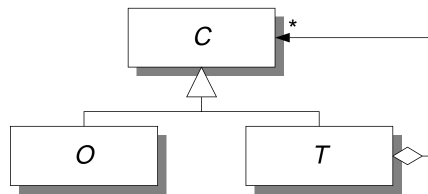
Development of the Pilot Application in Visual Basic

COT/3-11-V1.2



Centre for Object Technology

Revision history:	V0.1	10-08-98	First draft.
	V0.2	11-08-98	Format problems fixed.
	V0.3	11-08-98	More problems fixed.
	V0.4	11-08-98	More problems fixed.
	V1.0	19-08-98	Format problems fixed.
	V1.1	08-09-98	First public version.
	V1.2	08-09-98	Minor revisions.

Author(s): Henrik Lykke Nielsen, DTI

Status: Final

Publication: Public

Summary:

The report includes the investigation of Visual Basic as a COM development environment using a Pilot Application.

© Copyright 1998 Danish Technological Institute

The Centre for Object Technology (COT) is a three year project concerned with research, application and implementation of object technology in Danish companies. The project is financially supported by The Danish National

Participants are:
Maersk Line, Maersk Training Centre, Bang & Olufsen, WM-data, Rambøll, Danfoss, Systematic Software Engineering, Odense Steel Shipyard, A.P. Møller, University of Aarhus, Odense University,

Contents

1. INTRODUCTION 4

2. PILOT APPLICATION IMPLEMENTATION PROCESS 4

2.1 PHASE 1 4
2.1.1 Phase 1.1 - Create a component with an arithmetic interface 4
2.1.2 Phase 1.2 - Change the existing members of an interface 12
2.1.3 Phase 1.3 - Add a member to an existing interface 15
2.2 PHASE 2 16
2.2.1 Phase 2.1 - Add an interface to Deep_Thought's Calc component 16
2.2.2 Phase 2.2 - Remove an interface from Deep_Thought's Calc component 17
2.3 PHASE 3 17
2.3.1 Phase 3.1 - Add a component to Deep_Thought 17
2.3.2 Phase 3.2 - Remove a component from Deep_Thought 17
2.4 PHASE 4 18
2.4.1 Phase 4.1 - Delegation 18
2.4.2 Phase 4.2 - Aggregation 19
2.5 PHASE 5 - CREATE THE DEEP_THOUGHT SERVER 19
2.6 PHASE 6 - CREATE A CLIENT WITH A GRAPHICAL INTERFACE 20
2.7 PHASE 7 - CHANGE DEEP_THOUGHT TO IMPLEMENT THE IDISPATCH INTERFACE 20
2.8 PHASE 8 - CREATE INSTALLATION DISKS 20
2.9 PHASE 9 - EXCHANGE IMPLEMENTATIONS 20

1. Introduction

This document describes how Visual Basic can be used for implementing the COM taxonomy pilot application. A description of the application can be found in COT-3-15: The COM Pilot Application.

This report is part of an evaluation of four development environments with respect to their support for COM based development. The main document of this evaluation is COT-3-4: Evaluation of COM Support in Development Environments.

The intention of the document is to give the reader a feeling and understanding of how Visual Basic can be used for solving COM related tasks. Because Visual Basic's approach is based on abstraction by wrapping functionality into predefined packages, we will most probably encounter limitations in Visual Basic with respect to its handling of COM. Some of these limitations could be overcome by using additional tools. We will in this investigation limit ourselves to examine how the standard Visual Basic development environment can be used. Breaking the barriers of standard Visual Basic by using the Windows API, using external tools (i.e. Notepad) etc., is a very important task. Though it is not a task undertaken here.

2. Pilot Application Implementation Process

Two primary ways exists to implement interfaces in Visual Basic. The first is the standard way where the interface is defined by defining its members inside Visual Basic itself. The second way covers the situation where you want to implement an existing interface defined in an external type library. Both ways will be investigated here.

