Some COM Details

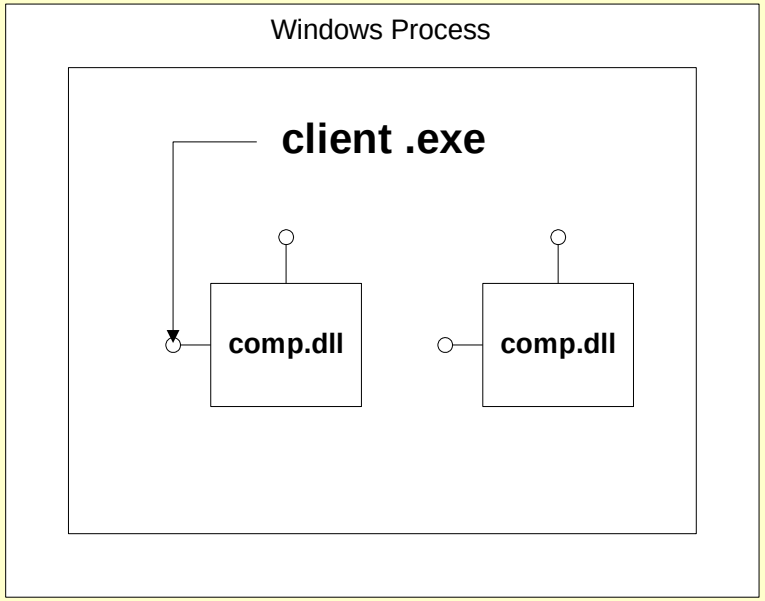
Jim Fawcett

CSE775 - Distributed Objects

Spring 2012

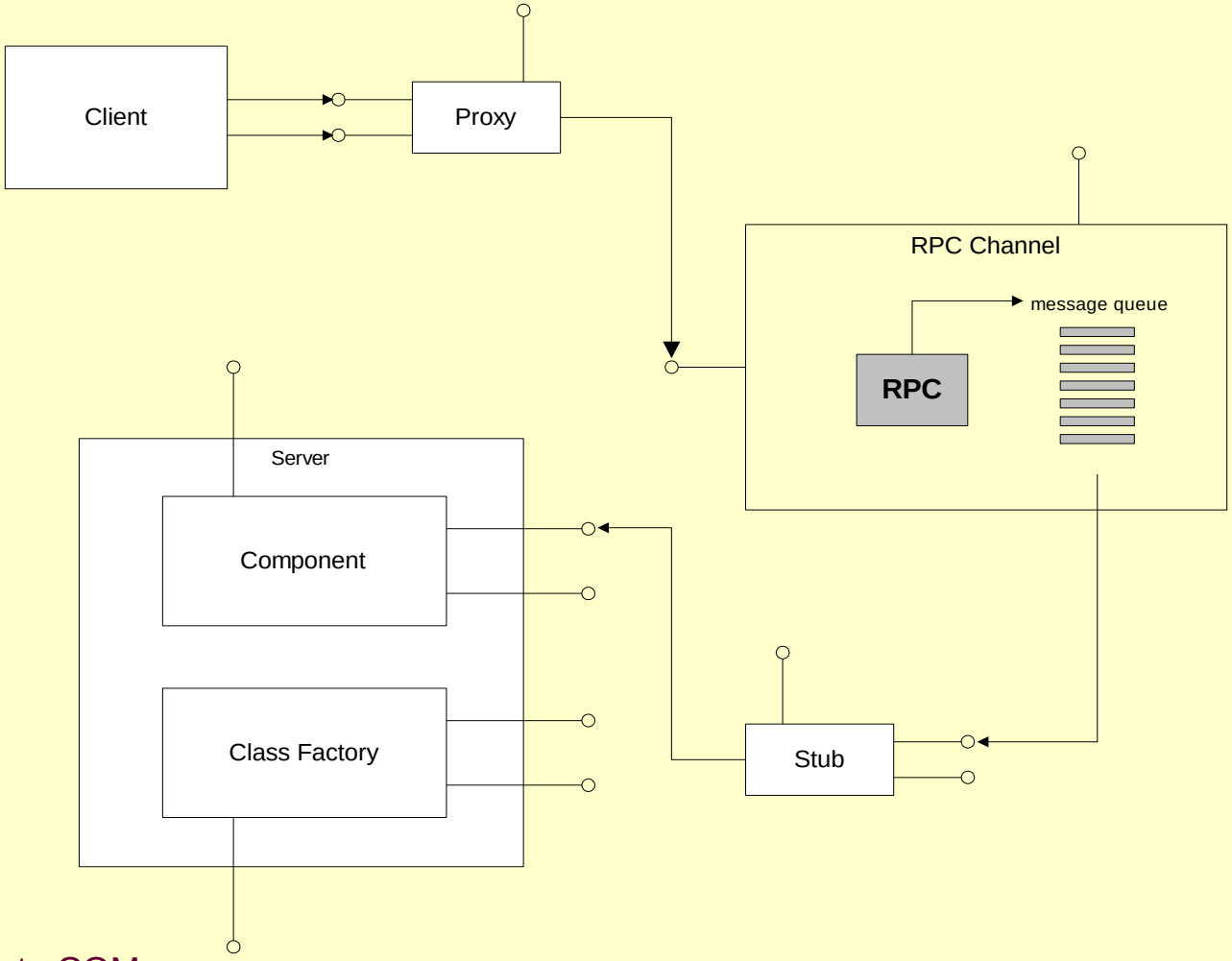
In-Proc Components

- Client loads COM dll into its own process.
- If client and component share the same