C# and COM Interop: Core concepts for Komax Developers

Overview

This page explains the essential concepts behind how .NET (and C#) interacts with COM components (like C++ COM components), which is critical for understanding what your code is doing under the hood.

What is COM?

COM (Component Object Model) is a Windows-native binary interface standard that:

- Defines how components expose functionality via interfaces
- Enables cross-language and cross-process communication
- Is still used in many Windows technologies (Office, DirectX, etc.)

Example: COM is a Microsoft technology that gives us the opportunity to have C# project (library for example) that use code written in C++ language and vice versa.

What is COM Interop?

COM Interop is a .NET feature that allows:

- C# to call native COM components (usually C++ DLLs)
- C# code to be exposed as a COM component used by C++ native COM components

Example: CSWIN nx software is written in C/C++ programming languages and uses COM technology and MFC framework. CSWIN nx uses as COM components many external COM components written in C#. So, this is a perfect example for software written in one language that uses (via COM technology) components written in another language/s.

What is a .tlb file?

A .tlb (Type Library) is a binary file that describes the interfaces, classes, enums, and structs that a COM component exposes (the metadata).

It’s essentially the IDL (Interface Definition Language) compiled - a form of metadata for COM.

It acts as a contract that contains:

- Interface names and methods
- CLSID and IID (GUIDs)
- Type signatures (parameters, return types)

In simple words it is a binary file that represents interface which helps the clients of the real code to know the classes, and methods that can invoke (in compile time).

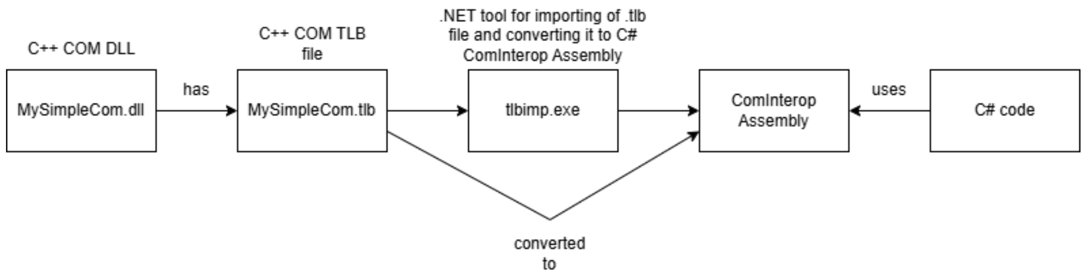
It is very important that the .tlb file is not used at runtime. For C# COM programmers the .tlb file is needed to generate so called ComInterop Assembly in .NET (next section). This ComInterop Assembly is necessary because the C# and .NET don’t understand directly .tlb files. They (the .tlb files) have to be converted to something understandable for .NET world.

What is a COMInterop DLL?

A ComInterop DLL is a .NET assembly (dll) that acts as a proxy library for the COM component. It's generated using *tlbimp.exe* or Visual Studio when you import a .tlb file to your C# project. Actually, when for example we import .tlb file in our C# project, under the hood is generated this ComInterop Assembly in our Release or Debug folder (depending on the configuration).

The purpose of COMInterop Assembly is:

- Provides .NET-friendly interfaces and classes that mirror the COM definitions
- Contains attributes like [Guid], [InterfaceType], [ComImport] so the CLR (Common Language Runtime) knows how to call into native COM.



Runtime Callable Wrapper (RCW)

An RCW is a CLR-generated object that wraps the actual native COM object and allows your C# code to call into it as if it were a normal .NET object.

The purpose of this object is:

- Method dispatch
- Parameter marshalling
- Return Value Conversion
- Reference counting/lifetime
- Exception translating (HRESULT to .NET Exception)
- Threading models (STA/MTA)

All Steps from C++ COM to C# client code:

- Step 1:
  - C++ COM developer gives you .tlb (type library) of installed on the PC .DLL (COM Component)
- Step 2:
  - You (as a C# developer) reference in your C# project the specified .tlb file (tlbimp.exe is used behind the scenes). This will generate COMInterop library for the same .tlb.
- Step 3:
  - You can use the classes and interfaces from the COMInterop proxy library in your C# project as a normal classes and interfaces (The COMInterop library is the .NET vision for the .tlb file).
- Step 4:
  - At runtime **CLR** (Common Language Runtime) reads **CLSID** from Interop **DLL** and looks into the Windows Registry. Loads the real .DLL from the presented file path there.
  - Calls **CoCreateInstance** (loads native DLL)
  - Native object returned
    - **CLR** wraps it in **RCW**
    - Your C# code calls **RCW**

Important: COM Infrastructure/technology behind the scenes uses Windows Registry to find and load the real .dll.

