



COM

Stephen Morris

TESSELLA SUPPORT SERVICES PLC

Issue V1.R1.M1

November 1999



COM - THE COMPONENT OBJECT MODEL

The Requirements

The object-oriented approach to software engineering decomposes a system into a set of interacting objects. Such an approach leads to software that is cheaper to build and easier to maintain. It is cheaper to build because the functionality of the objects is well defined, and it may be possible to buy-in object libraries, or re-use objects from earlier projects. Commonly, these objects are low-level, being compiled and linked directly into the application, but re-use also occurs with much larger pieces of software, mini-applications in their own right, which are termed “components”. The software is easier to maintain because changes have only localised effects. Providing an object or component displays correct behaviour, its internals are of no concern to other parts of the system (except, perhaps, how fast they run), and may be modified at will. This should be contrasted with some older, monolithic systems, where a small change in one area can have knock-on effects in other - often unpredictable - areas of the program.

Language is a limitation to achieving this goal. Although object-oriented Java and C++ support object re-use, many applications are written in other languages. To make full use of re-usable components, it should be possible to write them in one language and access them from another.

A second requirement is one of distribution. With the prevalence of networks, distributed applications are becoming more and more important. Many applications are now split into two parts, a client and a server. To a user, the detail is hidden; they use the client single machine and it communicates invisibly with the server. A more generalized form of this would have the application composed of a number of components, where each component may be located anywhere on the network. For complete generality, we also require the location of the components to be transparent to the application - remote components must be useable in the same way as local ones.

Finally, we would like to replace components dynamically. We may want to upgrade them without reissuing the entire application. A good example of this is Microsoft’s OLE (Object Linking and Embedding), where we may have an Excel spreadsheet embedded in a Word document. If we upgrade Excel, we would like to use this new version when we come to modify the spreadsheet. We do not necessarily want to upgrade Word to be able to do this. This sort of operation is

possible to a certain extent with dynamic-link libraries (DLL's). However, there are drawbacks, most notably that they provide functions, not objects. In addition, use of a DLL requires that it has a known name and is in a known location; if either of these change, the application will no longer work.

What is COM?

COM (Component Object Model) is Microsoft's answer. It is a specification of how to write software that meets the requirements listed above, together with a set of software libraries that facilitate the writing of compliant software. COM is now used extensively in Microsoft products, with OLE being one of the most notable uses. Distributed COM (DCOM) is an extension to COM that allows the linking together of components across a network.

As noted, COM is a Microsoft standard. A rival standard, CORBA (Common Object Request Broker Architecture) has been developed by the OMG (Object Management Group), a consortium of computing-involved companies (including, curiously enough, Microsoft). Since they address the same issues, there are many similarities between CORBA and COM, e.g. both use the Open Software Foundation's Interface Definition Language (IDL) to describe the component interfaces. In fact, the OMG have defined a mapping between CORBA and DCOM, and the two can be mixed. Regardless of how the standards evolve, COM - by virtue of Microsoft's dominance in the computing industry - will always be important. (More information about CORBA can be found in the Tessella Technical Supplement on the subject).

THE DETAILS

Having described the problems that COM is attempting to solve, we will now look at some of the technical aspects in detail.

Interfaces

The key idea behind object-oriented programming is one of interfaces. An object comprises a set of interfaces that define its behaviour. How that behaviour is implemented is not the concern of users of that object. Providing that the interfaces don't change, alterations to the internals of the object have no effect on the programs that use it. Although it is usual to equate an interface with a method on an object, it is more general to define an interface to be a collection of methods. As an example, an object representing an amphibious craft may well

have two interfaces, one defining the behaviour of the vehicle when on water, the other its behaviour on land. Each interface may have a number of methods, but together they completely define the behaviour in the chosen environment.

The same is true of COM. COM defines components (the equivalent of objects), which have a set of defined interfaces. A client program creates the COM object, then retrieves a handle to the interface it wants to use. Once it has this handle, it can use the services offered by the interface. At first sight, this seems unnecessarily complex; after all, in a language like C++, having got the object the interfaces (methods) can be used directly since they are defined in the class definition. Apart from the problem of exporting the definition to multiple languages, there is the question of encapsulation. If the layout of the component is defined in a header file which is used by the client, the client would have to be recompiled if the component were to change. That conflicts with the requirement of dynamic replacement, and is something that is avoided if the client queries the component for its interfaces at run time.