threading model then client gets a pointer to The component's instance.



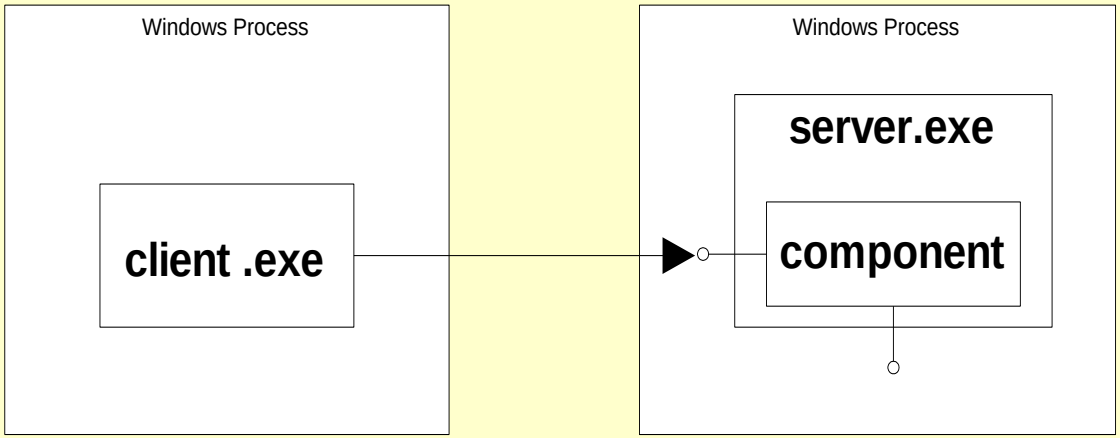
- If client and component do not share the same threading model then client gets a pointer to a proxy for marshaled communication with the component.