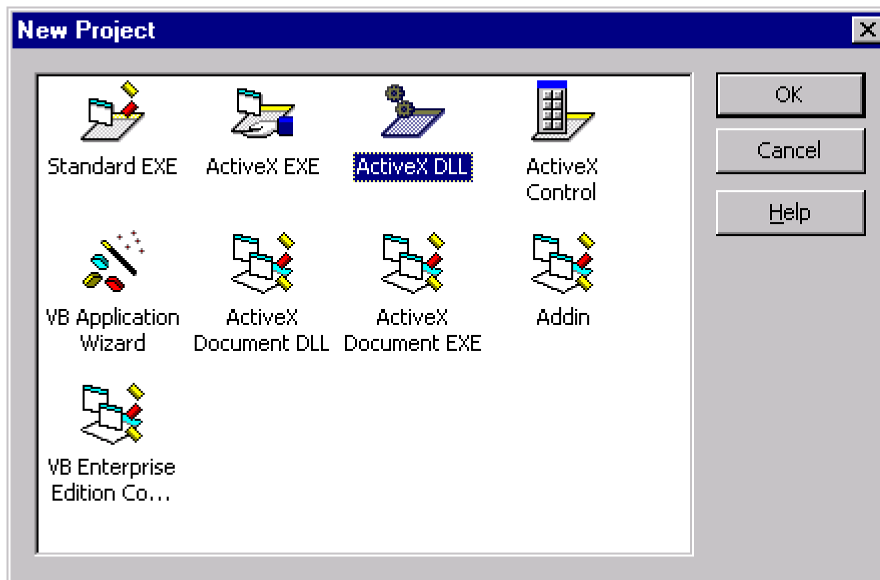
Standard Visual Basic only allows the creation of servers with dual interfaces (with both IDispatch and custom interfaces). It is not possible in standard Visual Basic to create servers based directly on IUnknown. So the phases 1 to 6 is strictly speaking not possible to implement in standard Visual Basic. But to give the reader a feeling of the philosophy in developing COM based components with Visual Basic the phases have been dealt with in spite of this restriction. You should note however that the implementation of the phases has all been based on dual interfaces!

The ITypeLib Viewer function in the OLE/COM Object Viewer (part of the VC++ package) can be used to view the IDL files corresponding to the respective servers.

2.1 Phase 1

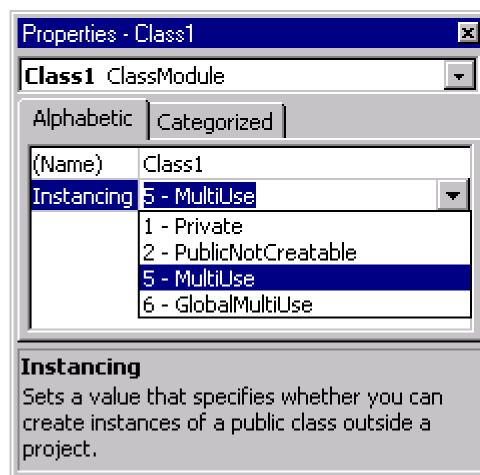
2.1.1 Phase 1.1 - Create a component with an arithmetic interface

To create a DLL-server select the menu-item "File\New Project". This opens up the following window:



Select "ActiveX DLL". We now have the skeleton of an ActiveX automation server.
Select the menu-item "Project\Project1 Properties..." and set *Project Name* = "Deep_Thought".

When starting a new ActiveX project a *ClassModule* is by default added to the project. A Visual Basic *ClassModule* has two functions: The *ClassModule* itself defines the type of objects that can be instantiated in the server. And the members of the *ClassModule* define the default interface for objects instantiated from the *ClassModule*. So a *ClassModule* with the *Name* = "Class1" defines both an object type "Class1" and a default interface "Class1". *ClassModule*s have an *Instancing* property that specifies instantiation and scope of the *ClassModule* with respect to the client. For ActiveX DLL's the *Instancing* property can take one of the following values:



A single default interface (Deep_Thought1_1a)

To create and use a class with a single default interface is very easy in Visual Basic. First define a *ClassModule* "Class" by defining the members and their implementation directly in the ClassModule as standard public Visual Basic methods and set the Instancing property of "Class" to *5 - Multiuse* (the default value).

Code for the "Calc" *ClassModule*:

```
Public Function Dad(ByVal n1 As String) As String

    Dad = n1

End Function

Public Function Divide(ByVal n1 As String, ByVal n2 As String) _
                                                        As String

    Divide = n1 & "/" & n2

End Function

Public Function Multiply(ByVal n1 As String, _
                              ByVal n2 As String) As String

    Multiply = n1 & "*" & n2

End Function

Public Function Subtract(ByVal n1 As String, _
                              ByVal n2 As String) As String

    Subtract = n1 & "-" & n2

End Function
```

Visual Basic does not as default produce independent type-libraries but rather type-libraries embedded in the server-dll itself. To create a server-dll select the menu-item "File\Make Deep.dll...". This will compile the dll as well as register the server (the same way RegSrv32.exe does it).

Note that Visual Basic names the coClass "Calc" just as the *ClassModule* and that the default interface is named "_Calc". The underscore is prefixed to the name of the *ClassModule* to avoid name clashing:

```
coclass Calc {
        [default] interface _Calc;
```

There is no way in the Visual Basic environment for the developer to define the name of the interface himself (to something independent of the class name that is).

