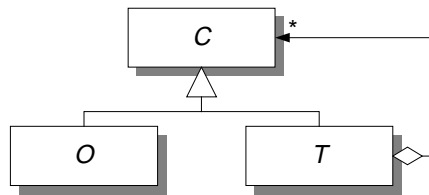


*Development of the Pilot Application in
Visual C++
COT/3-12-V1.0*



Centre for Object Technology

Revision history: V1.0 19-08-98 First version.

Author(s): Kristian Lippert, DTI

Status: Final

Publication: Public

Summary:

The report includes the investigation of Visual C++/ATL as a COM development environment using a Pilot Application.

© Copyright 1998 Danish technological Institute

Contents

1. INTRODUCTION.....	4
2. PILOT APPLICATION IMPLEMENTATION PROCESS	4
2.1 PHASE 1	4
2.1.1 Phase 1.1	4
2.1.2 Phase 1.2: Changing the interface	14
2.1.3 Phase 1.3: Adding a method to an existing interface	15
2.2 PHASE 2	15
2.2.1 Phase 2.1: Adding an interface to the component.....	15
2.3 PHASE 3	16
2.3.1 Phase 3.1	16
2.3.2 Phase 3.2	16
2.4 PHASE 4: AGGREGATION	16
2.4.1 Phase 4.1: Aggregation using delegation.....	16
2.4.2 Phase 4.2 Aggregation through interface aggregation	18
2.5 PHASE 5: THE FUTURE DEEP_THOUGHT SERVER	20
2.6 PHASE 6: A CLIENT WITH A GRAPHICAL INTERFACE	20
2.7 PHASE 7: IMPLEMENTING DEEP_THOUGHT THROUGH DISPATCH INTERFACES	20

1. Introduction

This document describes how Visual C++ and the ATL library 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.

2. Pilot Application Implementation Process

ATL means ActiveX Template Library. It is a C++ template based library that uses multiple inheritance to implement COM objects. The purpose of this library is to be able to create as small and efficient COM objects as possible without losing the flexibility a library can create. The smallness and effectiveness of the objects are such an important issue that sometimes it is more important to save 4 bytes than creating an easy and understandable library.

In the following presentation we show how to implement the Pilot Application: Deep_Thought using ATL. During the presentation we will describe what VC++, MIDL, ATL and the Wizards do.

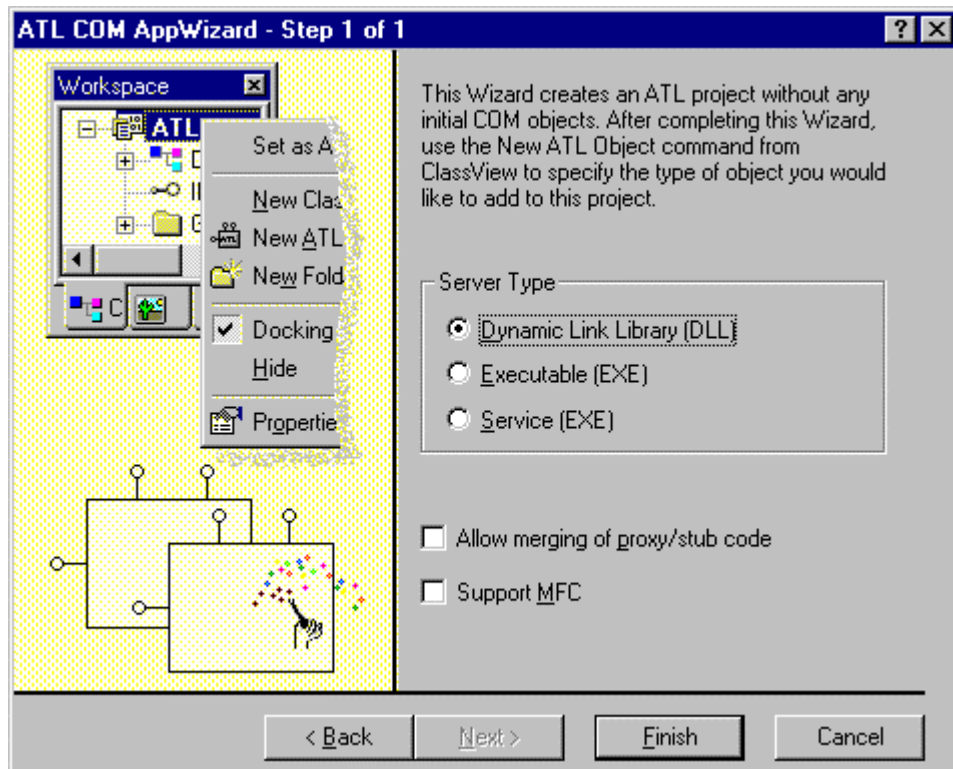
2.1 Phase 1

2.1.1 Phase 1.1

In this phase we will thoroughly see what to do and what is done when using ATL to create a simple COM component with a custom interface.