A further ramification of the definition of a component by its interface is that, once published, interfaces may never be changed. Although this seems restrictive, it is a very necessary requirement to allow the component to be used by clients. Remember that components must be independent of clients, and may be upgraded independently of them. This can only happen successfully if, to a client application, later versions of a component are guaranteed to look like earlier versions. If they do change, the component will appear to have changed, and may cease working with its clients. However, just because existing interfaces never change, it does not mean that later versions of a component cannot add additional interfaces. COM only guarantees upwards compatibility between client applications and components, not downwards.

Registration of COM Components

Given that the COM component can be anywhere - in a DLL or EXE file on the local machine, or even on another machine in the network, how can the application locate it? The key to this lies in the “Global Unique Identifier” or GUID¹. A GUID is a 128-bit number that identifies a COM constituent (interface, component etc.). The GUID is generated by the component author during the development process, with a program that combines the computer’s network address with the current time. As such, each GUID should be unique. (Admittedly, if two developers, each working on a computer with no network

card, generate a GUID there is a chance that the GUID's will be the same. However, since the GUID stores the time to a granularity of 100 nanoseconds, the chance of that happening is extremely small.)

Before using a COM component, the GUID is recorded on the local machine, the data being stored in the registry. The GUID of the component is published by the component developer, and applications look up the number in the registry to determine the component to which they must connect.

Independence of Location

One of the goals was to be able to access COM components no matter where they are located. This means that we need to include some component in the code that behaves exactly like the interface we want to use, but which communicates with the component wherever it is. Figure 1 shows how this is organised.

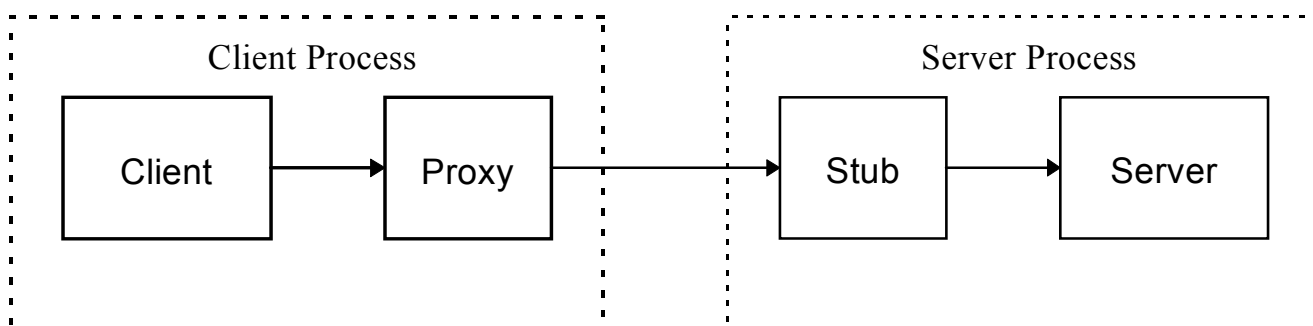


Figure 1. Communication between Client and Server

A definition of the interface offered by the server component is written in the Interface Definition Language (IDL). When processed by the MIDL compiler, it generates the code for two modules, the proxy and the stub. The proxy presents to the client the interface offered by the server. When the client makes a call to this interface, the proxy code marshals the data passed through the call, and sends it to the stub in the server process. The stub extracts the parameters from the marshalled data, and passes them to the corresponding interface of the server. When the call to the server returns, the data is passed back in the same way. Although figure 1 suggests that the client and the server must always be in

¹ This is also something that originates from the Open Software Foundation's Distributed Computing Environment, except there it is known as the "Universally Unique Identifier" or UUID. The term "UUID" persists in COM as a keyword in the Interface Definition Language File.

different processes, they can in fact be anywhere.

COM components can be in the same process, and often are, in the form of a DLL. But by using proxy and stub code, the location is invisible to the client. The server could be in the same address space, but a later version might move it to a different process, or even to a different machine. Wherever it is, the client uses the server in exactly the same way. Similarly, the server has no way of telling how it is being called.

The other important term in the description of client-server communication is that of “marshalling”. This describes the process whereby call parameters are passed between client and server. When a client calls the server within the same address space, there is no problem; both the client and the server can address the same memory, and the parameters passed to the stub by the client can be passed directly to the server. If the client and server are in different processes, or possibly on different machines, the call parameters must be copied to the target process in order to be used. Although this is fairly straightforward for simple data types, if a pointer to some data structures is a parameter, the entire data structure must be copied and the pointer adjusted to reflect the location of the structure in the server address space.