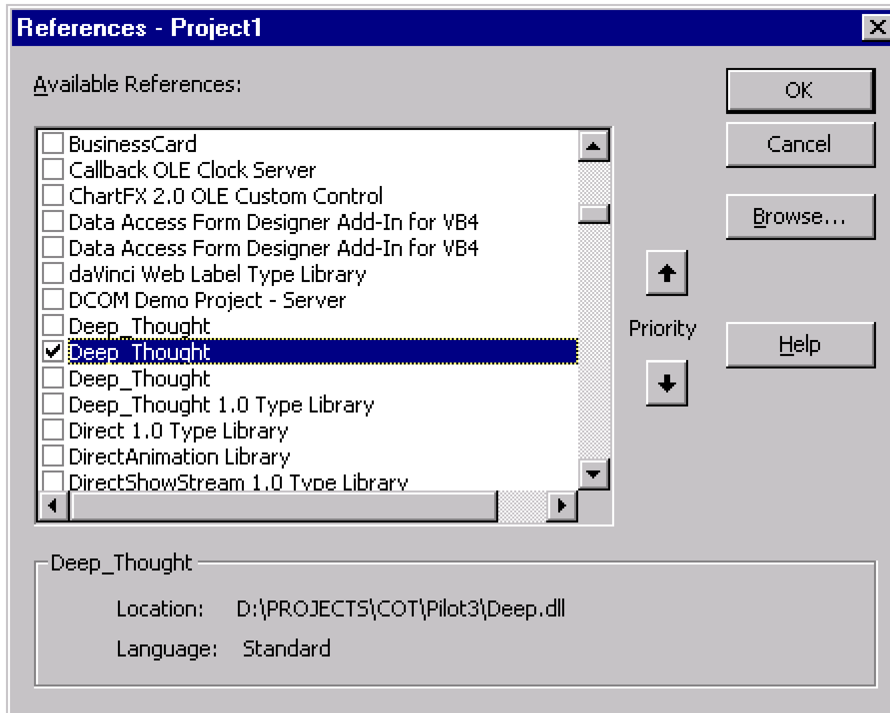
Building a client for Deep_Thought1_1a.dll

Create a new Visual Basic project (Standard EXE) and create the desired user interface.

If you want to use the default interface of a component there exists two ways to bind to the server: Early binding and late binding. In the last case it is possible by specifying the server- and class-name as a string, and this will automatically use the default interface of the class. But if you want to use a non-default interface defined in the server it is only possible to specify the interface-name as an identifier not as a string.

Early bound client for Deep_Thought1_1a

Select the menu-item "Project\References..." which opens up the following window:



Select the compiled server-dll. This will give the Visual Basic program a reference to the type-library embedded in the dll-file.

There are three ways to instantiate classes in Visual Basic.

```
1. Dim objCalc As New Deep_Thought1_1a.Calc

2. Dim objCalc As Deep_Thought1_1a.Calc

   Set objCalc = CreateObject("Deep_Thought1_1a.Calc")

3. Dim objCalc As Deep_Thought1_1a.Calc

   Set objCalc = New Deep_Thought1_1a.Calc
```

Way 1. delays the instatiation of the class until the first time one of its members is called.

Way 2. and 3. have the same functionality except that 2. allows the instantiation of a specific version of the component as in the following example:

```
Set objCalc = CreateObject("Excel.Application.8")
```

The following code is a simple example of how to early-bind to, and use the default interface of the Calc component from a client.

Code for the early bound client for Deep_Thought1_1a.dll:

```
Dim objCalc As Deep_Thought1_1a.Calc

Set objCalc = CreateObject("Deep_Thought1_1a.Calc")

MsgBox objCalc.Dad("This is just a string")

MsgBox objCalc.Divide("Hello ", " Barbie")

MsgBox objCalc.Multiply(n1:="Hello ", n2:=" Ken")

MsgBox objCalc.Subtract(n2:=" Basic", n1:="Visual ")
```

Late bound client for Deep_Thought1_1a.dll

In this case there is no need to make a reference to the compiled server-dll. And creating instances is only possible using *CreateObject*,

The following code is a simple example of how to late-bind to, and use the default interface of the Calc component from a client.

Code for the late bound client for Deep_Thought1_1a.dll

```
Dim objCalc As Object

Set objCalc = CreateObject("Deep_Thought1_1a.Calc")

MsgBox objCalc.Dad("This is just a string")

MsgBox objCalc.Divide("Hello ", " Barbie")

MsgBox objCalc.Multiply(n1:="Hello ", n2:=" Ken")

MsgBox objCalc.Subtract(n2:=" Basic", n1:="Visual ")
```

Multiple interfaces (Deep_Thought1_1b)

To create other interfaces than the default one use interface inheritance: First define a *ClassModule* "IClass". This *ClassModule* will function as an abstract base class that defines the members of the interface. Define the members by creating standard members but without the implementation code. The *Instancing* property for the "IClass" *ClassModule* should be set to *2-PublicNotCreatable*, so that clients are not able to create instances of the "IClass" *ClassModule*.