Creating the DLL Server for the Component

When starting this project an ATL Wizard guides you through the steps of creating an ATL DLL server.



The Wizard generates a project where the most important files are the following files:

Deep_thought.cpp
Deep_thought.def
Deep_thought.idl

Deep_thought.cpp

This file contains the usual functions for creating and deleting an inproc COM server:

```
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID )
STDAPI DllCanUnloadNow(void)
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
STDAPI DllRegisterServer(void)
STDAPI DllUnregisterServer(void)
```

The DllMain is the default DLL entry point so if you want to manipulate some TLS (Thread Local Storage) is it possible. It also gives you a default Object Factory from which COM objects can be created.

As seen we did not select **Allow merging of proxy/stub code**. This means that if we want Custom Marshalling we will have to create a separate DLL that contains this marshalling code. This can be done using the command line:

```
nmake -f Deep_Thoughtps.mk
```

This step can't be performed before you have added some objects and compiled the project from the IDE.

Object Maps and Class Factory

The file also contains a variable `_Module` of type `CComModule`. This is the ClassFactory. It can be returned by the method `DllGetClassObject`. When creating new objects it makes a lookup using the **Object Map**. The **Object Map** is also contained in .cpp file. Before any objects are added to the server it is empty, like:

```
BEGIN_OBJECT_MAP(ObjectMap)
END_OBJECT_MAP()
```

Deep_thought.def

Exports the DLL methods as defined in `deep_thought.cpp`, like:

```
LIBRARY "Deep_Thought.DLL"
EXPORTS
DllCanUnloadNow @1 PRIVATE
DllGetClassObject @2 PRIVATE
DllRegisterServer @3 PRIVATE
DllUnregisterServer @4 PRIVATE
```

Deep_thought.idl

This file contains the framework for the future typelibrary of the DLL server. It is fairly simple from the beginning. The Wizard generates a default UUID for the library. If you want to change it you can use an old one or generate a new one using the VC++ utility "uuidgen.exe". The default IDL looks like:

```
import "oaidl.idl";

import "ocidl.idl";

library DEEP_THOUGHTLib
{
importlib("stdole32.tlb");
importlib("stdole2.tlb");
};
```

The two imported idl files contain.....

Adding COM objects

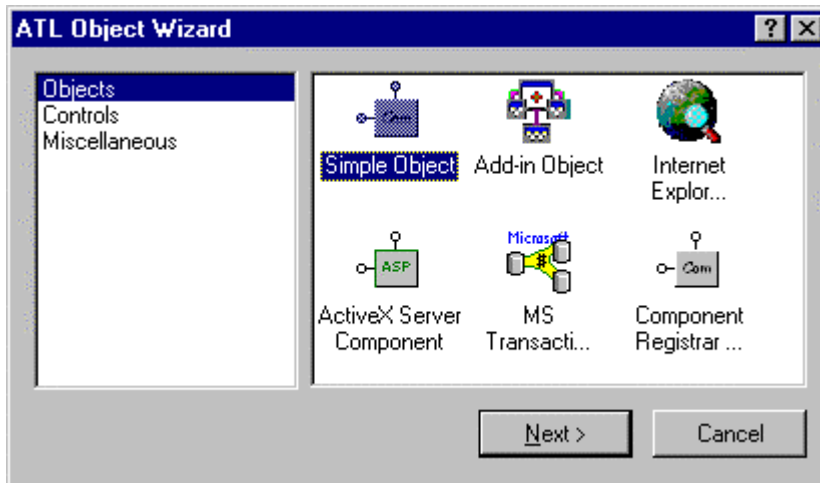
You now have an empty DLL server. You can't instantiate any objects from the DLL server yet. Before this is possible you have to add classes to the DLL server. This can be done in two ways: By hand or using a Wizard. Adding a class is a tedious and error prone process so the Wizard is the best choice.

There are at least two different ways to add an ATL class with custom interfaces to the project. The first one is simple and it is the one you will use when developing using "trial and error". In the second you can add many interfaces to the class at the same time.

Wizard Way 1

In the first phases we use COM objects that have custom interfaces. You can activate the wizard by choosing "New ATL object" from the "Insert" menu or by pressing right-mouse-button on the "...Classes" in the class view and the choosing "New ATL Object..."