What is IUnknown in COM?

IUnknown is the most fundamental interface in **COM**. Every **COM** object must implement it - it's the foundation that enables all COM-based programming.

Even if you don't see it in the C# code, it's always working behind the scenes. .NET automatically handles it for you via RCW.

Why IUnknown is important?

It provides three core methods that control how COM objects behave:

- QueryInterface
- AddRef
- Release

Every COM object must implement these three special methods. They are the foundation for everything in COM - memory management, interface discovery, and safe communication.

What is the QueryInterface method?

QueryInterface method is one of the most important methods is COM. It lets you ask a COM object if it supports a particular interface - and if it does, gives you a way to use that interface.

What does QueryInterface actually do?

- COM object can implement/support many different interfaces.
- **QueryInterface** is how you check if a certain interface is available.
- If the interface is supported, **QueryInterface** returns a pointer to it.
- If not, it tells you "No, this object doesn't support that interface".

What do you need to use QueryInterface?

- A pointer or reference to a COM object you already have.
- The ID (called IID) of the interface you want to check for.
- A place to store the pointer to the interface if it's supported.

Why is QueryInterface important?

- COM objects often support multiple interfaces
- **QueryInterface** is the only safe way to switch between interfaces on the same object.
- It ensures you only use interfaces the object actually supports, preventing errors.

In C# you don't usually call **QueryInterface** yourself. Instead, casting an object to a different COM interface automatically calls it behind the scenes.

```csharp
using System;
using System.Runtime.InteropServices;

[ComImport]
[Guid("00000000-0000-0000-C000-000000000046")]  // Example IID: IUnknown (replace with real IID)
[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
interface IMyInterface
{
    void SomeMethod();
}
```

```

class Program
{
    static void Main()
    {
        // Assume this gets a COM object as 'object'
        Type comType = Type.GetTypeFromProgID("Your.ProgID.Here");
        object comObject = Activator.CreateInstance(comType);

        // Cast to IMyInterface - triggers QueryInterface behind the scenes
        IMyInterface myInterface = (IMyInterface)comObject;

        // Now you can call methods on the interface
        myInterface.SomeMethod();

        // When done, runtime will handle releasing references
    }
}

```

So, the flow is:

- Get an **IUnknown** pointer (different ways in C++ and C#)
- Call **QueryInterface** on that **IUnknown** pointer with the **IID** of the interface you want.
- If supported, you get a pointer to that interface.

What is the AddRef method?

AddRef is one of the three core methods of the IUnknown interface in COM. It is used to manage the lifetime of the COM object.

In simple words: AddRef increases the internal reference counter of a COM object.

Whenever a new client (like a C# code) starts using COM object, AddRef is called automatically to tell the object:

> “Someone else is using you - don’t delete yourself yet.”

Why is AddRef important?

- COM uses reference counting to know when it is safe to destroy an object.
- Every time someone wants to use the object, they “Add a reference” using AddRef
- When someone is done, they call Release to decrease the count.
- When the count reaches **zero**, the COM object deletes itself.

> You don’t need to call **AddRef** manually in C#. .NET handles this behind the scenes using **RCW** (Runtime Callable Wrapper), so your C# code doesn’t have to worry about calling **AddRef** or **Release** directly.

What is Release method?

Release is the opposite of **AddRef**. In simple words: “**Release** tells the COM object: “I am done using you - you can lower your reference count.”

Every time someone finishes using a COM object, they must call **Release**. This decreases the internal reference counter by 1.

When the counter reaches **zero,** the object **automatically deletes itself -** because nobody is using it anymore.

What happens if you forget to call Release?

- The object stays alive in memory - this causes a memory leak

- It never gets cleaned up, even if it's no longer needed

In .NET (C#) there are good news!

.NET automatically tracks references through the RCW (Runtime Callable Wrapper) and calls **Release** for you when:

- The RCW is garbage collected
- Or you call Marshal.ReleaseComObject() manually

You don't need usually to worry about Release unless you're doing advanced interop or managing COM manually.

COM Registration

- COM components must be registered in the Windows Registry
- Registration maps CLSID to a file path (DLL/EXE)
- In C#, if registration is missing, you will get an error like: "COM object with CLSID...not registered."

HRESULT and Exceptions

- COM methods return HRESULT values to indicate success/failure
- .NET automatically converts HRESULT into exceptions

Example:

HRESULT 0x80004005 → **COM Exception**: Unspecified error

Early Binding vs Late Binding

Early Binding:

Definition: The method or property you are calling is known at compile time.

