

# Microsoft COM

Mark Morrissey  
CSE 58x  
Winter 2000

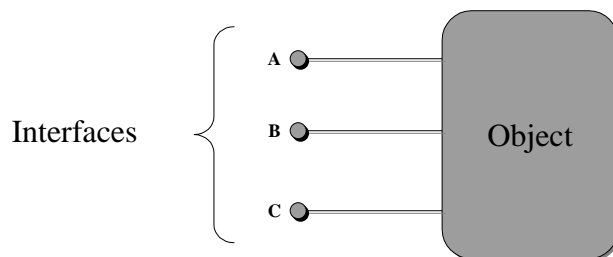
## COM, C++, Java

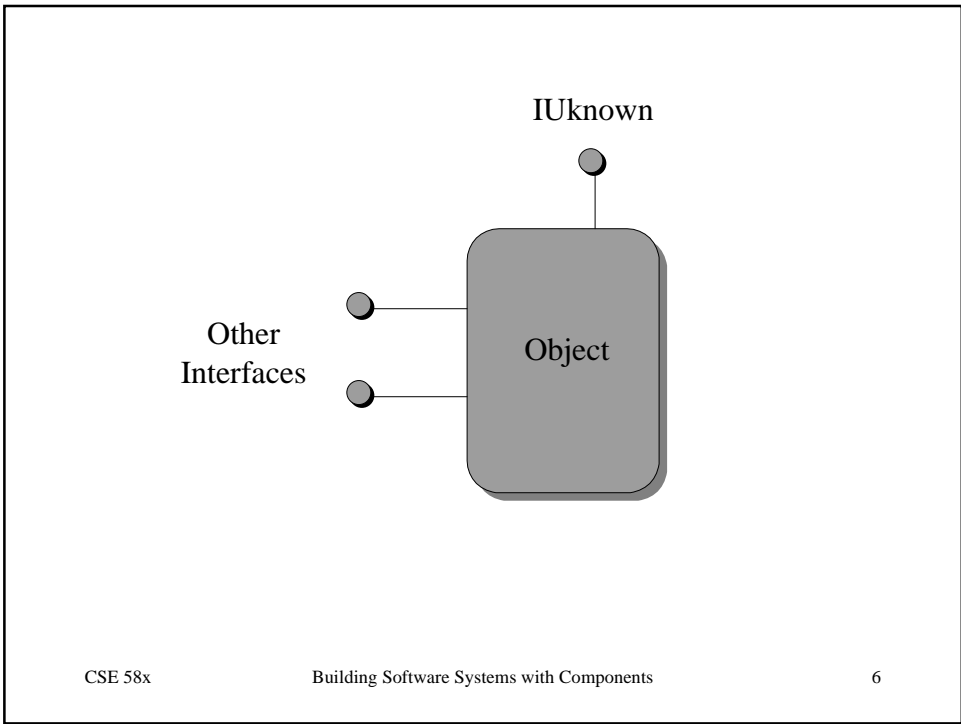
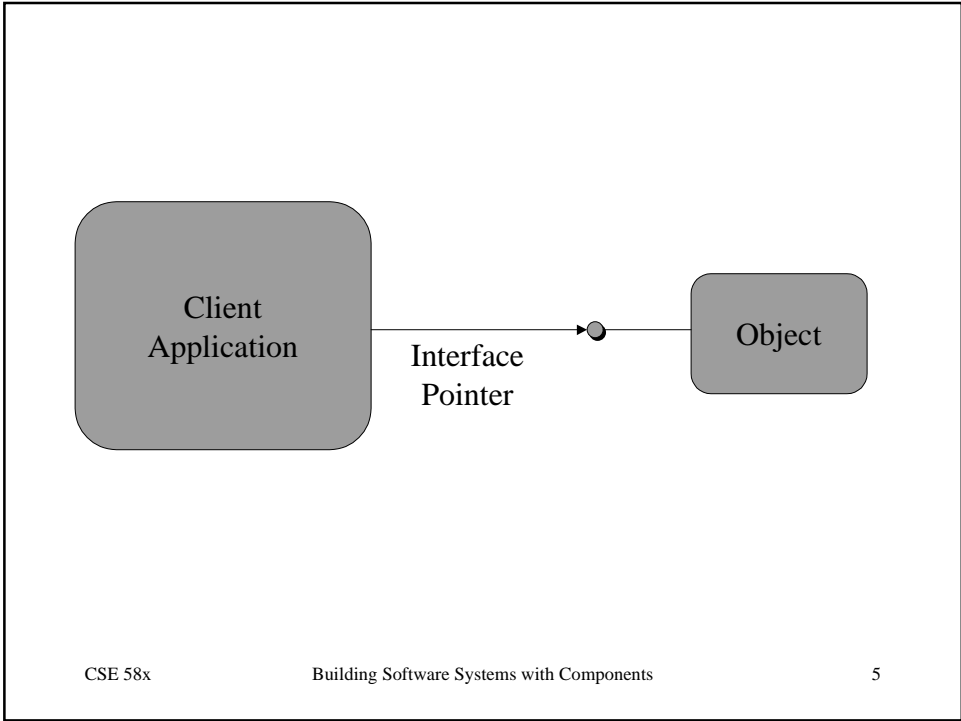
- In order to illuminate how COM works, the COM lectures will mostly be in C++
- This is because the Java Virtual Machine hides much of the COM structure from the programmer
- In particular, IUnknown, QueryInterface, AddRef, Release, IDispatch, etc. are not visible (normally) to the Java COM programmer