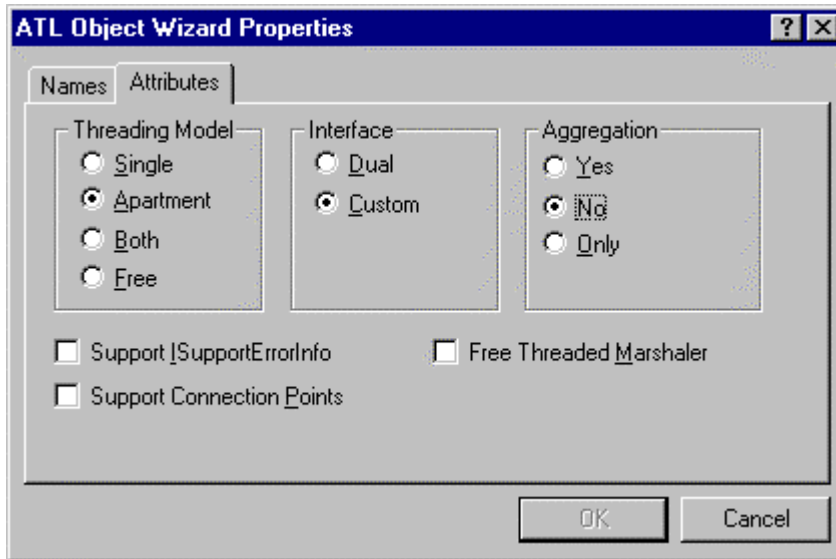
When using the Wizard you choose Simple object and press "Next".



You now have the choice to configure the names and attributes of the ATL object. The names are the names of the CoClass, the name of the interface, the Prog ID for the Verb in the registry and the C++ class that implements the COM object.



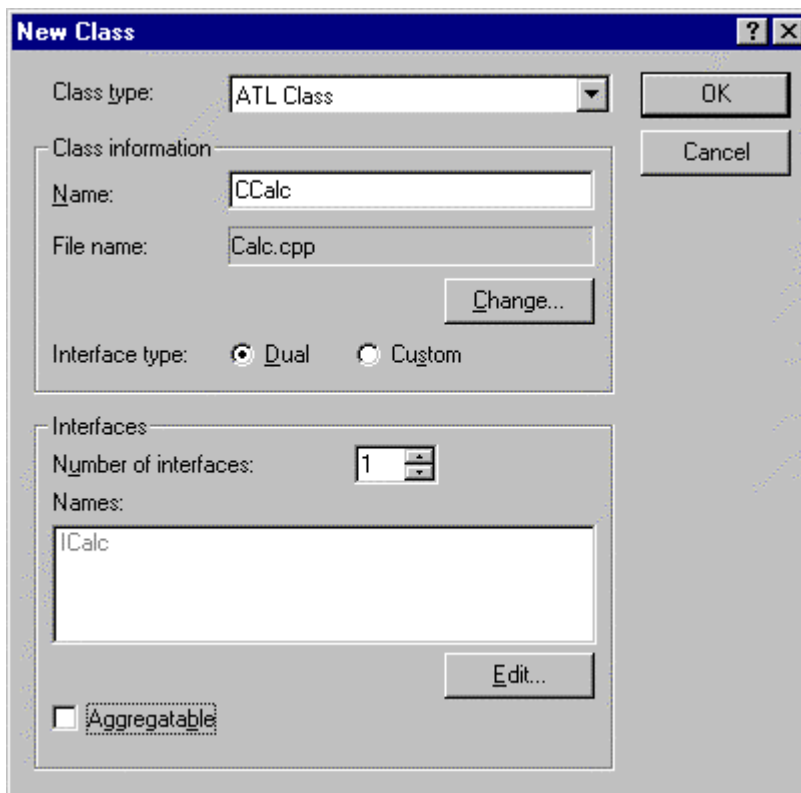
On the Attributes tab you can set the kinds of Threading models, the interface type and Aggregation. We here only changes from "Dual" to "Custom" and set Aggregation to "No"



As we will see later these choices are not the best but we will at that point see what we can do to fix it.

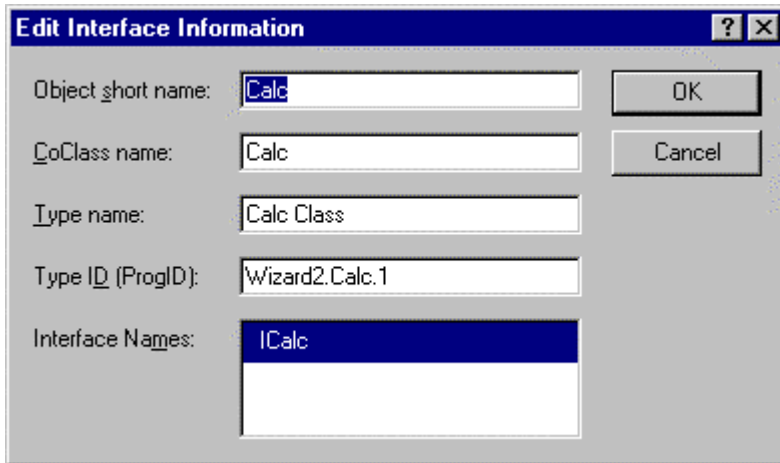
Wizard Way 2

If you instead when you right click on the mouse chooses **New Class...** the following appears:



As you can see you can add more than one interface in the beginning by using this Wizard. The strength of first wizard is that it can create a lot of different kinds of COM objects, while this is good for Automation objects only.