Although potentially very complicated, all this is taken care of by the MIDL compiler. The IDL file describes the interface, including the number and types of the parameters. MIDL generates the proxy and stub, which include the code needed for communication, together with that required to handle parameter copying. The proxy-stub code is then included as DLL's in both the client and server programs.

Component Lifetime and Reference-Counting

A frequent problem of programs that perform dynamic memory allocation is one of memory leaks, where allocated memory is never freed. This can become a problem if the program runs for a long time; the virtual memory allocated to the program can grow to a point where the system runs out of memory. The only thing that can be done is to shut the program down and restart.

A situation like this most frequently arises where there is confusion as to who is responsible for freeing the memory. For example, in the following code snippet:

```
void master(void)
{
    void subordinate(int *arg);
    :
    int *block = new int[200];
    subordinate(block)
    :
}
```

... who is responsible for freeing `block`: should `master` free it when `subordinate` returns, or should `subordinate` (or functions called by `subordinate`) perform the task? There is no right answer - it depends on what `subordinate` does.

When using COM components, we face similar problems. It is conceivable for components to be created by clients but never freed, so consuming virtual memory. To get round this, COM uses the technique of reference counting. When a client obtains an interface, a reference count within the interface is initialized to 1. If another reference is made to the interface (e.g. the pointer to the interface is copied), the reference count is incremented. When a pointer to an interface is destroyed or goes out of scope, the reference count is decremented, and when the count goes to zero, the interface destroys itself. Although the logical view is that each interface has its own reference count, in practice it may well be that it is the component that is reference counted. When the reference count falls to zero (i.e. no further use is being made of any of the component's interfaces), the component destroys itself.

Using a COM Component

The preceding sections have been concerned with the architecture and design of COM components. The following simple C++ example shows how a COM component is used. Here, we suppose that we have a COM component that provides an interface `IMatrix` to perform matrix-related operations. The example obtains a matrix, accesses the interface, and uses a method on the interface to invert the matrix.

```
#include <iostream.h>
#include <objbase.h>
#include "matrix.h"

int main(int argc, char **argv)
{
    // (1) Initialize COM library.

    CoInitialize(NULL);

    // (2) Create the component & get interface.

    IMatrix *pMatrix = 0;
    HRESULT hr = CoCreateInstance(
        CLSID_Component, // GUID of component.
        NULL, // Ignored in this example.
        CLSCTX_ALL, // Use any implementation.
        IID_MATRIX, // IMatrix interface GUID.
        reinterpret_cast<void **>(&pMatrix)
        // Interface pointer.
    );

    // (3) Check result and if OK use interface.

    if (SUCCEEDED(hr)) {

        // (4) Use interface.

        CMatrix aData, anInverse;
        GetData(aData);
        hr = pMatrix->Inverse(aData, anInverse);
        if (SUCCEEDED(hr)) {
            cout << "Matrix:\n"
                << aData
                << "Inverse:\n"
                << anInverse;
        }
    }
}
```

```
        // (5) Free up interface.

        PMatrix->Release();
    }

    // (6) Exit.

    CoUninitialize();
}
```

Admittedly this is a very simple example, but does illustrate some of the main points of using COM.

1. The client must initialize the COM library. Although COM is mostly a specification, it has supporting libraries and these require initializing.
2. The “meat” of the example. CLSID_Component is the GUID of the component holding the IMatrix interface, and IID_MATRIX the GUID of the interface itself. (In this example it is assumed for simplicity that both CLSID_Component and IID_MATRIX are defined in the “matrix.h” header file. More generally, CLSID_Component would be looked up in the registry, allowing the component to be upgraded without requiring a rebuild of the application.) The CLSCTX_ALL constant (defined in the COM-supplied header file objbase.h) tells CoCreateInstance that we don’t care where the component is; whether it is in-process (a DLL), in a different process (another EXE file on the same machine) or on a remote machine, we just want to use it (other constants allow the scope to be restricted). The pointer to the interface is returned through the pMatrix argument. CoCreateInstance is a general routine, so does not know the type of the pointer it is expected to return; it therefore expects a pointer to a void* (which has the unfortunate side effect of disabling some of the type-checking that the C++ compiler can perform). The reinterpret_cast statement casts the argument to the correct type.
3. After CoCreateInstance returns, the success status is checked. COM components and procedures can return alternate success statuses, so use of the COM-supplied SUCCEEDED (and corresponding FAILED) macro is recommended over a direct comparison with success codes

4. Having obtained the interface, we can now use it to call the Inverse() method (which in this example performs the matrix inversion). The matrix to be inverted (assumed to be described by a class Cmatrix defined in “matrix.h”) is read by some supporting function GetData(), and passed to Inverse() for processing. Inverse() returns the result through an argument, and a status through the function return value. The convention of returning a status through the function return value is primarily followed for components that are expected to be run across a network. Network connections can be broken, and COM requires some way of reporting the failure (something particularly relevant to this example; inverting large matrices can be a very compute-intensive operation, and it is quite feasible that an application may be designed to have these sorts of operations delegated to other, more powerful, machines.)
5. Having finished with the interface, we now release it. As there are no other references to it in this short example, releasing it will have the effect of deleting the component.
6. Finally, we rundown the COM library and exit.