Code for the "ICalc" *ClassModule*:

```
Public Function Dad(ByVal n1 As String) As String

End Function

Public Function Divide(ByVal n1 As String, ByVal n2 As String) _
                                                         As String

End Function

Public Function Multiply(ByVal n1 As String, _
                              ByVal n2 As String) As String

End Function

Public Function Subtract(ByVal n1 As String, _
                              ByVal n2 As String) As String

End Function
```

Then define a *ClassModule* "Calc" which implements the "ICalc" interface.
The *Instancing* property for the "Class" *ClassModule* should be set to *5 -MultiUse*.

Code for the "Calc" *ClassModule*:

```
Implements ICalc

Private Function ICalc_Dad(ByVal n1 As String) As String

  ICalc_Dad = n1

End Function

Private Function ICalc_Divide(ByVal n1 As String, _
                              ByVal n2 As String) As String

  ICalc_Divide = n1 & "/" & n2

End Function

Private Function ICalc_Multiply(ByVal n1 As String, _
                              ByVal n2 As String) As String

  ICalc_Multiply = n1 & "*" & n2

End Function

Private Function ICalc_Subtract(ByVal n1 As String, _
                              ByVal n2 As String) As String

  ICalc_Subtract = n1 & "-" & n2

End Function
```

Early bound client for Deep_Thought1_1b.dll:

Set a reference to the server-dll "Deep_Thought1_1b".

Code for the early bound client for Deep_Thought1_1b.dll:

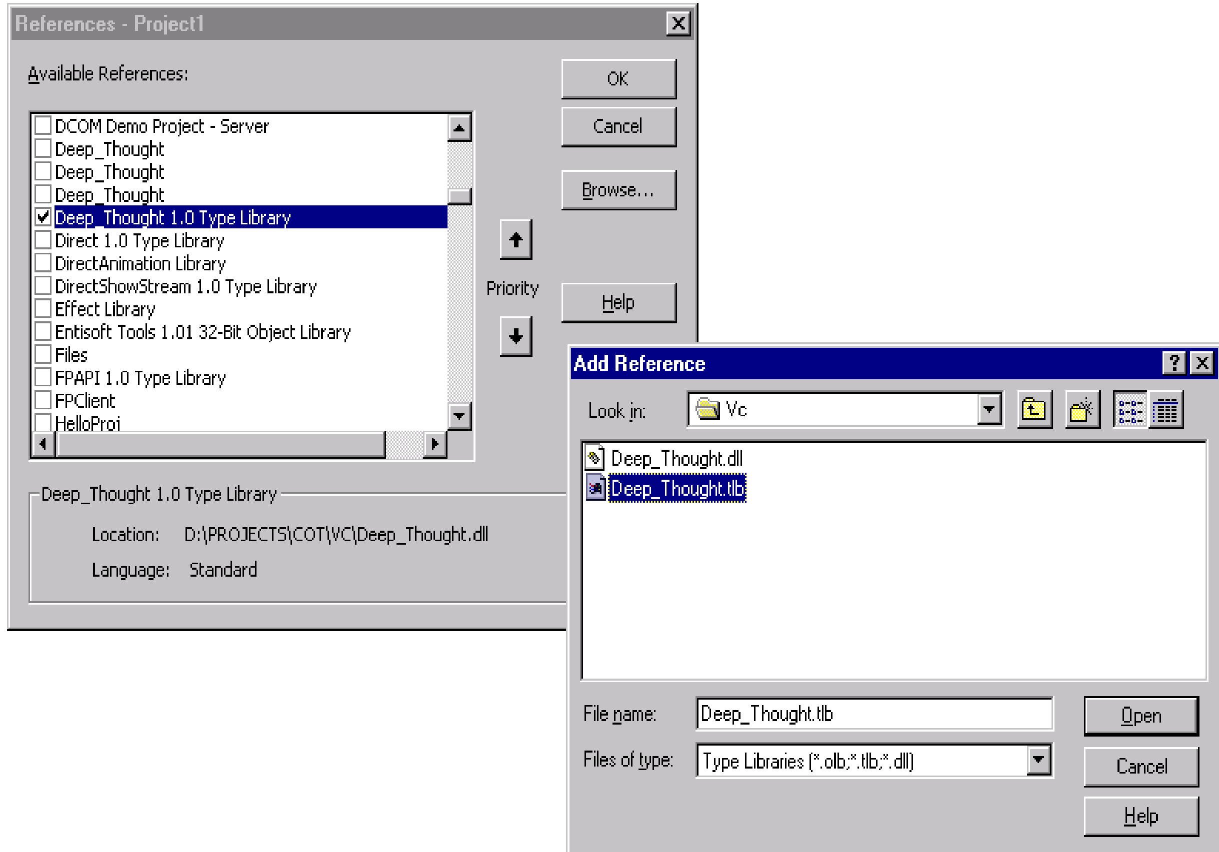
```
Dim objCalc As Deep_Thought1_1b.Calc
```

```
Dim infCalc As Deep_Thought1_1b.ICalc

Set objCalc = CreateObject("Deep_Thought1_1b.Calc")
Set infCalc = objCalc

MsgBox infCalc.Dad("This is just a string")

MsgBox infCalc.Divide("Hello ", " Barbie")

MsgBox infCalc.Multiply(n1:="Hello ", n2:=" Ken")

MsgBox infCalc.Subtract(n2:=" Basic", n1:="Visual ")
```

