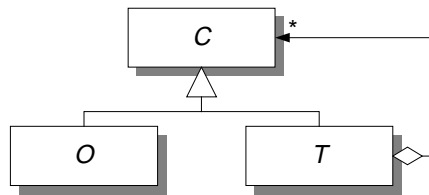


*Development of the Pilot Application in
Visual J++
COT/3-13-V1.0*



Centre for Object Technology

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

Author(s): Morten Grouleff, Aarhus University

Status: Final

Publication: Public

Summary:

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

© Copyright 1998 Aarhus 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.....	8
2.1.3 Phase 1.3 - Add a member to an existing interface	9
2.2 PHASE 2	9
2.2.1 Phase 2.1 - Add an interface to Deep_Thought's Calc component	9
2.2.2 Phase 2.2 - Remove an interface from Deep_Thought's Calc component.....	10
2.3 PHASE 3	10
2.3.1 Phase 3.1 - Add a component to Deep_Thought	10
2.4 PHASE 4	11
2.4.1 Phase 4.1 - Delegation	11
2.4.2 Phase 4.2 - Aggregation.....	11
2.5 PHASE 5 - CREATE THE DEEP_THOUGHT SERVER	11

1. Introduction

This document describes how Visual J++ 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

This document describes the process of implementing the pilot application in java, specifically using Microsoft Visual J++ version v1.1. (and Visual C++ v5.0 when needed).

2.1 Phase 1

2.1.1 Phase 1.1 - Create a component with an arithmetic interface

The Visual J++ way to proceed is to type in the following in a new project:

```
interface ICalc
{
    public abstract char Dad(char n1);
    public abstract char Subtract(char n1, char n2);
    public abstract char Multiply(char n1, char n2);
    public abstract char Divide(char n1, char n2);
}
```

```
class calcserver implements ICalc
{
    public char Dad(char n1)
    {
        return n1;
    }
    public char Subtract(char n1, char n2)
    {
        return n1;
    }
    public char Multiply(char n1, char n2)
    {
        return n1;
    }
    public char Divide(char n1, char n2)
    {
        return n1;
    }
}
```

```
}  
}
```

Creating a typelibrary and making a COM component out of this bit of code is done by using the "ActiveX Wizard for Java" in the Tools menu, and then doing as you're told. You have the choice of specifying your own CLSID, whether it should have a dual or dispinterface, and whether you want the IDL file to be compiled into a typelibrary. It is also possible to specify an IDL file. I think this can be used to tweak the type-mapping between Java and IDL types, but I haven't tried it yet. This is the IDL file generated.

```
[  
    uuid(ed99fb8c-9d68-11d1-a2e2-00c04fb954fb),  
    helpstring("calcserverLib Type Library"),  
    version(1.0)  
]  
library calcserverLib  
{  
    importlib("stdole32.tlb");  
  
    [  
        object,  
        uuid(ed99fb8b-9d68-11d1-a2e2-00c04fb954fb),  
        dual,  
        pointer_default(unique),  
        helpstring("Icalcserver Interface")  
    ]  
    interface Icalcserver : IDispatch  
    {  
        [ helpstring("Subtract Method") ]  
        HRESULT Subtract([in] char p1, [in] char p2, [out, retval]  
char * rtn);  
  
        [ helpstring("Dad Method") ]  
        HRESULT Dad([in] char p1, [out, retval] char * rtn);  
  
        [ helpstring("Multiply Method") ]  
        HRESULT Multiply([in] char p1, [in] char p2, [out, retval]  
char * rtn);  
  
        [ helpstring("Divide Method") ]  
        HRESULT Divide([in] char p1, [in] char p2, [out, retval] char  
* rtn);  
    }  
  
    [  
        uuid(ed99fb8a-9d68-11d1-a2e2-00c04fb954fb),  
        helpstring("Ccalcserver Object")  
    ]  
    coclass Ccalcserver  
    {  
        interface Icalcserver;  
    };  
};
```

A few interesting points about the above IDL:

- Even though I specified a interface to implement in the class, the Wizard decided to name the interface `Icalcserver` anyway.

- The type used for parameters is char. I believe that is a 8bit ASCII char in IDL. That is not a good match for the 16bit Unicode character, I have specified in Java.

The following client tests the rather stupid implementation above. It prints out "3" as expected.

```
import calcserver;

class client
{
    public static void main(String argv[])
    {
        calcserver cal = new calcserver();
        char a = cal.Subtract('3','4');
        System.out.println(a);
        try {
            Thread.sleep(5*1000);
        } catch (Exception e) {
        }
    }
}
```

Using an IDL file

A slightly different approach may be taken, if an IDL specification is given.