## COM's Role

- COM allows clients to discover functionality exposed by an object via an interface
- Once COM connects a client and object, the client and object communicate directly without COM overhead
- This is often compared to CORBA, but CORBA operates at a different level of abstraction

## COM Pictures





## COM is an *Interface Spec*

- COM Components must adhere to a binary external standard
  - The internal implementation is completely unconstrained
  - Provided, of course, that the pre- and post-conditions are not violated
- COM is targeted at creating an “off the shelf” component marketplace
  - Analogous to board-level hardware design where ICs are purchased out of a catalog
- To achieve, this COM provided an *interaction* model

## Help for Developers

- The interaction model helps developers
  - By providing a framework to design against
  - By giving application developers a *rendezvous point* for **choosing** services among many vendors
  - By keeping developers focused on the value-add of the product
  - By allowing developers the freedom to purchase commodity parts of the application

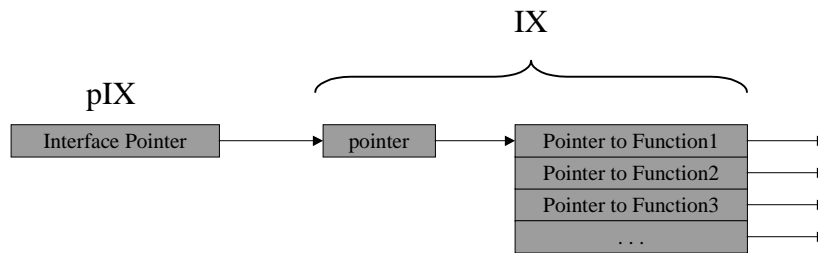
## Helps End Users

- Faster time to market
  - Solutions arrive as problems occur
- Easy field upgrades
  - Only upgrade the components that need upgrading
  - Helps keep unwanted new “features” from creeping in
- Allows for applications that independently interact at the customer’s direction
  - Why let the vendor choose your components, pick your own and deploy them for your environment
  - Allows customer-driven value add

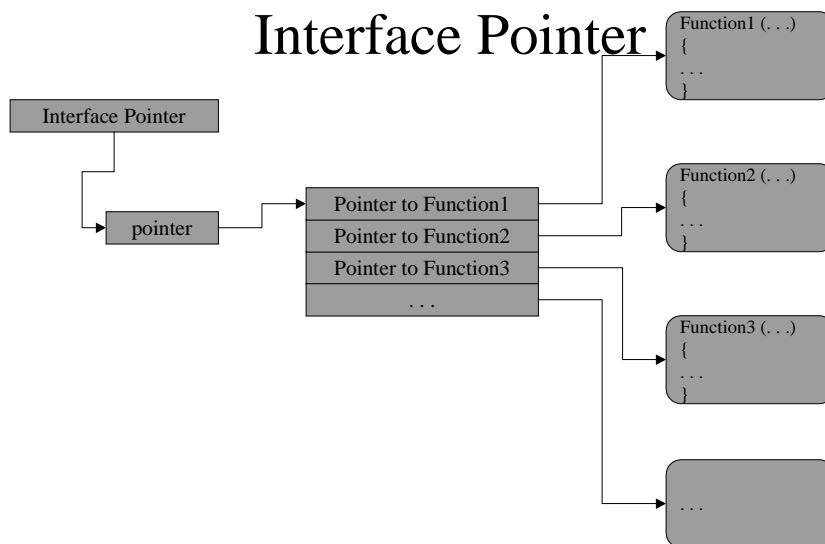
## What’s the Magic

- COM’s strength results from innovative stealing from C++
- The COM interface is really just a C++ virtual function pointer table structure
  - Called a vtbl for short
- Allows for the necessary indirection to facilitate late binding and language independence
- Any language that can support a vtbl structure can generate COM objects
- Simple, isn’t it?
- A vtbl is also called an interface pointer