Proxy Communication for STA



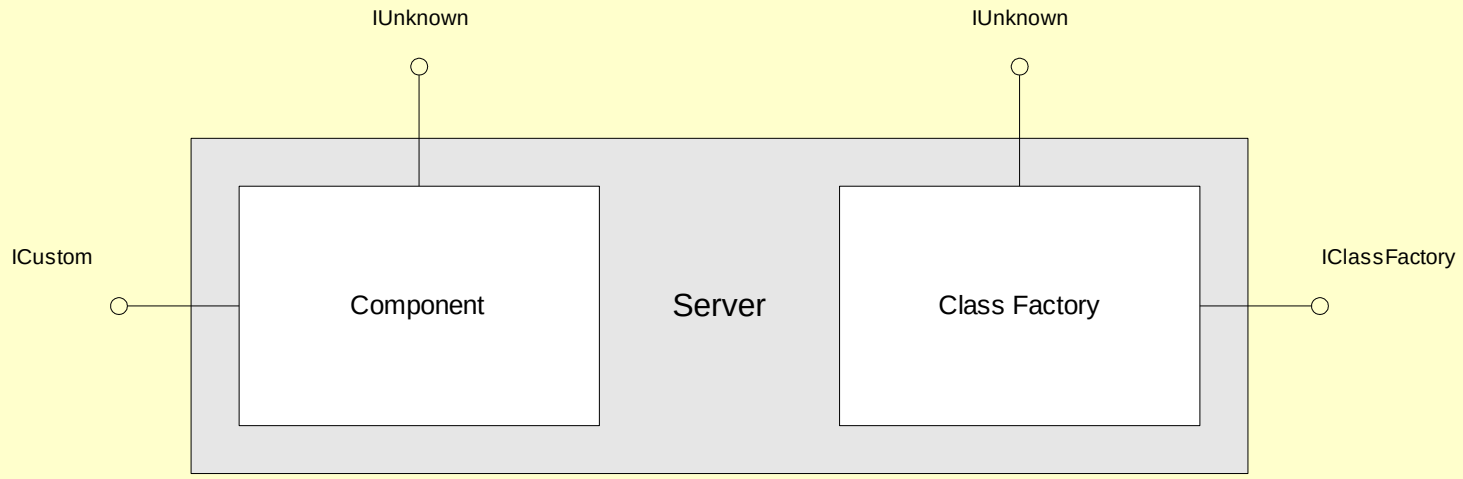
Out-of-Proc Components

- Client always gets a pointer to proxy for marshaled communication with the component.
- Client and Component reside in different processes.
- Component by default has no visible window.



COM Objects

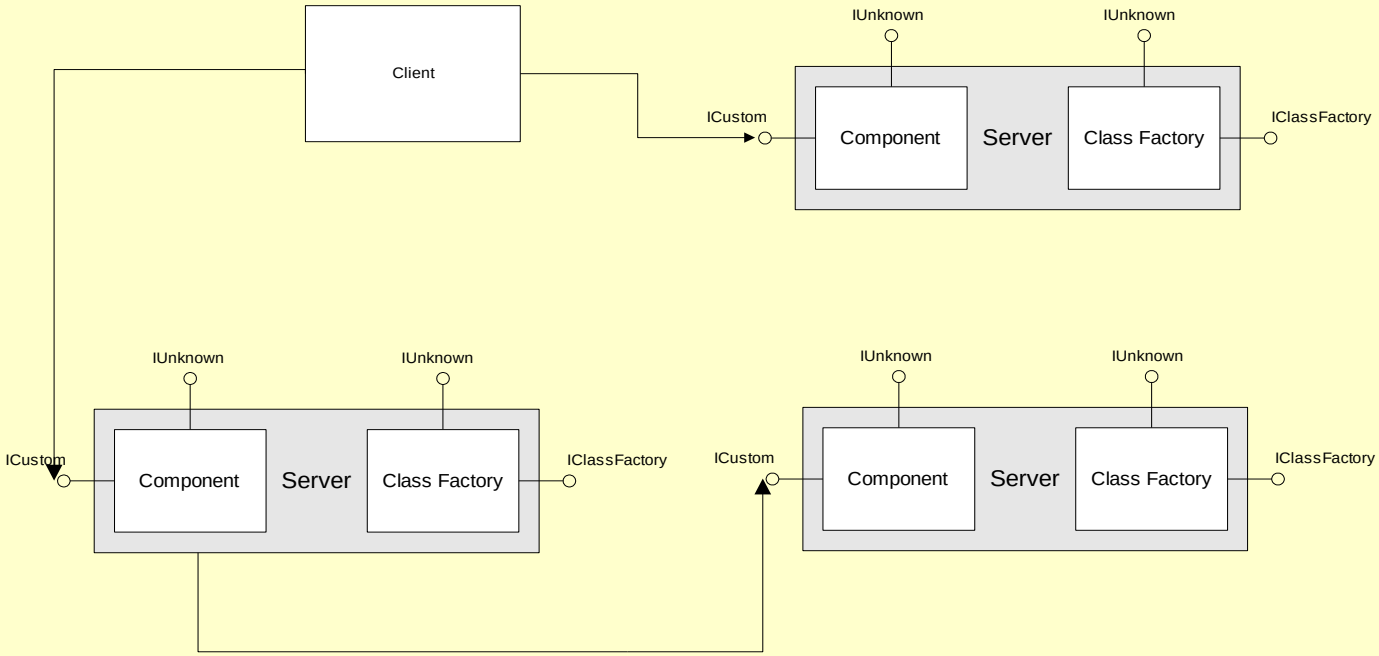
- COM objects are wrapped in dll or exe servers.
- Each server has a class factory, called by COM, to build an instance of the object for client use.



