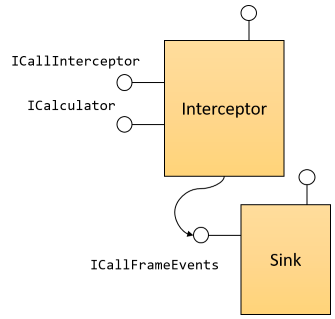
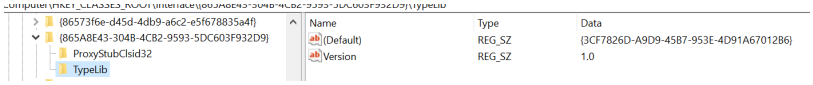
# COM Interception with CoGetInterceptor - Part 2

In the **previous post** I showed how one can use `CoGetInterceptor` to return an interceptor object to clients, where the interceptor has a dual face: on the one hand it's an interceptor (implements `ICallInterceptor`) and on the other hand it implements the interface to intercept. I hinted at some finer points of this technique that I would like to address in this post.

Here is what we have so far: an interceptor implementing the `ICallInterceptor` and `ICalculator`:




How does `CoGetInterceptor` "know" how the interface to intercept looks like? The answer is type library. `CoGetInterceptor` looks at the registry at **HKCR\Interface\{IID}** for a registered type library. For the example provided in the previous post, the registry entry for `ICalculator` looks like this:




And the registered pointed-to type library looks like this:


Pointing to the DLL (which includes the TLB file embedded as one of its resources).

These registration entries are typical of dual interface using the Type Library Marshalar. but what about interfaces that don't use TLB marshalling? Or there is no type library?

In such a case, we can load the type library directly, and use a `CoGetInterceptor`-sister function, `CoGetIterceptorFromTypeInfo`. In fact, this function may be more efficient as it doesn't have to hunt down the type library in the registry. We can just load the type library once and use it in subsequent calls. Note that this function seems to be undocumented, but it works as expected. Here's an example of loading a needed type library, locating an interface and using `CoGetIntercetorFromTypeInfo' to generate the interceptor (error handling omitted):

```
CComPtr<ITypeLib> spTypeLibrary;
::LoadTypeLib(L"mytlb.tlb", &spTypeLibrary);
CComPtr<ITypeInfo>spTypeInfo;
spTypeLibrary->GetTypeInfoOfGuid(__uuidof(ICalculator), &spTypeInfo);
::CoGetInterceptorFromTypeInfo(__uuidof(ICalculator), nullptr, spTypeInfo,
__uuidof(ICallInterceptor),
    reinterpret_cast<void**>(&spInterceptor));
```

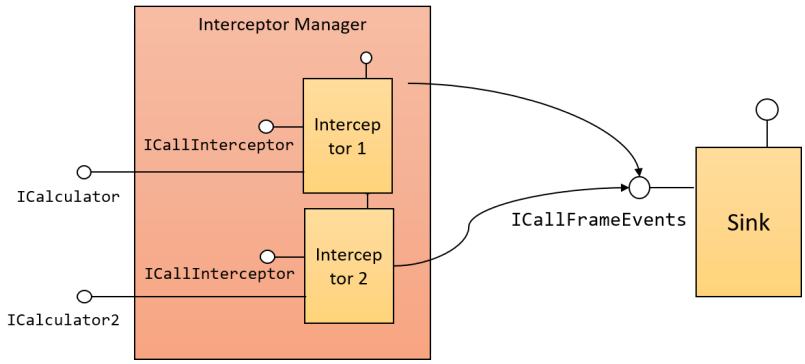
If the type library does not exist, it must be generated. In the worst case, we could reconstruct the IDL for the interfaces we're interested in and pass it to the MIDL compiler to generate the type library.

Now let's tackle a more complex problem. Let's suppose the Calculator object supports a second interface, ICalculator2. What would happens if the client queries this interface? the result would be that no such interface is supported. Do you see why?