Inheriting from a type-library:

There is no way inside the Visual Basic development environment itself to specify the usage of specific UUID’s. But it is possible to implement interfaces specified in type-libraries. So if a type-library with a specification for ICalc already exists (e.g. by constructing an IDL file and compiling it into a type-library with MIDL) you do the following to implement "Calc":

Set a reference to the type-library:



Then define a ClassModule "Calc" which implements the "ICalc" interface defined in the Deep_Thought.tlb file:

Code for the "ICalc" *ClassModule*:

```
Implements DEEP_THOUGHTLib.ICalc

Private Function ICalc_Dad(ByVal n1 As String, ByVal n2 As String) _
                                                            As String

  ICalc_Dad = n1 & "+" & n2

End Function

Private Function ICalc_Divide(ByVal n1 As String, _
                                    ByVal n2 As String) As String

  ICalc_Divide = n1 & "/" & n2

End Function

Private Function ICalc_Multiply(ByVal n1 As String, _
                                    ByVal n2 As String) As String

  ICalc_Multiply = n1 & "*" & n2

End Function

Private Function ICalc_Subtract(ByVal n1 As String, _
```

```
                                          ByVal n2 As String) As String
  ICalc_Subtract = n1 & "-" & n2
End Function
```

Early bound client for Deep_Thought1_1c.dll:

Set a reference to both the server-dll and the type-library file.

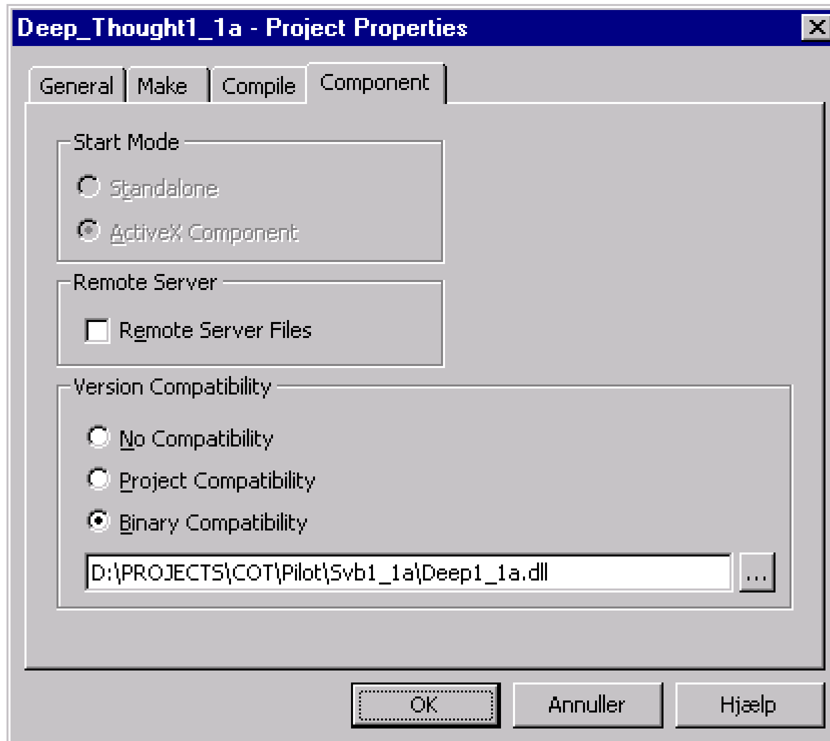
The following code is a simple example of how to use the ICalc interface from a client.

```
Dim objCalc As Deep_Thought1_1c.Calc
Dim infCalc As DEEP_THOUGHTLib.ICalc

Set objCalc = CreateObject("Deep_Thought1_1c.Calc")
Set infCalc = objCalc

MsgBox infCalc.Dad("This is just a string ", _
        " And this is just another string")

MsgBox infCalc.Divide("Hello ", " Barbie")

MsgBox infCalc.Multiply(n1:="Hello ", n2:=" Ken")

MsgBox infCalc.Subtract(n2:=" Basic", n1:="Visual ")
```