Usage in C#: Typical use when referencing a COM object via a strongly-typed interop assembly.

Performance: Faster, because the method addresses are resolved ahead of time.

Mechanism: Uses the COM **vtable** (virtual table) to call functions directly via memory address.

```
var wordApp = new Microsoft.Office.Interop.Word.Application();
wordApp.Quit(); // Early-bound: compiler knows Quit() method exists.
```

Late Binding:

Definition: Method or property to call is determined at runtime.

Usage in C#: Common when using dynamic or object or calling COM without a PIA (Primary Interop Assembly).

Performance: Slower, as reflection or IDispatch must resolve the method name at runtime.

Mechanism: Uses the IDispatch interface

```
Type excelType = Type.GetTypeFromProgID("Excel.Application");
object excelApp = Activator.CreateInstance(excelType);
excelType.InvokeMember("Quit", BindingFlags.InvokeMethod, null, excelApp, null);
```

What is the IDispatch interface?

IDispatch is a standard COM interface that enables late binding.

- Inherits from IUnknown.
- Introduced for scripting languages (VBScript, JavaScript) that don't support early binding or vtables.
- Allows methods/properties to be invoked by name at runtime.

IDispatch methods:

```
HRESULT GetTypeInfoCount(UINT *pctinfo);
HRESULT GetTypeInfo(UINT iTInfo, LCID lcid, ITypeInfo **ppTInfo);
HRESULT GetIDsOfNames(
    REFIID riid,
    LPOLESTR *rgszNames,
    UINT cNames,
    LCID lcid,
    DISPID *rgDispId);

HRESULT Invoke(
    DISPID dispIdMember,
    REFIID riid,
    LCID lcid,
    WORD wFlags,
    DISPPARAMS *pDispParams,
    VARIANT *pVarResult,
    EXCEPINFO *pExcepInfo,
    UINT *puArgErr);

```

How it works:

- **GetIDsOfNames:** Converts a method/property name (e.g. "Quit") to a numeric ID (DISPID)
- **Invoke:** Calls the method using its DISPID.

Behind the scenes in C#:

When we use late binding via InvokeMember or dynamic, .NET:

- Calls IDispatch::GetIDsOfNames
- Then calls IDispatch::Invoke

So, we are not calling the method directly - we are telling the COM object "Invoke this method name, if it exists".

What is VTable (Virtual Method Table)?

A **vtable** is a low-level structure used in C++ and COM that holds pointers to methods for an interface.

- Each COM interface like IUnknown or custom one, has its own vtable.
- Early Binding uses these tables for direct method dispatch - fast and type safe

Relations to COM:

- COM interfaces are essentially pointers to **vtables**.
- When you cast an interface, you are working with that interface **vtable**.

VTbale vs IDispatch:

Feature	**VTable binding (Early)**	**IDispatch binding (Late)**
Speed	Fast	Slower (name lookup)
Method Resolution	Compile-time	Runtime

Used by	C++, C# with Interop assemblies	Scripting, Dynamic
Implementation	Direct method calls via pointer	Name → DISPID → Invoke

64-Bit vs 32-Bit COM compatibility

A very common real-world problem:

- COM DLLs are either 32-bit or 64-bit
- Your C# app must match the bitness of the native consumed COM DLL.

COM works on binary level, meaning:

- A 64-bit process cannot load 32-bit .DLL (COM or not)
- A 32-bit process cannot load 64-bit .DLL (COM or not)

There is not automatic translation or compatibility layer for this - you'll get errors like:

"Retrieving the COM class factory for component with CLSID {xxx} failed due to the following error: 80040154 Class not registered. "

Even the class is registered, the error usually means: "You are trying to load a 32-bit COM .DLL into 64-bit process, or vice versa. "

Advanced and practical COM + C# tips:

Use dynamic for Late Binding (Caution)

If you are working with COM objects like Excel, Word, or other office interop, you can use **dynamic** to simplify the access.

```
dynamic excelApp = Activator.CreateInstance(Type.GetTypeFromProgID("Excel.Application"));
excelApp.Visible = true;
```

- Pros - Cleaner syntax, no casting needed
- Cons - No Compile-time checking, errors only appear in Runtime

Tip: For production code, prefer strongly typed interop when is possible

Debugging COM Interop:

- Use **OLE/COM Object Viewer** to inspect **COM** type libraries
- **GitHub** page of **OLE/COM Object Viewer**: [tyranid/oleviewdotnet: A .net OLE/COM viewer and inspector to merge functionality of OleView and Test Container]()

You can download the latest release as a .zip file and to extract it. This will bring to you two .exe files for 32-bit and 64-bit.