# Virtual Function Table (vtbl)



# Interface Pointer



## Question

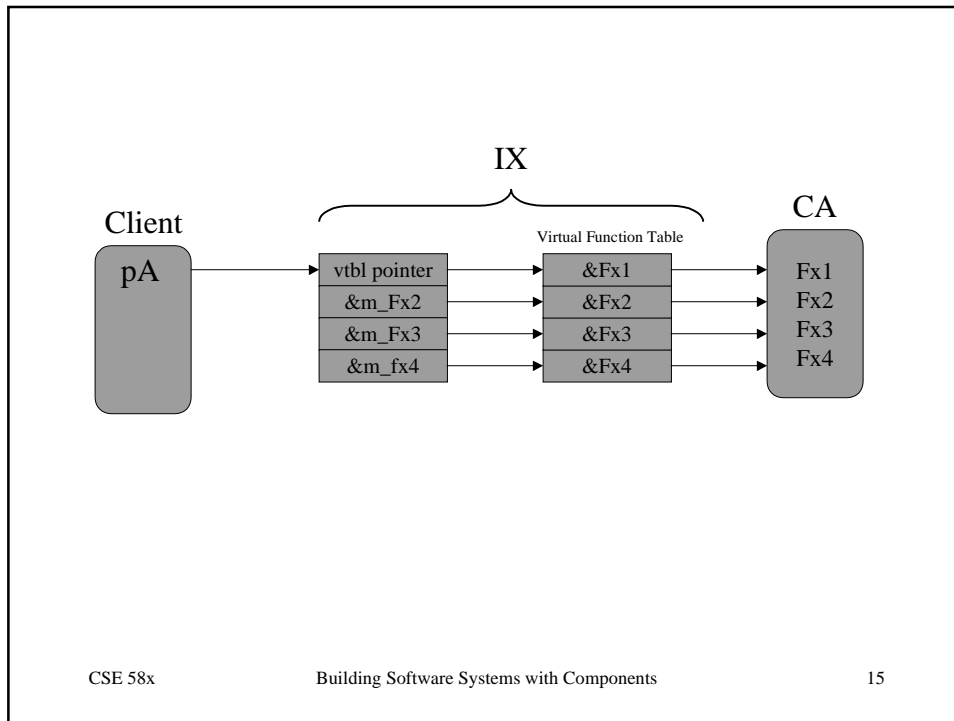
- Where do we store instance specific data?
- Can we somehow store it in the vtbl
  - We don't want to force the vtbl structure to “know” about instance data since that would cause the interface to change for each COM object
  - Finding a way to “hide” instance data would be useful
  - This would allow the basic vtbl structure to remain pure
- Can we do this without changing the basic vtbl structure?
- The answer is “yes”