The query goes to the interceptor that is only aware of `ICalculator`, so it would reply "no such interface". How can we tell the interceptor about this interface? Calling `CoGetInterceptor' again with the ICalculator2 interface is not good enough, because now we would have two distinct objects. Which one should we`

Fortunately, there is a way. This is that `IUnknown` pointer that we can provide to `CoGetInterceptor`. We need to create an aggregated object that aggregates the two interceptors (one for `ICalculator` and the other for `ICalculator2`) and this aggregated object is the one returned to the client. Here is what it looks like graphically:

# COM Interception with CoGetInterface - Part 2




This architecture now requires a new class to serve as the Interception Manager and aggregate the two interceptors:

```
struct __declspec(uuid("{CAEEA7BE-BB01-4B01-AEF4-DD3EDD914930}")) IDummyInterceptor : IUnknown
{};

class CCalculatorInterceptor : public CComObjectRoot, public IDummyInterceptor {
public:
    BEGIN_COM_MAP(CCalculatorInterceptor)
        COM_INTERFACE_ENTRY(IDummyInterceptor)
        COM_INTERFACE_ENTRY_AGGREGATE(__uuidof(ICalculator), m_spCalcInterceptor.p)
        COM_INTERFACE_ENTRY_AGGREGATE(__uuidof(ICalculator2), m_spCalc2Interceptor.p)
    END_COM_MAP()

    DECLARE_GET_CONTROLLING_UNKNOWN()
    DECLARE_PROTECT_FINAL_CONSTRUCT()

    void SetInterceptors(IUnknown* i1, IUnknown* i2) {
     m_spCalcInterceptor = i1;
     m_spCalc2Interceptor = i2;
    }

    CComPtr<IUnknown> m_spCalcInterceptor, m_spCalc2Interceptor;
};
```

I created a second factory method, CreateCalculator2 to do the job, so not to disturb the existing code. First, we create the manager and the real calculator (error handling omitted):

```
// create the manager

CComObject<CCalculatorInterceptor>* pIntManager;
pIntManager->CreateInstance(&pIntManager);

// create the real object

CComObject<CCalculator>* pCalculator;
pCalculator->CreateInstance(&pCalculator);
```

Now we create the two interceptors and point the controlling IUnknown to our manager:

```
// create the interceptors

CComPtr<IUnknown> spUnkInterceptor1;
::CoGetInterceptor(__uuidof(ICalculator), pIntManager->GetControllingUnknown(),
__uuidof(IUnknown),
    reinterpret_cast<void**>(&spUnkInterceptor1));

CComPtr<IUnknown> spUnkInterceptor2;
::CoGetInterceptor(__uuidof(ICalculator2), pIntManager->GetControllingUnknown(),
__uuidof(IUnknown),
    reinterpret_cast<void**>(&spUnkInterceptor2));

// let the manager know where to dispatch QI calls

pIntManager->SetInterceptors(spUnkInterceptor1, spUnkInterceptor2);
```

Next, we create the generic sink and register both interceptors to use it:

```
CComObject<CGenericInterceptorSink>* pGenericInterceptor;
pGenericInterceptor->CreateInstance(&pGenericInterceptor);
pGenericInterceptor->SetObject(pCalculator->GetUnknown());

CComQIPtr<ICallInterceptor> spInterceptor1(spUnkInterceptor1);
spInterceptor1->RegisterSink(pGenericInterceptor);

CComQIPtr<ICallInterceptor> spInterceptor2(spUnkInterceptor2);
spInterceptor2->RegisterSink(pGenericInterceptor);
```

Finally, we're ready to return the manager interface to the client:

```
return pIntManager->QueryInterface(ppCalculator);
```

This is it. A complete interception mechanims with objects that support multiple interfaces. I have updated the GitHub repo with these code changes.

## More interception options

We can combine this technique with others, such as `CoTreatAsClass` I blogged about previously, or something more interesting like **`CoRegisterActivationFilter`**. I'll leave these for the interested reader to explore.