- Run `midl /I "c:\Progra-1\DevStu-1\VJ\include" filename.idl` on the IDL file.
- Run `javatlb filename.tlb`. This generates a Java Class file that describes the interfaces in the IDL file. This only works if the types used in the IDL file are in the list of supported types. All automation types are on that list, though. Also generated is a file `summary.txt` that contains a text version of the interfaces translated.
- Write your implementation of the interface using the `implements` Java construct to name the COM interfaces you implement.
- Export the class by using the "ActiveX Wizard for Java". This can be done with respect to an IDL file. By doing this, it is possible to specify the GUIDs used for CLSID and IID.

To test the ability to use multiple interfaces, I tried, using the IDL approach above, changing the project into the following:

```
public class calcserver
    implements handwritten.IcalcserverSimple,
    handwritten.IcalcserverAdvanced
{
    private static final String CLSID =
        "ed99fb8a-9d68-11d1-a2e2-00c04fb954fb";

    public char Dad(char n1)
    {
        return n1;
    }
    public char Subtract(char n1, char n2)
```

```
    {
        return n1;
    }
    public char Multiply(char n1, char n2)
    {
        return n2;
    }
    public char Divide(char n1, char n2)
    {
        return n2;
    }
}
```

and the client into

```
class client
{
    public static void main(String argv[])
    {
        handwritten.IcalcserverSimple is = null;
        handwritten.IcalcserverAdvanced ia = null;

        calcserver cal = new calcserver();
        char a = cal.Subtract('3','4');
        System.out.println(a);

        is = (handwritten.IcalcserverSimple) cal;
        System.out.println(is.Subtract('3','4'));

        ia = (handwritten.IcalcserverAdvanced) cal;
        System.out.println(ia.Multiply('3','4'));

        try {
            Thread.sleep(5*1000);
        } catch (Exception e) {
        }
    }
}
```

But this does not really use COM at all! It cheats, as it knows that calcserver is a java-object. To force it to use COM the following approach must be taken:

1. Translate the IDL-file describing the interface ICalc into a typelibrary, if that has not already been done.
2. run javatlb on the typelibrary. This generates a subdirectory in c:\winnt\java\trustlib that contains .class files with declarations corresponding to those in the IDL/TLB file and a summary.txt that textually describes this.
3. The client must then be rewritten to use the generated interfaces.

```
public class handwritten/Ccalcserver extends java.lang.Object
{
}
public interface handwritten/IcalcserverAdvanced extends com.ms.com.IUnknown
{
    public abstract char Multiply(char, char);
}
```

```
    public abstract char Divide(char, char);
}
public interface handwritten/IcalcserverSimple extends com.ms.com.IUnknown
{
    public abstract char Subtract(char, char);
    public abstract char Dad(char);
}

import handwritten.*;
class client
{
    public static void main(String argv[])
    {
        IcalcserverSimple is = (IcalcserverSimple) new Ccalcserver();
        IcalcserverAdvanced ia = (IcalcserverAdvanced) is;

        System.out.println(is.Subtract('3','4'));
        System.out.println(ia.Multiply('3','4'));

        try {
            Thread.sleep(5*1000);
        } catch (Exception e) {
        }
    }
}
```

As there is no java-class called Ccalcserver, I find it hard to see how it can cheat now. It must use COM, at least to get hold of the reference to the calcserver. After that, it might notice that both are implemented in java and short-circuit the calls for efficiency.

2.1.2 Phase 1.2 - Change the existing members of an interface

With what was learned in part one in mind, the following is written in a new java-file.

```
public class calcserver
{
    public int Add(int v1, int v2)
    {
        return v1+v2;
    }
    public int Subtract(int v1, int v2)
    {
        return v1-v2;
    }
    public int Multiply(int v1, int v2)
    {
        return v1*v2;
    }
    public int Divide(int v1, int v2)
    {
        return v1/v2;
    }
}
```

To export is as an ActiveX object, the header must be changed into the following as is suggested by the wizard.

```
public class calcserver
    implements calcserverlib.Icalcserver
{
    private static final String CLSID =
        "f4a97c3e-a3c1-11d1-a2e3-00c04fb954fb";
```

The client simply call all functions to test them.

```
class client
{
    public static void main(String argv[])
    {
        calcserver cal = new calcserver();
        int a = cal.Subtract(3,4);
        System.out.println(a);

        System.out.println(cal.Subtract(17,4));
        System.out.println(cal.Divide(17,4));
        System.out.println(cal.Multiply(17,4));

        try {
            Thread.sleep(5*1000);
        } catch (Exception e) {
        }
    }
}
```

2.1.3 Phase 1.3 - Add a member to an existing interface

There is really no work involved here. Just add the function to the calcserver implementation, re-export it, add a call of the function in the client, and test it. It works just fine...