- Class factory and component expose interfaces for use by COM and clients.

COM Programs

- clients request COM to load servers, instantiate their components and return pointers to the component interfaces.



COM Interfaces

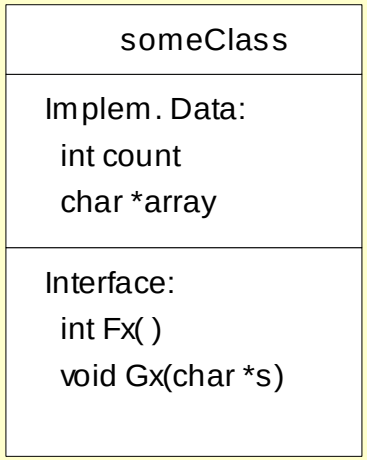
- Interfaces play a critical role in the COM architecture.
  - Interfaces declare the services provided by their components.
  - They support C++ strong type checking and provide a logical view of the component's activities.
  - The interfaces a component exposes are the only means of access they provide to clients.
  - COM interfaces are like C++ class public interfaces, but they do not provide implementations for the functions they declare.
  - They are often implemented with C++ abstract base classes.

- This means that reuse of software implementation through inheritance is not supported by COM.

C++ Interfaces

- C++ defines an interface as a set of public member functions of some class:

```
class someClass {
  public:
    someClass();
    ~someClass();
    int  Fx();
    void Gx(char *s);
  private:
    int  Count;
    char *Array;
};
```



- Note that implementation details are not accessible to clients, but also, not hidden either. So clients have compilation dependence on implementation.

C++ Inheritance

- C++ classes can be composed through inheritance. A derived class inherits the public interface and any implementation provided by the base class.

