C# and C++ Interoperability: High-level Roadmap

Daniel J. Duffy dduffy@datasim.nl
October 2009

Summary
In this short note we give an overview of some techniques for creating Windows-based applications that use a mix of C# and C++. In particular, we discuss using a software component written in one language by a component in another language. Some scenarios are shown in Figure 1:

1. Integration with native DLLs.
2. Using COM components (for example, Excel) from .NET using Runtime Callable Wrappers (RCW).
3. COM clients that call C# components using COM Callable Wrappers (CCW).
4. Calling ISO C++ code from C# using C++/CLI and wrapper classes.
5. Calling C# code from ISO C++ using C++/CLI and wrapper classes.

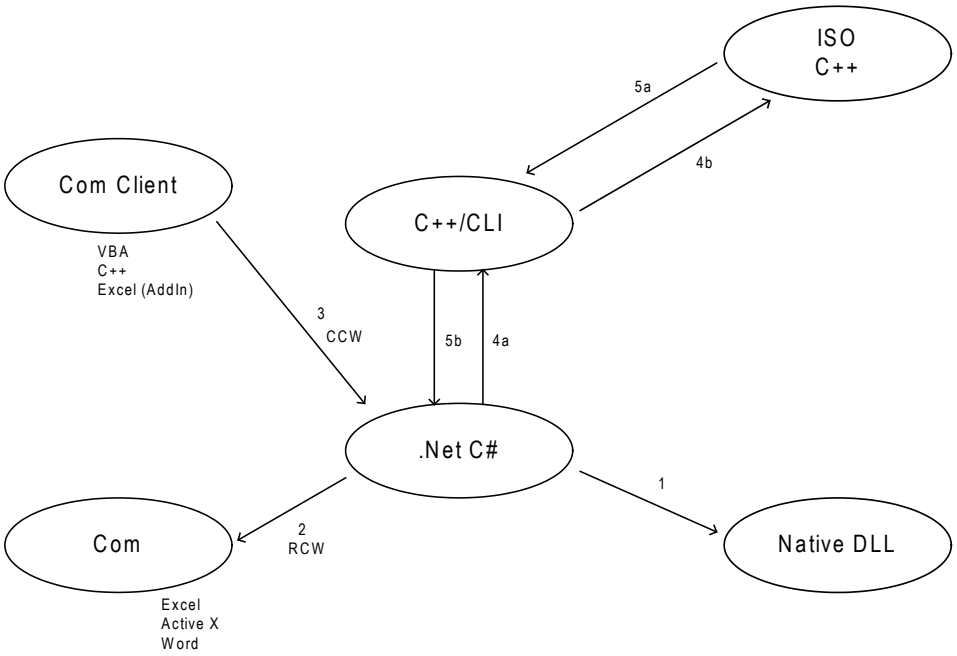

Figure 1 C#/C++ Interoperability Models

We now discuss each of these scenarios in some more detail. Full details are in the distance learning course.

1. Native DLL Integration
DLLs encapsulate unmanaged C code which is by definition non-object oriented. We may need to implement this scenario when we already have developed and tested C libraries and have placed them in DLLs. Some of the supported functionality is:
a. Calling a function defined in a DLL.
b. Marshalling of function input arguments and return values.
c. Marshalling classes and structs.

d. Callbacks from unmanaged code.
e. Simulating C unions: explicit layout and field offsets.

In this way we can reuse existing legacy C code in new applications.

2. Using COM Components from C# Clients

In this case we wish to use COM code (which is unmanaged) from C# client code. To this end, we create a .NET Runtime Callable Wrapper (RCW) for the COM component as shown in Figure 2.

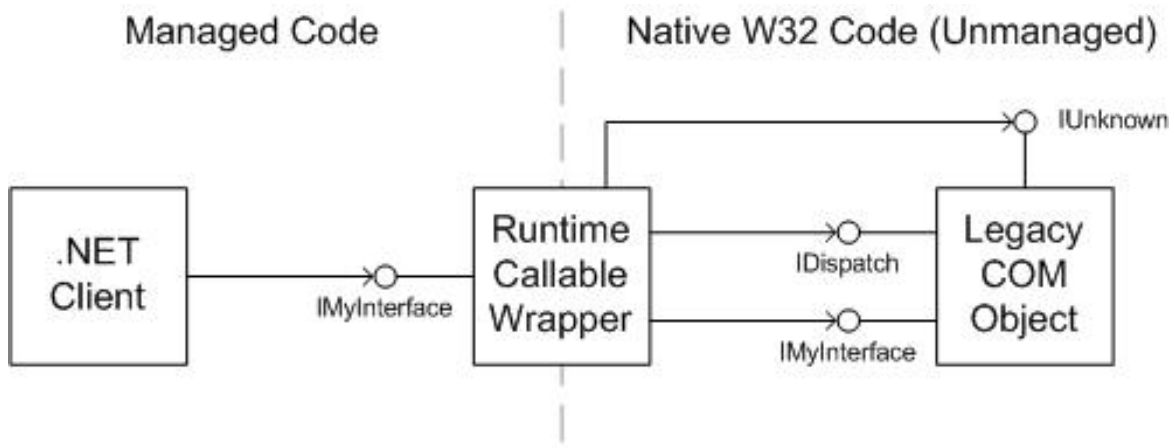

Figure 2 Runtime Callable Wrapper

Then the C# code interoperates with the wrapper component in a .NET environment. The RCW is in fact a proxy and it is generated automatically by Visual Studio after the corresponding COM component has been referenced.
An important example of this scenario is when we wish to use Excel from a C# client application. In this case we include the Excel component in our project whence the RCW will be generated during the build process. Another case is the use of ActiveX controls; many (e.g. WebBrowser) are COM objects.