By pressing the **Edit** button you can edit the same names as you could on the **Names** tab on the **ATL Object Wizard Properties**, like:



The Wizard Generated Code

The big question is now: What did the Wizards do for you? In this case they would create exactly the same code. Each of the wizards does the following:

It adds

an entry to the object map (Deep_thought.cpp) that connects the CLSID (class ID) to the C++ class CCalc, like:

```
BEGIN_OBJECT_MAP(ObjectMap)
OBJECT_ENTRY(CLSID_Calc, CCalc)
END_OBJECT_MAP()
```

two files calc.h and calc.cpp to the project that contains the definition and implementation of the C++ object that implements the COM object. More about that later.

to the IDL-file an empty interface ICalc that inherits from IUnknown and a CoClass Calc in the Library DEEP_THOUGHTLib that implements

```
ICalc:
[
  uuid(6FB6D32F-9E06-11D1-AE78-0020AF72F3D6),
  helpstring("ICalc Interface"),
  pointer_default(unique)
]
interface ICalc : IUnknown
{
};
[
  uuid(6FB6D322-9E06-11D1-AE78-0020AF72F3D6),
  version(1.0),
  helpstring("Deep Thought 1.0 Type Library")
]
library DEEP_THOUGHTLib
{
  importlib("stdole32.tlb");
  importlib("stdole2.tlb");
```

```
[
uuid(6FB6D330-9E06-11D1-AE78-0020AF72F3D6),
helpstring("Calc Class")
]
coclass Calc
{
[default] interface ICalc;
};
};
```

The GUIDs for the interface and the CoClass are generated automatically.

a resource file called "calc.rgs". This file contains the registry-code so the component can self-registry. This code is awful to implement by hand. It looks like:

```
HKCR
{
Calc.Calc.1 = s 'Calc Class'
{
CLSID = s '{6FB6D330-9E06-11D1-AE78-0020AF72F3D6}'
}
Calc.Calc = s 'Calc Class'
{
CurVer = s 'Calc.Calc.1'
}
NoRemove CLSID
{
ForceRemove {6FB6D330-9E06-11D1-AE78-0020AF72F3D6} = s 'Calc
Class'
{
ProgID = s 'Calc.Calc.1'
VersionIndependentProgID = s 'Calc.Calc'
InprocServer32 = s '%MODULE%'
}
val ThreadingModel = s 'Apartment'
}
}
}
```

Changing the CoClass GUID

This is made in two stages:

Change the GUID in the idl-file for the class (the first uuid above the coclass statement).

Change the GUIDs in the Registry-file (Calc.rgs). This has to be done at least two places.

Calc.h and Calc.cpp

The file Calc.cpp contains some of the C++ implementation of the COM object
The file Calc.h contains the definition of the C++ class (CCalc) that implements the COM class Calc.

```
class ATL_NO_VTABLE CCalc :
```

```
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CCalc, &CLSID_Calc>,
public ICalc
{
public:
CCalc()
{
}
DECLARE_REGISTRY_RESOURCEID(IDR_CALC)
BEGIN_COM_MAP(CCalc)
COM_INTERFACE_ENTRY(ICalc)
END_COM_MAP()
// ICalc
public:
};
```

As seen the class multiple inherits from the classes `ComObjectRootEx`, `CComCoClass`, `ICalc`.

"`CComObjectRootEx` handles object reference count management for both nonaggregated and aggregated objects..." [VC++ reference manual], meaning it implements the reference count (more about aggregation later).

"`CComCoClass` provides methods for retrieving an object's CLSID and setting error information. Any class object that can be created externally should be derived from `CComCoClass`. `CComCoClass` also defines the default class factory and aggregation model for your object..."[VC++ reference manual].

The `ICalc` class is the abstract class as defined in "Deep_Thought.h".

ATL Macros

The `CCalc` has three different macro sections. These are `ATL_NO_VTABLE`, `DECLARE_REGISTRY_RESOURCEID`, and `COM_MAP`. Taking them from the beginning:

ATL_NO_VTABLE: This is an macro that uses an VC++ 5.0 compiler optimisation (`__declspec(novtable)`) so the linker can remove the vtable from the objects. If you are unsure whether you should use the `__declspec(novtable)` attribute, you can remove the `ATL_NO_VTABLE` macro from any class definition or you globally disable it by specifying `#define _ATL_DISABLE_NO_VTABLE` in `stdafx.h`, before all other ATL header files are included.

DECLARE_REGISTRY_RESOURCEID: This macro connects the `CCalc` class to the registry file "Calc.rgs" using the resource ID for this file.