## vtbl Pointers and Instance Data

```
class CA : public IX
{
public:
    virtual void __stdcall Fx1() {cout << "CA::Fx1" << endl; }
    virtual void __stdcall Fx2() {cout << m_Fx2 << endl; }
    virtual void __stdcall Fx3() {cout << m_Fx3 << endl; }
    virtual void __stdcall Fx4() {cout << m_Fx4 << endl; }

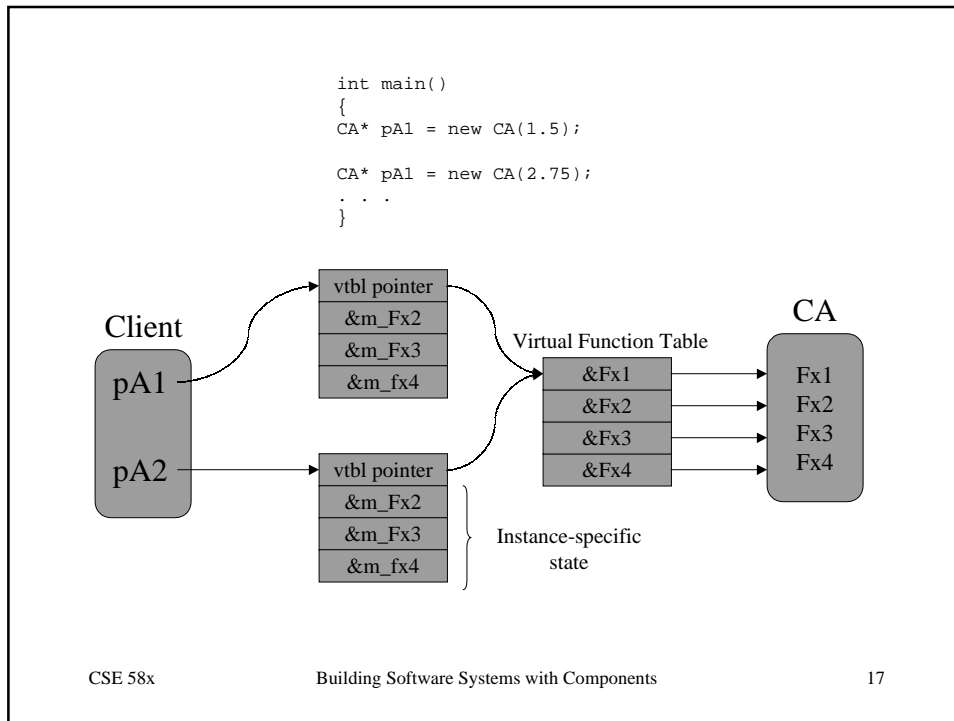
    CA (double d)
    : m_Fx2(d*d), m_Fx3(d*d*d), m_Fx4(d*d*d*d)
    { }

    double m_Fx2;
    double m_Fx3 ;
    double m_Fx4;
};
```



## But. . .

- What about efficiency?
- These vtbl structures look an awful lot alike
- Wouldn't it be more memory efficient to only duplicate that part of the vtbl you needed to?
- Very good, grasshopper



## About Interfaces

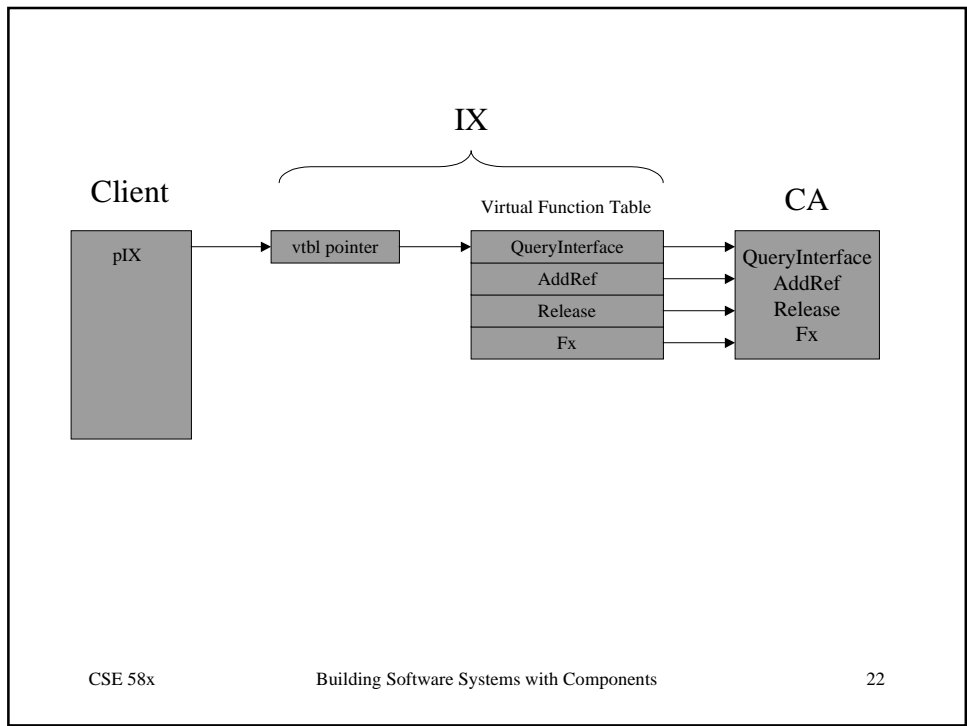
- An interface is not a class
  - An interface carries no implementation and cannot be instantiated by itself
  - Multiple classes can be used to implement an interface and all may be interchangeable
- An interface is not an object
  - Merely a related group of functions that provides a communications mechanism for clients and objects
- Interfaces are strongly typed
  - Conflicts cannot happen by accident
  - GUIDs must be consciously assigned to an interface and an interface must be consciously supported