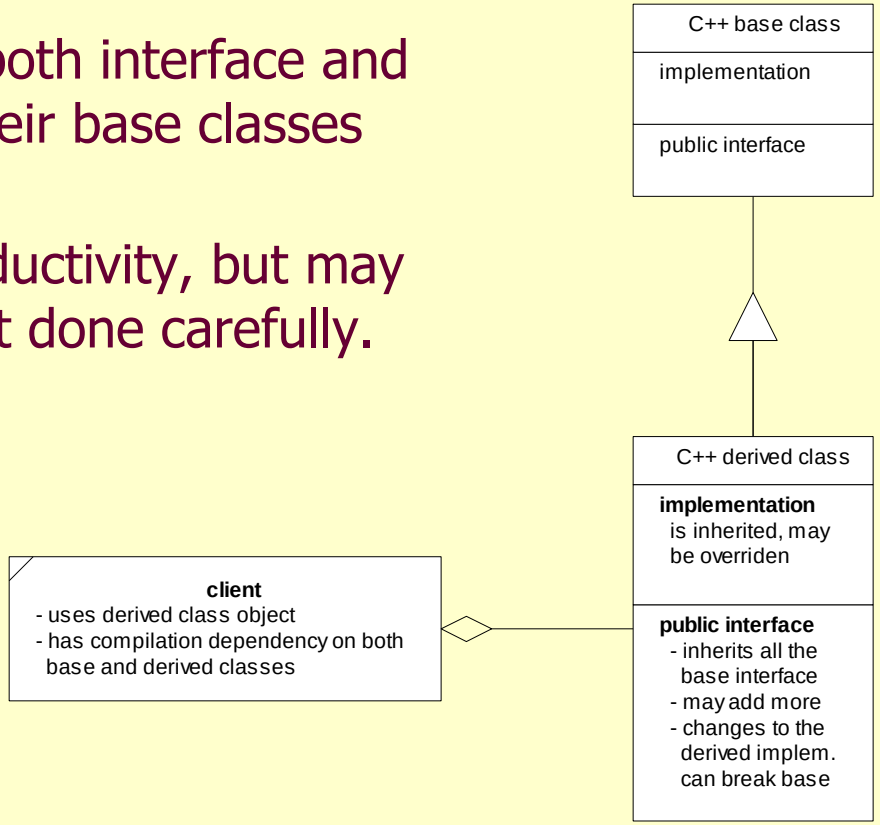
```
class baseClass { public: virtual rType operation(); … };
class derivedClass : public baseClass { public: … };
```

- derivedClass inherits the base member operation() and any implementation that baseClass has provided.

- derivedClass is free to override virtual operations provided by its base, but is not required to do so.

Reusing Software Implementations

- C++ classes inherit both interface and implementation of their base classes
- This reuse helps productivity, but may cause breakage if not done carefully.



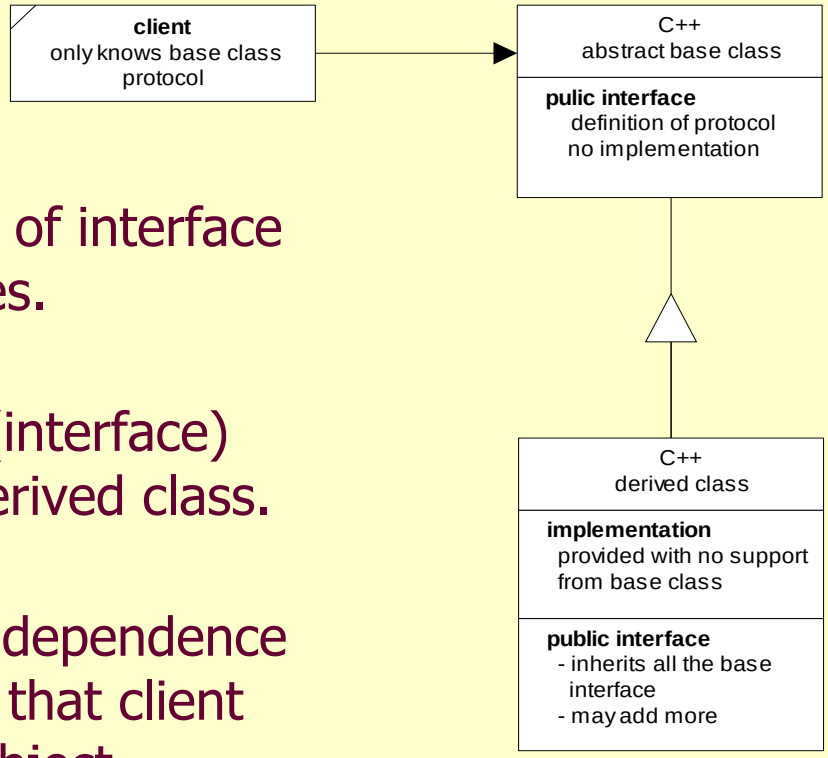
Abstract Base Classes

- A C++ class can provide an interface without providing an implementation if it is constructed as an abstract class, e.g.:

```
class anAbstrBase {
  public:
    virtual bool operation( ) = 0;   // pure virtual function
     :                               //   no definition provided
};

class getsBaseInterface : public anAbstrBase {
  public:
    virtual bool operation( );     // definition must be provided
      :                            //   in implementation body so
};                                 //   objects can be created
```