COM_MAP: The COM map is the mechanism that exposes interfaces on an object to a client through `QueryInterface`. It contains of at least two macros `BEGIN_COM_MAP([class name])` (here `BEGIN_COM_MAP(CCalc)`) and `END_COM_MAP()`. Between these macros you can expose different interfaces by using the macro `COM_INTERFACE_ENTRY([Interface name])` (here

COM_INTERFACE_ENTRY(ICalc)). Other interfaces entry types can be defined here, like tear off interfaces (using COM_INTERFACE_ENTRY_TEAR_OFF). For each interface you add to the coclass you have to add entry here

Adding methods to the COM class

You now have an empty COM object (ActiveX control). It only contains IUnknown specific methods but no custom method (yet).

Adding methods to the COM object using a Wizard

If you in the class view opens CCalc and right clicks on the interface symbol (O--) and select method you are now ready to insert a method. The HRESULT type is the default type. Use this as often as possible because you give your client information about **Error! Not a valid filename.**

The method-Wizard generated code

The wizard does the following to the idl-, calc.h-, and calc.cpp-file:

In the IDL file it add the method definition to the ICalc interface:

```
interface ICalc : Iunknown
{
[helpstring("method Subtract")] HRESULT Subtract([in] BSTR n1,
[in] BSTR n2, [out, retval] BSTR* n3);
};
```

In the definition for the COM-object implementation class CCalc in calc.h the definition for the method is added:

```
// ICalc
public:
STDMETHOD(Divide)
(/*[in]*/ BSTR n1,
/*[in]*/ BSTR n2,
/*[out, retval]*/ BSTR* n3);
```

The STDMETHOD macro means:

```
#define STDMETHOD(method)virtual \
HRESULT STDMETHODCALLTYPE method
```

And the macro STDMETHODCALLTYPE means:

```
#define STDMETHODCALLTYPE __export __stdcall
```

So all in all the method definition becomes:

```
virtual HRESULT __export __stdcall Divide
(/*[in]*/ BSTR n1,
/*[in]*/ BSTR n2,
/*[out, retval]*/ BSTR* n3);
```

This is what we would expect a method to be when it is virtual and has to be exported from a DLL.

In the implementation for COM-object implementation class CCalc in calc.cpp the implementation for the method is added:

```
STDMETHODIMP CCalc::Divide(BSTR n1, BSTR n2, BSTR * n3)
{
// TODO: Add your implementation code here
return S_OK;
}
```

The STDMETHODIMP macro expands to HRESULT __export __stdcall

All these macros can be found in `..\VC\INCLUDE\BASETYPES.H` and `..\VC\INCLUDE\OBJBASE.H`.

You can now easily add the rest of the methods. The source code shows how to go from **BSTR** to **long** and back.

Compiling the component

When you compile the project from the IDE the makefile first creates a typelibrary from the IDL-file using the MIDL compiler (Microsoft IDL compiler). If you want this tlb-file to be added to the DLL you have to add as an resourcefile to the .. , like:

....

The MIDL compiler also generates a file called "[projectname].i.c" (in this case "Deep_thought.i.c"). This file contains the definitions of the GUID so they can be used in either C or C++.

It also generates a file called "[project name].h" (in this case "Deep_thought.h") which contains the abstract definitions of the C++ classes that implements the COM-objects interfaces. This file can be used either by C through creation of a Vtable or from C++ through the creation of an abstract superclass (abstract baseclass). In this case it will contain a class ICalc that defines the interface ICalc for the coclass Calc.

The default setting for the target (the DLL) is DEBUG. In the **Build | Set Active Configuration** you can change the target type. These other types are Release Minimal Size, Release Minimal Dependency, and Unicoded versions of all three.

Debug

.....

Minimal Size

This configuration creates a very small component. The problem about both the release configurations are that they both expect you not to use the C-Runtime library because the components have to be so small as possible. Because our component uses strings the component wont link. This can be fixed by opening the **Project | Settings | C/C++ | Pre-processor** and removing the **ATL_MIN_CRT** from the pre-processor definitions because this symbol stops ATL from using C Runtime Library. The same place contains the symbol **_ATL_DLL**. This symbol causes a dynamic link with the utility functions in the ATL.DLL. This is a very small DLL that has to be distributed with the minimal-size-component. It contains utility functions and the Component Registrar. These functions are:

Minimal Dependency

This one is self-contained. It statically links to the methods (functions) in ATL.DLL. This means the registrar is contained in the DLL.