2.1.2 Phase 1.2 - Change the existing members of an interface

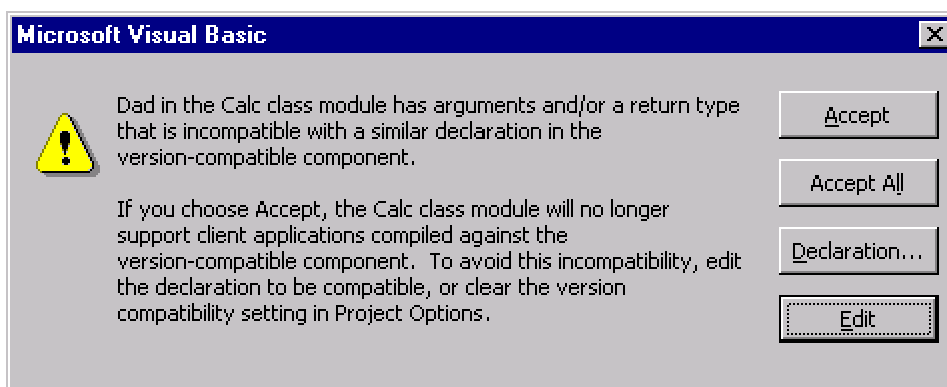
By setting version compatibility for the component to *Binary Compatibility*, and pointing to the dll-file compiled in phase 1.1, the developer achieves two goals: To give Visual Basic the ability to check whether interfaces changes in the component, and to give Visual Basic the ability to reuse UUIDs created during earlier compilations.

Select the menu-item "Project\Deep_Thought1_1a Properties...":

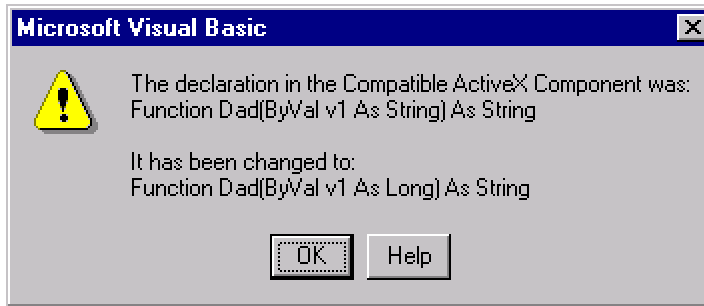


When you try to compile a server which interface is defined in a previously compiled server using Binary Compatibility and that interface has changed Visual Basic will bring up a warning screen.

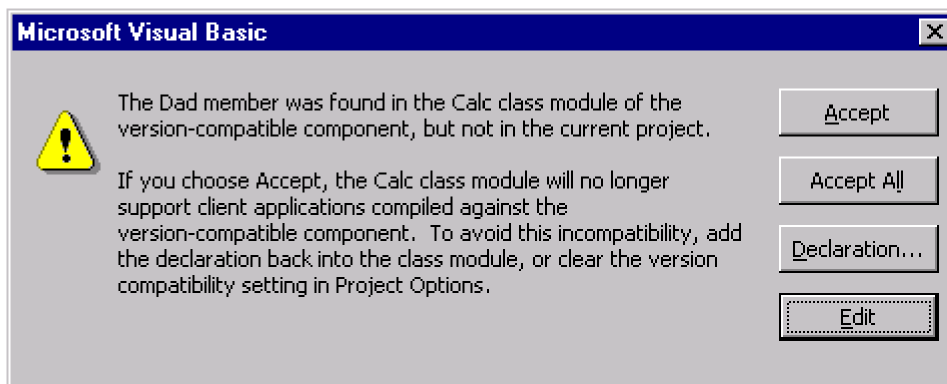
If the parameter type or the return-type of members change, if a parameter changes from being passed ByVal to being passed ByRef or if the number of parameters change the following screen will be presented:



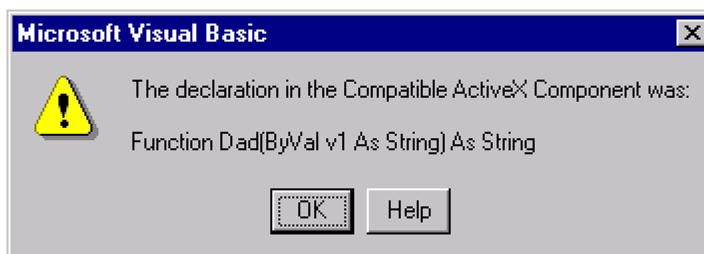
Selecting the "Declaration..." button show the following messagebox:



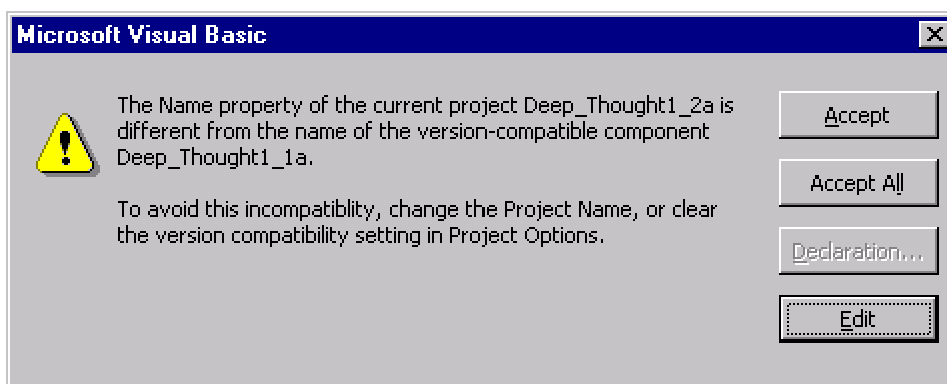
If the name of a member changes then the following screen will be presented:



Selecting the "Declaration..." button show the following messagebox:



If the Name of the project changes the following screen is presented:



If the parameter names change no warning is presented.