Summary

In a short article such as this, it has only been possible to touch the surface of COM, and the reader is urged to consult specialist books on the subject for more detail. COM was developed to provide object-oriented language and location independent re-usable components. It has largely succeeded, although the standard is still evolving; Microsoft are developing COM+ to address remaining problems in COM. Being a Microsoft standard, COM will play a major part in the development of software in the Windows environment.

Bibliography

Inside COM - Microsoft's Component Object Model, Dale Rogerson (Microsoft Press, 1997, ISBN 1-57231-349-8)

Beginning ATL COM Programming, Richard Grimes, Alex Stockton, George Reilly, Julian Templeman (Wrox Press, 1998, ISBN 1-861000-11-1).

Tessella Support Services plc
Creating Software for Science and Engineering

Tessella's services range from feasibility studies, through system design, development, implementation and ongoing support. Our expertise includes:

Data Analysis Software
Data Capture
Simulation Software
Advanced Graphics
Systems Support
Database Applications

Other Technical Supplements available include:

- | | |
|---|--|
| <input type="checkbox"/> Archiving of Electronic Info | <input type="checkbox"/> Object Oriented Programming |
| <input type="checkbox"/> Active Server Pages | <input type="checkbox"/> Pocket PC |
| <input type="checkbox"/> Automated GUI Testing | <input type="checkbox"/> Portable GUI Development |
| <input type="checkbox"/> Bayesian Statistics | <input type="checkbox"/> Printer Technology Guide |
| <input type="checkbox"/> Beowulf Clusters | <input type="checkbox"/> Real Time Systems |
| <input type="checkbox"/> C++ | <input type="checkbox"/> Regression Testing |
| <input type="checkbox"/> Client-Server Technology | <input type="checkbox"/> Security and the Internet |
| <input type="checkbox"/> COM | <input type="checkbox"/> Simulation |
| <input type="checkbox"/> Computational Fluid Dynamics | <input type="checkbox"/> Soft Computing |
| <input type="checkbox"/> Computer Image Processing | <input type="checkbox"/> Software Design Methodologies |
| <input type="checkbox"/> Decision Support Systems | <input type="checkbox"/> Software Development Cycle |
| <input type="checkbox"/> Electronic Data Capture | <input type="checkbox"/> Software Documentation |
| <input type="checkbox"/> Electronic Lab Notebooks | <input type="checkbox"/> Software Portability |
| <input type="checkbox"/> Excel | <input type="checkbox"/> Software Re-engineering |
| <input type="checkbox"/> Extending the Life of Software | <input type="checkbox"/> Software Specification |
| <input type="checkbox"/> Federal Drug Administration | <input type="checkbox"/> SQL |
| <input type="checkbox"/> FORTRAN 90 | <input type="checkbox"/> UNIX Inter-Process Comms |
| <input type="checkbox"/> Grid Computing | <input type="checkbox"/> UNIX Systems Performance |
| <input type="checkbox"/> High Throughput Screening | <input type="checkbox"/> UNIX Workstations |
| <input type="checkbox"/> Instrumentation | <input type="checkbox"/> Visual Basic 6 |
| <input type="checkbox"/> Integrated Lab Systems | <input type="checkbox"/> WAP |
| <input type="checkbox"/> J2EE | <input type="checkbox"/> Web Services |
| <input type="checkbox"/> Java | <input type="checkbox"/> Windows 2000 Services |
| <input type="checkbox"/> Lims | <input type="checkbox"/> XML |
| <input type="checkbox"/> Linux | <input type="checkbox"/> X Windows |
| <input type="checkbox"/> Microsoft Net | |



INVESTOR IN PEOPLE

Tessella Support Services plc

3 Vineyard Chambers, Abingdon, Oxon, OX14 3PX, England

Tel: (+44) (0) 1235 555511 Fax: (+44) (0) 1235 553301

E-mail: info@tessella.com Web Address: <http://www.tessella.com>