All sizes are in Kbytes

Deep_Thought (phase1.1 using BSTR) Debug Release Min Size Release Min. Dependency Debug (Unicode) Release Min Size (Unicode) Release Min. Dep. (Unicode)	Component	DLL size	Atl.DLL size	TLB-size	Distr. Size
		212		1.95	
		65	18.5	1.95	
		72		1.95	
		219		1.95	
		65	18.5	1.95	
		72		1.95	

2.1.2 Phase 1.2: Changing the interface

In this phase we have to change the following : Method names, parameter names, and parameter types, as (-> means *substituted by*):

BSTR -> long
Dad -> Add
(n1,n2) -> (v1,v2)
n3 -> res

There are no wizard that can do that for you. It has to be done by hand but that is fairly easy too. The files calc.idl, calc.h, and calc.cpp are affected:

calc.idl: The interface ICalc are changed so it conforms to the wanted using a text editor

calc.h: The types in the definition are changed by hand from (with Divide as example):

From:
STDMETHOD(Divide)
(/*[in]*/ BSTR n1,
/*[in]*/ BSTR n2,
/*[out, retval]*/ BSTR* n3);

To:
STDMETHOD(Divide)
(/*[in]*/ long v1,
/*[in]*/ long v2,
/*[out, retval]*/ long* res);

calc.cpp: The types in the header are changed by hand from (with Divide as example):

From:
STDMETHODIMP CCalc::Divide(BSTR n1, BSTR n2, BSTR * n3)
{
// TODO: Add your implementation code here
_bstr_t val1(n1);
_bstr_t val2(n2);
long result = m_mother.mul(atol(val1),atol(val2));
wchar_t buffer[20];
_itow(result,buffer,10);
*n3 = W2BSTR(buffer);
return S_OK;
}

```
To:
STDMETHODIMP CCalc::Divide(long v1, long v2, long* res)
{
// TODO: Add your implementation code here
*res = m_mother.mul(v1,v2);
return S_OK;
}
```

As seen the code is simpler and more beautiful than in phase 1.1. Because we no longer uses strings (BSTR) we no longer need the C-Runtime library. We then get the following size-table:

All sizes are in Kbytes

Deep_Thought (phase1.1 using BSTR)	Component	DLL size	Atl.DLL size	TLB-size	Distr. Size
Debug		203		1.95	
Release Min Size		12	18.5	1.95	
Release Min. Dependency		19			
Debug (Unicode)		210		1.95	
Release Min Size (Unicode)		12	18.5	1.95	
Release Min. Dep. (Unicode)		19			

As you can see excluding the C-Runtime Library makes the controls very small.

2.1.3 Phase 1.3: Adding a method to an existing interface

In this phase we have to add a method to the ICalc interface.

Luckily there are a wizard that can do the hard work for us. We only do as we did in **Adding methods to the COM class in Phase 1.1.**

We then get the following size-table: All sizes are in Kbytes

Deep_Thought (phase1.1 using BSTR)	Component	DLL size	Atl.DLL size	TLB-size	Distr. Size
Debug		203		2.06	
Release Min Size		12	18.5	2.06	
Release Min. Dependency		19			

Luckily it does not add much to the size of the component to add a method

2.2 Phase 2

2.2.1 Phase 2.1: Adding an interface to the component

In this phase we add a new custom interface (IConv) to an existing component. There is no wizard for adding an interface to a component. We shall now see how an interface can be manually added to the component.

Step 1: IDL
Step 2: Calc.h
Step 3: Calc.cpp

2.3 Phase 3

2.3.1 Phase 3.1

In this phase we examine the tools ability to create DLL servers with more than one component.

We have to add a coclass that has the IConv interface. This is fairly easy done with the ATL component and method wizards as in phase 1.1.
All the idl-code is still in the same file.

2.3.2 Phase 3.2

In this phase we remove the component as we created in phase 3.1.
We simply reverse the code as described in phase 1.1. This include removing:

- The interface definition from the idl-file.
- The coclass from the idl-file.
- The CConv from the object-map in Deep_Thought.cpp.
- Removing the Conv.cpp from the project.
- Removing the registry script.

2.4 Phase 4: Aggregation

2.4.1 Phase 4.1: Aggregation using delegation

In this phase we use an already created component. It will be contained inside a new component (...). The new component has a new interface ICalc2 and should implement the old ICalc. The implementation of the old interface ICalc should be taken from an old component.

As in phase 2.1 we have two different interfaces to the same COM object: ICalc and ICalc2. The tricky part of this way to implement aggregation through delegation is the lifetime of the component. When using ATL this can be done in the virtual pseudo constructors FinalConstruct and FinalRelease. These are called right after construction and right before destruction where all states should be intact. The aggregated component that has the implementation of ICalc is contained in the outer component through a

Direct-To-Com (DTC) smart pointer. The type of the pointer is gained through the **#import** statement. This statement imports the type library for the inner component. We use the type library for the component in phase 2.1, like:

```
#import      "..\..\phase2_1\Deep_Thought\Deep_Thought.tlb"  
rename_namespace("old_Deep_Thought")
```

This statement parses the typelibrary on compiletime and creates a temporary files in the Debug (or Release..) directory. These files contain the definitions for the smartpointers through the macro ..

It also renames the namespace defined in the typelibrary due to the fact that the namespace for this component has the same name.