Compiling the Deep_Thought after changing the interface shows the warnings described above.

Forcing the compilation (or removing the *Binary Compatibility* option) creates and register the server.

Changes to the code of both the servers and clients are necessary but trivial and without any problems.

Deep1_1a.dll is changed to Deep1_2a.dll and Deep1_1b.dll is changed to Deep1_2b.dll.

2.1.3 Phase 1.3 - Add a member to an existing interface

Adding members to an already existing interface is very simple. Just add the public methods to the class and the abstract base class (if the interface isn't the default one)..

A single default interface (Deep_Thought1_3a)

Compiling the server reuses the UUIDs in the server compiled in phase 1.2 except for the interface UUID (for the _Calc interface when the default interface of a component is extended).

The client only has to be changed if the new members functionality is used. If extension of the client's functionality is not necessary then the existing client will still work and there is no need to even recompile.

Multiple interfaces (Deep_Thought1_3b)

Compiling the server reuses the UUID's in the server compiled in phase 1.2 except for the interface UUID for the ICalc interface (the UUID's for the type-libraries, the coClass and for the default interface _Calc are all unchanged).

The interface UUID for the ICalc interface gets versioned as can be seen in the IDL file:

```
[
        odl,
        uuid(28446527-A62C-11D1-91F3-006097636369),
        version(1.1),
        dual,
        nonextensible,
        oleautomation
]
interface ICalc : IDispatch {

...

typedef [uuid(F24A0096-A62B-11D1-91F3-006097636369), version(1.0), public]
```

ICalc ICalc___v0;

typedef [uuid(28446526-A62C-11D1-91F3-006097636369), version(1.1), public]

ICalc ICalc___v1;

Two version dependent UUID's are defined in the IDL: The UUID for ICalc___v0 is the UUID for ICalc in phase 1.2. The UUID for ICalc___v1 identifies the new version of the interface. A new UUID for the ICalc interface is created and both the version dependent UUID's forwards to this interface.

The client created to test Deep_Thought1_2b.dll will not run without changing the reference to the new version of Deep_Thought1_2b.dll. When a previously compiled client runs against the new server the interface casting

```
Set infCalc = objCalc
```

raises the run-time error: Type mismatch.

Changing a client to take advantage of the member is very easy and trivial.

2.2 Phase 2

2.2.1 Phase 2.1 - Add an interface to Deep_Thought's Calc component

There is no important difference in adding a new interface to Deep_Thought1_3a and Deep_Thought1_3b so we will focus on Deep_Thought1_3b.

We have already seen how to add an interface to a component:
Add a *ClassModule* "IConv" to function as an abstract base class to the server. Set its instancing property to *2 - PublicNotCreatable*. Define the members of the interface as empty procedure definitions without any implementation code.
Finally add the following code to the "Calc" *ClassModule*:

```
Implements IConv

Private Function IConv_Dec2Hex(ByVal v As Long) As String

  IConv_Dec2Hex = Hex$(v)

End Function

Private Function IConv_Hex2Dec(ByVal v As String) As Long

  IConv_Hex2Dec = Val("&H" & v)

End Function
```

Compile the extended server using *Binary Compatibility*.

All UUID's remain the same.

Existing clients will only have to be changed and recompiled if they want to take advantage of the new interface. The usage of the new interface is similar to the usage of the "ICalc" interface.

2.2.2 Phase 2.2 - Remove an interface from Deep_Thought's Calc component

Removing the "IConv" interface and its implementation is a very simple process and removing the "IConv" interface and its implementation doesn't present any problems for the client which uses the "IConv" interface ; ^) . In fact it wasn't possible to compile until the name of the server was changed and the *Binary Compatibility* option was removed. Visual Basic didn't come up with any error messages which might seem a bit strange.

A completely new type-library was generated when the new server was compiled.

2.3 Phase 3

2.3.1 Phase 3.1 - Add a component to Deep_Thought

Adding a "Conv" component with an "IConv" interface to the server presents no problems.
It is very simple and straightforward, and existing clients do not have to be recompiled.

Changing existing clients to make use of the new component is very straightforward too.

All existing UUID's remain the same.

2.3.2 Phase 3.2 - Remove a component from Deep_Thought

Removing the "Conv" component and its associated "IConv" interface and its implementation is a very simple process. When the server is compiled with *Binary Compatibility* Visual Basic presents a warning. And the compiled server reuses no UUID's.