Inheriting only Interfaces



- C++ supports inheritance of interface using abstract base classes.
- Clients hold a base class (interface) pointer attached to the derived class.
- Client has no compilation dependence on derived class provided that client does not instantiate the object.

Abstract Classes are Interfaces

- Non-abstract base class
  - no pure virtual functions in either base or derived
  - base class has:
    - data members
    - constructors
    - destructor
  - base class implements all its member functions
  - derived class inherits most base class members.
  - derived class may override an inherited virtual member function implementation but does not have to do so
- Abstract base class
  - base must have at lease one pure virtual function
  - base class usually has no members or constructor
  - should provide virtual destructor
  - it simply defines a protocol consisting of all pure virtual member functions
  - derived class must implement every pure virtual function
  - clients can use protocol on any derived object through a base class pointer, e.g., an interface pointer

COM Interface Policy

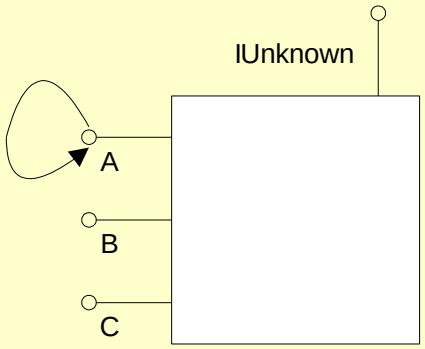
- COM defines a fundamental interface named IUnknown with three methods:
  - QueryInterface, used by clients to inquire if a component supports other specific interfaces.
  - AddRef, used by COM to increment a reference count maintained by all COM objects
  - Release, used by clients to decrement the reference count when finished with interface. When the reference count decrements to zero the object's server is unloaded from memory.
- All COM interfaces must declare the IUnknown methods, usually done by inheriting from IUnknown.
- All COM objects are required to implement the IUnknown interface along with their own operations.

COM Interface Policy

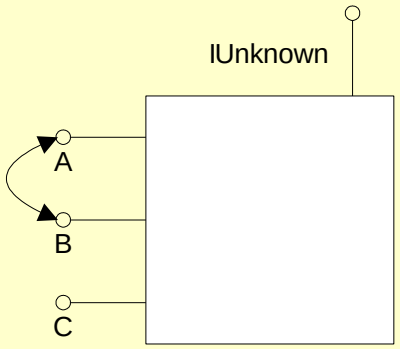
- COM requires that:
  - all calls to QueryInterface for a given interface must return the same pointer value
  - the set of interfaces accessible from QueryInterface must be fixed
  - if a client queries for an interface through a pointer to that interface the call must succeed
  - if a client using a pointer for one interface successfully queries for a second interface the client must be able to successfully query through the second interface pointer for the first interface
  - if a client successfully queries for a second interface and, using that interface pointer successfully queries for a third interface, then a query using the first interface pointer for the third interface must also succeed.