We can now define a pointer to the inner component in the outer component by defining a protected member variable, like:

```
old_Deep_Thought::ICalcPtr m_pCalc;
```

We can initiate and delete the component through **FinalConstruct** and **FinalRelease**, like:

```
HRESULT FinalConstruct( )  
{  
    return  
    m_pCalc.CreateInstance(__uuidof(old_Deep_Thought::Calc));  
}  
void FinalRelease( )  
{  
    m_pCalc.Release();  
}
```

The methods for the interface are implemented in the same manner as in phase 2.1. We add the interface to the IDL file. We let the implementation class inherit from ICalc2 (the abstract class generated by the MIDL compiler) and then we add the interface to the COM map, like:

```
COM_INTERFACE_ENTRY(ICalc2)
```

The methods are now defined as usual, like:

```
public:  
STDMETHOD(Modulus)(/*[in]*/ long v1, /*[in]*/ long v2,  
/*[out, retval]*/ long* res);  
STDMETHOD(Divide)(/*[in]*/ long v1, /*[in]*/ long v2,  
/*[out, retval]*/ long* res);  
...
```

and implemented by delegating method call to the inner component, like:

```
STDMETHODIMP CCalcDel::Divide(long v1, long v2, long* res)  
{  
    // TODO: Add your implementation code here
```

```
*res = m_pCalc->Divide(v1,v2);  
return S_OK;  
}  
STDMETHODIMP CCalcDel::Modulus(long v1, long v2, long *  
res)  
{  
// TODO: Add your implementation code here  
*res = m_pCalc->Modulus(v1,v2);  
return S_OK;  
}
```

2.4.2 Phase 4.2 Aggregation through interface aggregation

Aggregation of COM objects is a tricky problem. This has been pretty elegantly solved in ATL. When defining the components we can define the components to be one of three things

Always aggregatable (default in ATL)

Never aggregatable

Only aggregatable

in the sense of COM aggregation. When calling the COM method CoCreateInstance that creates the COM object one can add a parameter that tells whether or not the created component is aggregated inside another COM object. This is done through an interface pointer. If this pointer is NULL the component is not aggregated. If it is a pointer to a real interface it is aggregated. This interface pointer is used when QueryInterface is called on the inner (aggregated) components interface to delegate it to the outer QueryInterface, due to the fact of object ...

In the case of calling CoCreateInstance with the different kinds of declarations and outer interface pointers it will return different values. These are listed in the following table:

	ATL macro	pOut = NULL	pOut != NULL
Always	DECLARE_AGGREGATABLE	S_OK	S_OK
Never	DECLARE_NOT_AGGREGATABLE	S_OK	CLASS_E_NOAGGREGATIC
Only	DECLARE_ONLY_AGGREGATABLE	E_FAIL	S_OK

E

The superclass for our ActiveX control defines by default the component to be aggregatable. If you want something else you add the macros by hand. Both wizards for adding ATL COM classes gives you the choice of choosing the way of aggregation as seen in phase 1.1. The first wizard let you choose between all three alternatives. The second only lets you choose between the first two. If you make the wrong choice it is fairly easy to change the decision.

Offering the inner interface to the user

How can we offer the interface of the inner COM object to the user of the outer component without doing too much programming.

Contrary to what was the case in phase 4.1 we don't let this component inherit from the ICalc interface. We implement the delegation of the QueryInterface method by using the macro **COM_INTERFACE_ENTRY_AGGREGATE**, like:

```
BEGIN_COM_MAP(CCalcAggr)
COM_INTERFACE_ENTRY(ICalc2)
COM_INTERFACE_ENTRY_AGGREGATE(IID_ICalc, m_pUnkAgg.p)
END_COM_MAP()
```

This macro attaches an inner interface pointer m_pUnkAgg to the interface ICalc. When doing a QueryInterface on the outer component, this call is delegated through m_pUnkAgg to the QueryInterface of the inner component. This interface pointer is defined as a member variable in the outer class, like:

```
CComPtr<IUnknown> m_pUnkAgg;
```

What is left is construction and destruction of the inner COM object. This happens as usual in FinalConstruct and FinalRelease (and has to be hand-coded), like:

```
HRESULT CCalcAggr::FinalConstruct( )
{
    IUnknown* pUnkOuter = GetControllingUnknown();
    HRESULT hRes = CoCreateInstance(
        __uuidof(old_Deep_Thought::Calc),
        pUnkOuter,
        CLSCTX_INPROC_SERVER,
        IID_IUnknown,
        (void**)&m_pUnkAgg);
    if (hRes != S_OK)
        return hRes;
    return S_OK;
}

void CCalcAggr::FinalRelease( )
{
    m_pUnkAgg.Release();
}
```

Because our newly created component CalcAggr can also be aggregated itself we add the statement

```
IUnknown* pUnkOuter = GetControllingUnknown();
```

to FinalConstruct. This pointer is used in CoCreateInstance so we always use a pointer to the outer-most interface in the aggregation hierarchy. For being able to use this method (GetControllingUnknown) in FinalConstruct we have to define the macro **DECLARE_GET_CONTROLLING_UNKNOWN()** in the definition of the CCalcAggr class, like:

```
DECLARE_REGISTRY_RESOURCEID(IDR_CALCAGGR)

DECLARE_GET_CONTROLLING_UNKNOWN()
```

```
BEGIN_COM_MAP(CCalcAggr)  
COM_INTERFACE_ENTRY(ICalc2)  
COM_INTERFACE_ENTRY_AGGREGATE(IID_ICalc, m_pUnkAgg.p)  
END_COM_MAP()
```

So by doing almost nothing the inner component and its interface are presented to the user of the component (pretty neat, don't you think so?)