Existing clients have to be recompiled and new references have to be made to the server.

2.4 Phase 4

2.4.1 Phase 4.1 - Delegation

Delegation doesn't involve any features of Visual Basic which we haven't already seen.

The specification of the "ICalc2" interface looks like this:

```
Public Function Power(ByVal v As Long) As Long

End Function
```

And the code for the "CalcDel" component looks like this:

```
Implements ICalc
Implements ICalc2

Private objCalc As Calc
Private infCalc As ICalc

Private Sub Class_Initialize()

  Set objCalc = New Calc

  Set infCalc = objCalc

End Sub

Private Function ICalc_Add(ByVal v1 As Long, ByVal v2 As Long) _
                                                        As Long

  ICalc_Add = infCalc.Add(v1, v2)

End Function

Private Function ICalc_Divide(ByVal v1 As Long, _
                                    ByVal v2 As Long) As Long

  ICalc_Divide = infCalc.Divide(v1, v2)

End Function

Private Function ICalc_Modulus(ByVal v1 As Long, _
                                    ByVal v2 As Long) As Long

  ICalc_Modulus = infCalc.Modulus(v1, v2)

End Function

Private Function ICalc_Multiply(ByVal v1 As Long, _
                                    ByVal v2 As Long) As Long

  ICalc_Multiply = infCalc.Multiply(v1, v2)

End Function

Private Function ICalc_Subtract(ByVal v1 As Long, _
                                    ByVal v2 As Long) As Long

  ICalc_Subtract = infCalc.Subtract(v1, v2)
```

```
End Function

Private Function ICalc2_Power(ByVal v As Long) As Long

  ICalc2_Power = v * v

End Function
```

The client code look like this:

```
Dim objCalc As Deep_Thought4_1b.CalcDel
Dim infCalc As Deep_Thought4_1b.ICalc
Dim infCalc2 As Deep_Thought4_1b.ICalc2

Set objCalc = New Deep_Thought4_1b.CalcDel

Set infCalc = objCalc
Set infCalc2 = objCalc

MsgBox infCalc.Add(6, 2)

MsgBox infCalc.Divide(6, 2)

MsgBox infCalc.Multiply(6, 2)

MsgBox infCalc.Modulus(6, 2)

MsgBox infCalc.Subtract(6, 2)

MsgBox infCalc2.Power(16)
```

2.4.2 Phase 4.2 - Aggregation

Aggregation (in the COM sense) is not possible in Visual Basic (?).

2.5 Phase 5 - Create the Deep_Thought server

The final version of the Deep_Thought server is constructed using designs already studied in the previous phases.

To change Deep_Thought to be an EXE select the menu-item "Project\Deep_Thought Properties...". Change "Project Type" to "ActiveX EXE" and recompile the server.

All UUID's are the same as for the DLL server.

If *Binary Compatibility* has been used; existing clients (which have been compiled against the dll version of the server) will continue to work against the new EXE version of the server without changes or recompilation.

To change the EXE version of Deep_Thought to be an DLL select the menu-item "Project\Deep_Thought Properties...". Change "Project Type" to "ActiveX DLL" and recompile the server.

All UUID's are the same as for the EXE server.

If *Binary Compatibility* has been used; existing clients (which have been compiled against the dll version of the server) will continue to work against the new DLL version of the server without changes or recompilation.

2.6 Phase 6 - Create a client with a graphical interface

A simple graphical client is constructed using Visual Basic intrinsic controls.

2.7 Phase 7 - Change Deep_Thought to implement the IDispatch interface

This part is really simple as all Visual Basic components supports both vTable and IDispatch bindings automatically (see the first section: "Pilot Application Implementation Process").

So we have in other words already created the specified servers supporting IDispatch interfaces.

2.8 Phase 8 - Create installation disks

Visual Basic comes with "Setup Wizard" which is a tool to create installation programs with. Many other third-party tools exists for creating installation programs. What is rather unique about setup wizard is that is integrates with the Visual Basic development environment. Setup Wizard determines which servers are referenced from the program (be it a standard EXE program or a server itself) and which files therefore should be distributed with the program.
Setup Wizard handles installation and registration of servers automatically (as do most installation tools).

2.9 Phase 9 - Exchange implementations

During the exchange of servers one rather irritating problem popped up:
When implementing an interface defined in a type-library Visual Basic insisted on overwriting the components ClassID. In practical terms this means that the developer has no way of defining the GUID for a component being created in Visual Basic. That is if you won't use the method mentioned by one of the developers on the Visual Basic team when he was asked for a solution:
"Alternately, you can binary search a made exe/dll for all occurrences of the offending GUIDs and change them to what you want, then set compatibility to that file, but I'm not recommending the approach :)".

Well as I said: In practical terms this means that…