3. Calling .NET Components from COM Clients

This scenario is the reverse of scenario 2; we now wish to use a .NET component from a COM client (for example, VBA, C++ or an Excel AddIn). As before, a so-called COM Callable Wrapper (CCW) is generated that can then communicate with the COM client, as shown in Figure 3. This is a proxy and it is automatically generated by the .NET runtime. The .NET component must be registered as a COM component.

4. C# Client calling C++ Code

The C++/CLI is Microsoft's proprietary managed C++ language. It can be likened to a bridge between the managed and unmanaged worlds. The two main issues are:
4a: C# component uses a C++/CLI component.
4b: C++/CLI component is a wrapper (using composition) for an ISO C++ class.

This process is easy to achieve thanks to the C++/CLI language.

5. C++ Client calling C# Component

In this case unmanaged C++ code uses C++/CLI code which subsequently calls C# code. For example, it is possible to add Windows capabilities to a C++ application by letting it communicate with a visual control or dialog in C++/CLI. In this way we 'beef up' C++ by allowing it to take advantage of the rich functionality in .NET.

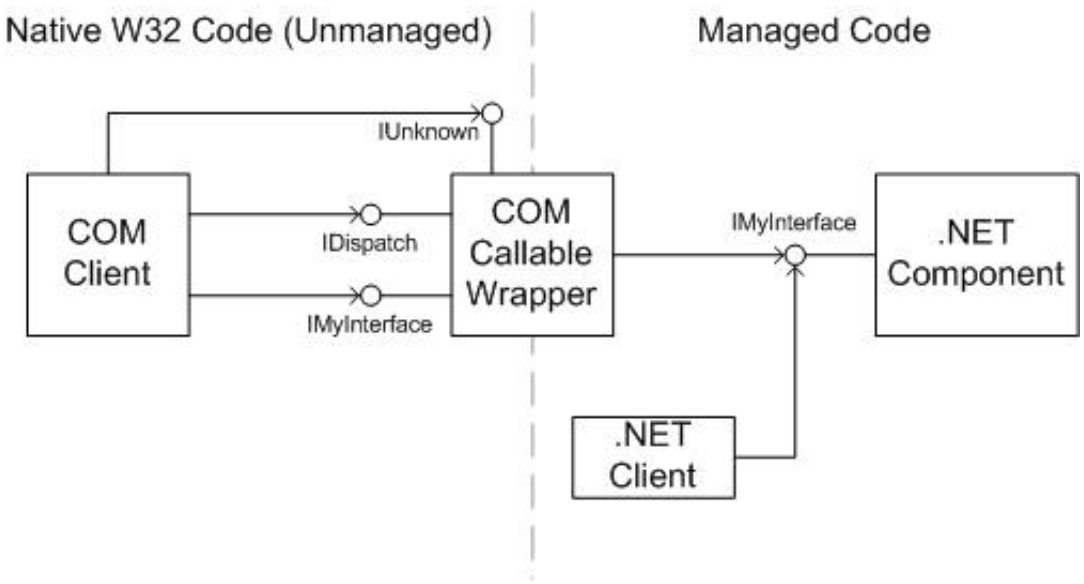

Figure 3 COM Callable Wrapper

6. Code Example

We include the code for a managed C++ class that functions as a wrapper for a native C++ class to give the reader a feeling for how ISO C++ and C++/CLI function together:

```
// ManagedWrapper.hpp
//
// C++/CLI wrapper class for a unmanaged C++ class

#pragma once

#include "NativeClass.hpp"

using namespace System;

// ManagedWrapper has similar interface as the native class that it wraps
public ref class ManagedWrapper
{
private:
   // Embedded native class
   NativeClass* m_nativeClass;

public:
   // Default constructor
   ManagedWrapper()
   {
       // Create native class instance
       m_nativeClass=new NativeClass();
   }

   // Constructor with data
   ManagedWrapper(int data)
   {
       // Create native class instance with data
       m_nativeClass=new NativeClass(data);
   }

   // Copy constructor
   ManagedWrapper(ManagedWrapper^ source)
   {
       // Copy native class instance
```

```
        m_nativeClass=new NativeClass(*source->m_nativeClass);
    }

    // Finaliser (called by the garbage collector or destructor)
    !ManagedWrapper()
    {
        // Delete the native class instance
        delete m_nativeClass;
    }

    // Destructor (Dispose)
    ~ManagedWrapper()
    {
        // Call finaliser
        this->!ManagedWrapper();
    }

    // Get- and set data as property
    property int Data
    {
        int get() { return m_nativeClass->GetData(); }
        void set(int data) { m_nativeClass->SetData(data); }
    }

    // Get- and set string as property
    property String^ Str
    {
        String^ get()
        {
              return gcnew String(m_nativeClass->GetString().c_str());

              // Alternative method
        }

        void set(String^ str)
        {
              // Convert to temporary char*
              // IntPtr chars= edited

              // Create STL string from char*
              m_nativeClass->SetString(std::string(pChars));

              // Delete temporary char*
              // edited
        }
    }
};
```