- Interfaces are immutable
  - Avoids versioning problems
  - Once released, can never be modified
  - A new version of the interface needs an entirely new GUID
- Clients only interact with pointers to interfaces
  - The pointer is opaque, unlike C++ object pointers
- Objects can have multiple interfaces
  - This is the common case
  - e.g. an object may have one interface to exchange data with the client and one interface to save persistent state information

## Into the Great IUnknown

- The Universal Starting Point™
- You have to have someplace to go to find out what interfaces a COM object supports
- The only interface that every COM object is *required* to support
- Contains 3 interfaces
  - QueryInterface
  - AddRef
  - Release

- Used to query the object for other interfaces and maintain object-wide reference counts
- All COM objects inherit from IUnknown



## QueryInterface (QI)

- Clients discover interface support at run time
- Clients query the object for a specific interface
- The object hands back an interface pointer if the interface is supported by the object
- The client now communicates with the object through the opaque interface pointer

## Getting to QI

- Use CreateInstance instead of new
  - IUnknown CreateInstance()
- QI takes two parameters:
  - IID - a GUID representing the interface we want
  - An address to place the interface pointer
  - An HRESULT is returned

```
HRESULT __stdcall QueryInterface(const IID& iid, void** ppv)
```

## QI Example

```
void foo (IUnknown* pI)
{
    IX* pIX = NULL;

    HRESULT hr = pI->QueryInterface(IID_IX, (VOID**)&pIX);

    if (SUCCEEDED(hr))
    {
        pIX->Fx();    // use the interface
    }
}
```

## Implementing QI

```
HRESULT __stdcall CA::QueryInterface(const IID& id, void ** ppv)
{
    if (iid == IID_Iunknown)
        *ppv = static_cast<IX*>(this);
    else if (iid == IID_IX)
        *ppv = static_cast<IX*>(this);
    else if (iid == IID_IY)
        *ppv = static_cast<IY*>(this);
    else {
        *ppv = NULL; // good, defensive programming
        return E_NOINTERFACE;
    }
    static_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}
```

## QI errata

- It is very important to cast the *this* pointer before storing into the \*ppv pointer because different languages handle multiple inheritance differently
- No matter what, always return the same interface for IUnknown
  - The only way to tell if two objects are the same is to compare IUnknown pointers
  - Returning different pointers defeats the ability to compare objects

## Reference Counting

- Used for object life-cycle management
- AddRef
  - Defined on a per-interface basis, but most objects implement it per-object
  - However, do not get lazy, always access through the correct interface just in case it is implemented per-interface
- Release
  - Used to delete an object

- Instead of deleting an object directly, we tell the object that we are through using it and let the object decide if it is time to delete itself
- Basic rules:
  - Call AddRef before returning. Functions that return interfaces should call AddRef on the pointer before returning
  - Call Release when you are done. When finished with an interface, call Release on that interface
  - Call AddRef after assignment. Whenever you assign an interface pointer to another interface pointer, call AddRef.

```
IUnknown* pIUnknown = CreateInstance();

IX* pIX = NULL;
HRESULT hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX);

pIUnknown->Release();

if (SUCCEEDED(hr)) {
    pIX->Fx();
    pIX->Release();
}
```

- **AddRef:**

```
ULONG __stdcall AddRef() {  
    return ++m_cRef;  
}
```

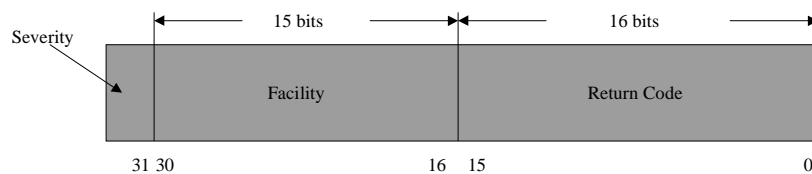
- **Release:**

```
if (--m_cRef == 0) {  
    delete(this);  
    return 0;  
}  
return m_cRef
```

- Why not always return m\_cRef?
- You can optimize AddRef/Release pairs away at times, but you must be very, very careful