2.5 Phase 5: The Future Deep_Thought Server

By using the techniques as described in phase 1 to 4 it is pretty easy to create a server that offers the functionality as defined in the description.

2.6 Phase 6: A Client with a graphical interface

Not implemented in Visual C++/ATL!

2.7 Phase 7: Implementing Deep_Thought through Dispatch interfaces

In the preceding phase we have seen how to implement simple ActiveX controls with custom interfaces. When using late binding as in some script languages the client of the component has to use the IDispatch interface to use the methods of the component. This is also called **Automation**. In ATL the Dispatch interface can be exposed through at least two mechanisms. The first is the disp-interface and the second is the dual interface.

Disp interface

In the disp interface the methods of the component are only exposed through the Dispatch interface. You cannot access the custom interface in any way. Actually there are no vtable entries for the COM objects custom methods. In the IDL file the interface has the tack **dispinterface**.

Dual Interfaces

The dual interface exposes the methods of the interface. This includes both the custom and the dispatch interface. In the IDL file this kind of interface is has the tack **dual**.

Dispatch interface in ATL

These two usages of the dispatch interface can of cause be made by hand in ATL through different macros. The wizards only supports dispatch interfaces through dual interfaces. In phase 1.1 we could choose custom or dual interface when using the wizards. The wizards do not leave any room for defining a pure dispinterface. In the following section we show how to make both the dual and disp interface by hand.

Dual interfaces in ATL

The interface in the IDL file has to inherit from Dispatch, like:

```
[
  uuid(6FB6D32F-9E06-11D1-AE78-0020AF72F3D6),
  dual,
  helpstring("ICalc Interface"),
  pointer_default(unique)
]
interface ICalc : IDispatch
{
  [helpstring("method Add")] HRESULT Add([in] long v1, [in]
  long v2, [out, retval] long* res);
  [helpstring("method Subtract")] HRESULT Subtract([in] long
  v1, [in] long v2, [out, retval] long* res);
  [helpstring("method Multiply")] HRESULT Multiply([in] long
  v1, [in] long v2, [out, retval] long* res);
  [helpstring("method Divide")] HRESULT Divide([in] long v1,
  [in] long v2, [out, retval] long* res);
  [helpstring("method Modulus")] HRESULT Modulus([in] long
  v1, [in] long v2, [out, retval] long* res);
};
[
  uuid(08407980-9e15-11d1-ae78-0020af72f3d6),
  dual,
  helpstring("ICalc2 Interface"),
  pointer_default(unique)
]
interface ICalc2 : IDispatch
{
  [helpstring("method Power")] HRESULT Power([in] long v,
  [out, retval] long* res);
};
```

The implementation of the methods is similar to phase 5 except for two things:

The CCalc no longer inherits directly from ICalc and ICalc2 but uses the macro CComDualImpl to inherit from the Dispatch interface and implement the dual interface, like:

```
class ATL_NO_VTABLE CCalc :
  public CComDualImpl<ICalc, &IID_ICalc, &LIBID_DEEP_THOUGHTLib>,
  public CComDualImpl<ICalc2, &IID_ICalc2,
  &LIBID_DEEP_THOUGHTLib>,
  public CComObjectRoot,
  public CComCoClass<CCalc, &CLSID_Calc>
{
  public:
  .....
```

As seen this class inherits from two different dispatch interfaces. This introduces a new problem. How do we resolve ambiguities between which branch to select the methods in the Dispatch interface from. This is a general problem in C++ when talking about multiple inheritance. In ATL this is solved through the COM map entry **COM_INTERFACE_ENTRY2**. The arguments to this macro tells which branch to use. The map now looks like:

```
// 2 for ambiguities
```

```
BEGIN_COM_MAP(CCalc)  
COM_INTERFACE_ENTRY2(IDispatch, ICalc)  
COM_INTERFACE_ENTRY(ICalc)  
COM_INTERFACE_ENTRY(ICalc2)  
END_COM_MAP()
```

In this case we choose to use the methods in the ICalc branch

When there is only one interface involved as for the IConv class we don't have to use the COM_INTERFACE_ENTRY2 macro, but only the CComDualImpl macro in the inheritance list for the object.

Pure Dispinterfaces in ATL

.....