2.2 Phase 2

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

First step is to implement the class. This was done by adding the following codesnippet to the previous implementation:

```
public int Hex2Dec(String v)
{
    return Integer.parseInt(v, 16);
```

```
}  
public String Dec2Hex(int v)  
{  
    return Integer.toString(v, 16);  
}
```

The next and last step is to run the "ActiveX wizard for Java". To get multiple interfaces in the resulting component, a IDL file must be used. **Error! Bookmark not defined.** specified in the **Error! Bookmark not defined.** specification was used. But that did not work! The problem is that the wizard insists on running MIDL in a /mktypelib203 compatibility mode. Moving the interface declarations into the library definition fixed that. This change does not change the semantics of the IDL-file for the current purpose. The typelibrary is generated automatically from the IDL file during the process described above. The (complete) client looks like this:

```
import deep2_1.*;  
class client  
{  
    public static void main(String argv[])  
    {  
        deep2_1.ICalc calc = (deep2_1.ICalc) new Calc();  
        IConv conv = (IConv) calc;  
        System.out.println(calc.Subtract(17,4));  
        System.out.println(calc.Divide(17,4));  
        System.out.println(calc.Modulus(17,4));  
        System.out.println(calc.Multiply(17,4));  
        System.out.println(conv.Dec2Hex(17));  
        System.out.println(conv.Hex2Dec("17"));  
  
        try {  
            Thread.sleep(5*1000);  
        } catch (Exception e) {  
        }  
    }  
}
```

Trying to call a function though the wrong interface, say Hex2Dec on calc, gives a compile-time error.

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

There is no tools-support here. Just removing the interface from the IDL file and recompiling will do the trick.

2.3 Phase 3

2.3.1 Phase 3.1 - Add a component to Deep_Thought

There is no concept of "server" in Java. The code is not translated into a dll but rather

into a .class file. There is no way to put two public classes into one classfile. On the other hand, all Java components may be considered part of the same DLL, msjava.dll.

2.4 Phase 4

2.4.1 Phase 4.1 - Delegation

There is no tool-support for delegation in J++ 1.1. In essence the new server just acts a client of the inner component. As this has to be implemented by hand, there is no new knowledge to gain here.

2.4.2 Phase 4.2 - Aggregation

Aggregation is directly supported by Java. The component to be aggregated *must* explicitly specify its aggregatability in its typelibrary. It is *not* possible to aggregate more than one object using java.

My experiments with aggregation has not worked yet. I will keep trying to track down the problem. Right now I get

```
ERROR: com.ms.com.ComFailException: (80004005h) Unspecified error
```

2.5 Phase 5 - Create the Deep_Thought server

The task at hand is a bit awkward for Java. Java components can only be represented as .class files, not as EXE or DLL files. The concept of ProgIDs does not appear to be directly supported by Java. By browsing the documentation I found that it is possible to associate a ProgID with a Java class by adding the following to the coclass statement:

```
[ custom(GUID_PROGID, "DAO.Dynaset") ]
```

Another option is to run `JavaReg` by hand and give it option `ProgID` to specify the ProgID.

Creating a out-of-process server is possible by using `JavaReg` as a surrogate server according to the manual. This has not been tested.

This phase is to be used during the exchange of components to test interoperability between the tools. Extra care must be taken to make sure all the details like GUIDs, ProgIDs, etc. make this possible. As Java cannot put two public classes in one file, I implemented all the interfaces in one class. The IDL file had to be modified slightly, as described in phase 2.1. Then the class header must be modified as requested by the wizard. But that is not enough! The wizard does not mention all the three interfaces in the implements line. The final code looks like this:

```
import deep5.*;

public class deep_thought
    implements deep5.ICalc, deep5.ICalc2,deep5.IConv
{
    private static final String CLSID =
        "97723080-9e15-11d1-ae78-0020af72f3d6";

    public int Modulus(int v1, int v2)
    {
        return v1 % v2;
    }

    ...

    public int Power(int v1)
    {
        return Multiply(v1,v1);
    }
    public int Hex2Dec(String v)
    {
        return Integer.parseInt(v, 16);
    }
    public String Dec2Hex(int v)
    {
        return Integer.toString(v, 16);
    }
}
```