## Error Codes

- Most people think that HRESULTs are handles to result
  - This isn't true
- An HRESULT is a 32-bit value divided into three different fields



- winerror.h contains the definitions for all COM error codes

# Enumerators

- Enumerator Interfaces are used to iterate through a sequence of items
- Enumerators are just a concept
  - There is no actual IEnum or IEnum
  - Function signatures in an enumerator interface must include the types of things to be enumerated
  - The difference in the type being enumerated is the only different between each interface
  - Each enumerator interface is essentially used the same way

```
[
    object,
    uuid (<ID_Ienum<ELT_T>>), // IID_Ienum<ELT_T>
    pointer_default(unique)
]
interface Ienum<ELT_T> : Iunknown
{
    HRESULT Next([in]ULONG celt,
                [out]IUnknown **rgelt,
                [out]ULONG *pceltFetched);
    HRESULT Skip([in]ULONG celt);
    HRESULT Reset(void);
    HRESULT Clone([out]IEnum<ELT_T>**ppenum);
}
```

```

// Somewhere there is a string called "String"
typedef char *String;

typedef IEnum<char*> IEnumString;
. . .
interface IStringManager {
    virtual IEnumString* EnumStrings(void) = 0;
}
. . .
void SomeFunc(IStringManager* pStringMan) {
    char* psz;
    IEnumString* penum;
    penum = pStringMan->EnumStrings();
    while (S_OK == penum->Next(1, &psz, NULL))
    {
        // Do something with string in psz and free it
    }
    penum->Release();
    return;
}

```

## Enumerator Summary

- We saw that enumerator interfaces are used to iterate through a collection
- Collections can be implemented in any way: heaps, linked lists, etc.
- But the enumerator interface provides a consistent interface to iterate through the collection without knowledge of the underlying implementation details

## Class Factory

- A class factory is a specialized component whose only job is to create other components
- Class factories exist because the standard way to create components (CoCreateInstance) isn't flexible enough to meet the needs of all components
- Guess what, CoCreateInstance is really just a generic wrapper around a class factory
- If you call the class factory directly, you have more flexibility

## CoCreateInstance

```
HRESULT __stdcall CoCreateInstance (  
    const CLSID& clsid,  
    IUnknown* pIUnknownOuter,    // Outer Component  
    DWORD dwClsContext,          // Server Context  
    const IID& iid,  
    void** ppv  
);
```

We'll ignore the pIUnknownOuter parameter. It is used for containment and aggregation

```

// Create A Component
IX* pIX = NULL; // good, defensive programming
HRESULT hr = ::CoCreateInstance(
    CLSID_Component1,
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_IX,
    (void**) &pIX );

if (SUCCEEDED(hr)) {
    pIX->Fx();
    pIX->Release();
}

```

- As you can see, using CoCreateInstance is similar to using QueryInterface

## Class Context

- The third parameter to CoCreateInstance is used to create the process context for the component
  - There are 4 general categories
- CLSCTX\_INPROC\_SERVER
  - Component is allowed in the same process space. Requires that the component be implemented as a DLL
- CLSCTX\_INRPOC\_HANDLER
  - Part is in-proc and the rest local or remote

- CLSCTX\_LOCAL\_SERVER
  - Different process, same machine
- CLSCTX\_REMOTE\_SERVER
  - Components may reside on remote computers (DCOM)
- These categories are only *hints* to the system about where the client will tolerate the component being located
- The hints may be combined. COM will return the one most suited to your needs
  - Components have a say as to how they will be created
  - If common ground can't be found, creation fails

## Context Shortcuts

- CLSCTX\_INPROC
  - CLSCTX\_INPROC\_SERVER  
| CLSCTX\_INPROC\_HANDLER
- CLSCTX\_ALL
  - CLSCTX\_INPROC\_SERVER  
| CLSCTX\_INPROC\_HANDLER  
| CLSCTX\_LOCAL\_SERVER  
| CLSCTX\_REMOTE\_SERVER
- CLSCTX\_SERVER
  - CLSCTX\_INPROC\_SERVER  
| CLSCTX\_LOCAL\_SERVER  
| CLSCTX\_REMOTE\_SERVER

## CoCreateInstance Problems

- Most of the time, CoCreateInstance suffices
- However, once CCI returns, the object is fully created
- It's now too late to control issues like
  - Where the component is loaded in memory
  - Checking to see if the client has sufficient permissions to create the component
  - Logging the creation, etc., etc.
- The problem is how to control the creation
  - Controlling the initialization is easy since we do that after creation
  - So how do we put conditions on creation?

