as well.

The last step that is required is to model the implementation class for the interfaces and introduce the information for its class factory.

Diagram 6 – Modelling coclass in the class diagram

The interfaces are drawn with UML interface symbols, having *implements* links to a class labeled with *TimeServer* and stereotyped as "COMObject". This is the way we express that class TimeServer is the (co) class implementing the interfaces ITime and ITimeFormat.

Now, only the factory class information has to be specified. This is done as a tagged value that is associated with the class object it has to instantiate. In our example, class TimeServer has the tagged value "Factory=TimeServerFactory". Where TimeServerFactory is the name of the factory class to be generated (there is an exception, where a COM class can have no class factory, so it was decided to add the tag Factory to make it explicit when a class factory should be generated).

Now we have all the necessary model information to generate the code following our requirements.

In the table below, are the stereotypes and tagged values needed for modelling the coclass, to which UML model elements they are added and how they map to source code:

Stereotype	**Modelling element**	**Mapping**
COMObject	Class (coclass)	**IDL**: keyword coclass, the label of the class represents the name of the coclass. **CPP**: class label is used as name for the class implementing the associated interfaces as well as the name for the appropriate C++ files (header and impl.). This is although the name of the class, that is instanciated by the class factory (s. although tagged value below).

Table 5 - Stereotype for Modelling the coclass and Mapping to Source Code

Tagged Value	**Modelling element**	**Mapping**
Factory=<name of factory class>	Class (coclass)	**CPP:** value used for the name of the class factory as well as the file name for the factory (header and implementation).
Helpstring=<string>	Class (coclass)	**IDL**: helpstring attribute

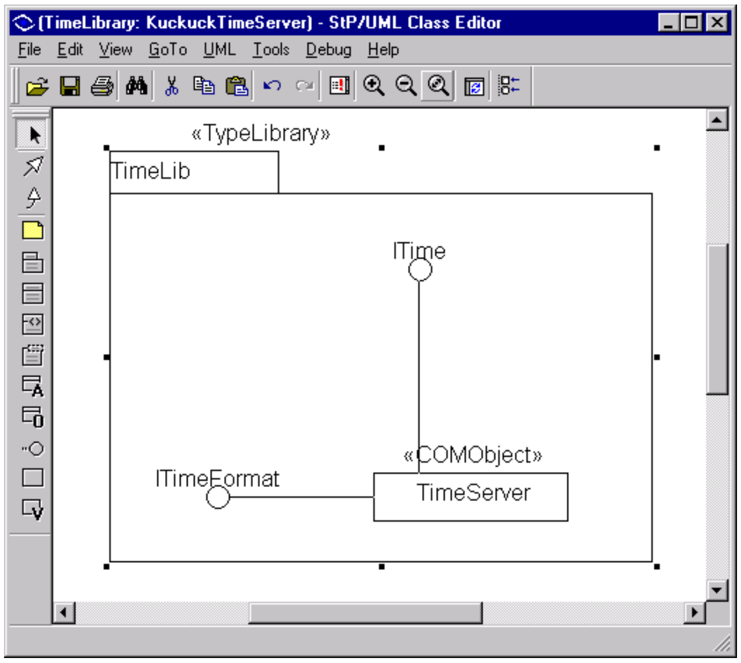

(TimeLibrary: KuckuckTimeServer) - StP/UML Class Editor
File Edit View GoTo UML Tools Debug Help
«TypeLibrary»
TimeLib
ITime
«COMObject»
TimeServer
ITimeFormat

Table 6 – Tagged values as additional information to the coclass

(Re)Using a COM component

The modelling of COM components and generating code for them has demonstrated a server that makes services available to clients. How can clients can use the services of such a component? Normally a client uses a COM component by using its interfaces. What must be modelled is a client class and a relationship to the used interface.

Take a look to the following class diagram:

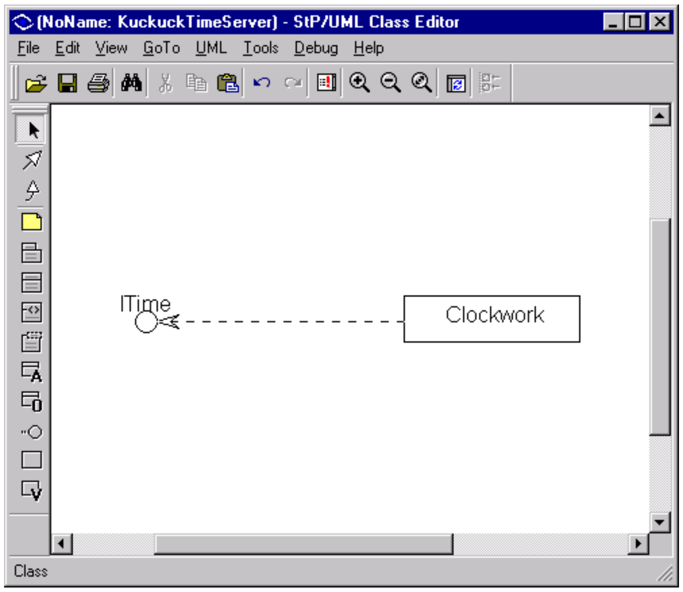

[NoName: KuckuckTimeServer] - StP/UML Class Editor
File Edit View GoTo UML Tools Debug Help
ITime
Clockwork
Class

Diagram 7 – Modelling a client class that uses an interface

This class diagram could be part of the StP system holding the client model. *Clockwork* is a class that uses a service from our COM component that is made available by the interface ITime. The relationship between Clockwork and ITime is of type *dependency*. We have therefore expressed the usage of the service by the client in modelling terms.

Summary

This paper has demonstrated the effective modelling, and generation for COM components using UML, a state of the art modelling tool and code generation technology. Significant benefits such as re-use, quality and productivity can be gained from using these approaches.