## Controlling Creation

- Class factories create components corresponding to a single, specific CLSID
- The standard interface to a class factory is IClassFactory
- How do we get a pointer to the class factory?
  - Use CoGetClassObject
- CoGetClassObject returns a pointer to a class factory whereas CoCreateInstance returns a pointer to the component itself

# IClassFactory

```
Interface IClassFactory : IUnknown {
    HRESULT __stdcall CreateInstance(
        IUnknown pUnknownOuter,
        const IID& iid,
        void** ppv);

    HRESULT __stdcall LocalServer (BOOL bLock);
};
```

We won't go into all the gory details. Consult the online MS documentation if interested

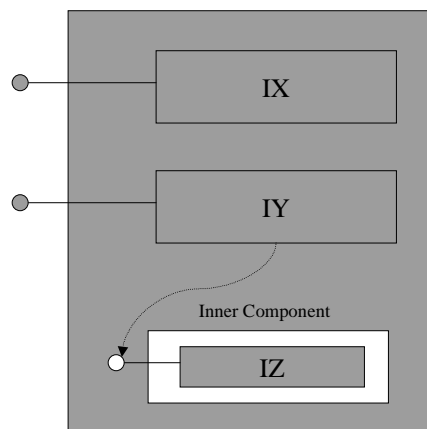
# Why CoGetClassObject?

- There are two cases where CGCO should be used over CCI
  - If you want to use a creation interface other than IClassFactory
  - You want to create many components all at one time
- You would create specialized component creation interfaces if you had different needs than the standard interface
  - e.g. if you need to authenticate the client
- Creating many components at one time is more efficient than one-at-a-time for components in separate process spaces or on remote machines

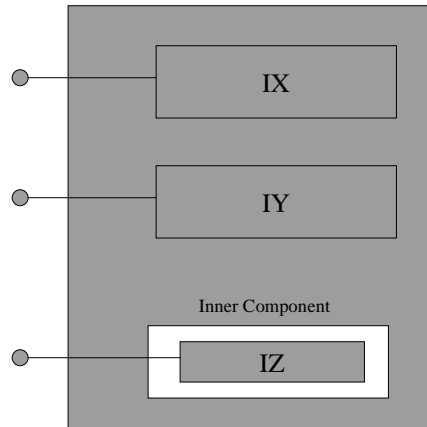
# Component Reuse

- Components may be specialized using containment and aggregation
- Both techniques allow a component to use other components
  - The using component is called the *outer* component and the re-used component is called the *inner* component

# Containment



# Aggregation



# Subtleties

- In containment, the outer component is the gatekeeper to the inner component
- In aggregation, the outer component never has a chance to intercept invocations on the contained object
- Should be invisible to the client, but is very, very visible to the component designers
  - The issues are subtle, revolving mostly around the IUnknown interface
  - Recall that all interfaces can be used to ask for the IUnknown interface and that all interfaces must return the *same* IUnknown
  - Also, reference counting and deleting objects becomes harder
  - There are standard techniques in the COM documentation

## Dispatch and Automation

- Automation exists so that interpretive and macro languages can access COM object
- Automation is built on top of COM
- Automation invokes COM functions through dispatch interfaces
- IDispatch is a way for a component to offer all its services through a single interface
- IDispatch is a way for a macro in a run-time system to execute a function via the function's name
  - Recall that we have been using CLSIDs so far

## Dual Interface

- Dual interfaces are the preferred mechanism for implementing dispatch interfaces
- Dual interfaces contain normal vtbl interfaces and IDispatch interfaces
  - Allows the run-time to choose the easier (and for C++ faster) invocation mechanism
- The gory details are somewhat advanced, so if you need to know them, get a good COM reference

## Events and Connection Points

- We talked before about callbacks
- In Java, we use the observer / observable classes
- In COM, these are really event sinks
  - Event sources and sinks are implemented using connection points
  - These are invisible below the Java cover
  - However, it is important to know that the java observer and observable constructs map down to COM connection points

## Sigh, Made it

- The gory details of COM are mostly hidden by J++
- This can be good, this can be bad
  - Anyone see “The Wizard of Oz”?
- However, knowing what is really going on behind the scenes can greatly affect your use of the technology and also aid substantially in finding and, more importantly, **avoiding** problems
- Know your technology before you use it