COM Interface Policy

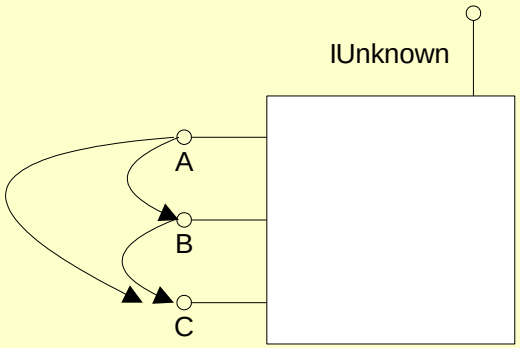
Symmetric



Reflexive



Transitive



COM Configuration Management

- COM objects and their interfaces are identified by Globally Unique Identifiers (GUIDs). These are 128 bit numbers generated by an algorithm based on machine identity and date.
- COM requires that interfaces are immutable. That is, once an interface is published it will never change. A component may change its implementation, removing latent errors or improving performance, but interface syntax and semantics must be fixed.
- Components may add new functionality, expressed by additional interfaces, but the component must continue to support its orig-inal interface set.

COM Class Factories

- A COM class object is a component that creates new instances of other objects.
- Class objects implement the IClassFactory interface and are called class factories. IClassFactory interface has two methods:
  - CreateInstance accepts an interface identity number and returns a pointer, if successful, to a new component object.
  - LockServer turns on, or off, locking of the factory's server in memory.
- COM instantiates factories using a global or static member function provided by the factory code:

```
DllGetClassObject(REFCLSID clsid, REFIID riid, void **ppv)
```

Standard COM Interfaces

- IClassFactory is used by COM to create instances of the component

IClassFactory
CreateInstance(IUnknown *pUnknownOuter, REFIID riid, void **ppv) LockServer(BOOL bLock)

- IUnknown is used by Clients to get interface pointers

IUknown
QueryInterface(REFIID riid, void **ppv) Addref( ) Release( )

- There are many other standard COM interfaces.

Standard COM interfaces

File OBJIDL.H: - interfaces used by COM and Windows 2000 for interprocess communication, etc.

interface IMarshal
interface IMalloc
interface IMallocSpy
interface IStdMarshalInfo
interface IExternalConnection
interface IMultiQI
interface IEnumUnknown
interface IBindCtx
interface IEnumMoniker
interface IRunnableObject
interface IRunningObjectTable
interface IPersist
interface IPersistStream
interface IMoniker
interface IROTData
interface IEnumString
interface ISequentialStream
interface IStream
interface IEnumSTATSTG
interface IStorage
interface IPersistFile
interface IPersistStorage
interface ILockBytes
interface IEnumFORMATETC
interface IEnumSTATDATA
interface IRootStorage
interface IAdviseSink
interface IAdviseSink2
interface IDataObject
interface IDataAdviseHolder
interface IMessageFilter
interface IRpcChannelBuffer
interface IRpcChannelBuffer2
interface IRpcChannelBuffer3
interface IRpcProxyBuffer
interface IRpcStubBuffer
interface IPSFactoryBuffer
interface IChannelHook
interface IPropertyStorage
interface IPropertySetStorage
interface IEnumSTATPROPSTG
interface IEnumSTATPROPSETSTG
interface IClientSecurity
interface IServerSecurity
interface IClassActivator
interface IRpcOptions
interface IComBinding
interface IFillLockBytes
interface IProgressNotify
interface ILayoutStorage
interface ISurrogate
interface IGlobalInterfaceTable
interface IDirectWriterLock
interface ISynchronize
interface ISynchronizeMutex
interface IAsyncSetup
interface ICancelMethodCalls
interface IAsyncManager
interface IWaitMultiple
interface ISynchronizeEvent
interface IUrlMon
interface IClassAccess
interface IClassRefresh
interface IEnumPackage
interface IEnumClass
interface IClassAdmin

Standard COM Interfaces

File OLEIDL.H: - interfaces used for containers like Word and Viso

interface IOleAdviseHolder
interface IOleCache
interface IOleCache2
interface IOleCacheControl
interface IParseDisplayName
interface IOleContainer
interface IOleClientSite
interface IOleObject
interface IOleWindow
interface IOleLink
interface IOleItemContainer

interface IOleInPlaceUIWindow
interface IOleInPlaceActiveObject
interface IOleInPlaceFrame
interface IOleInPlaceObject
interface IOleInPlaceSite
interface IContinue
interface IViewObject
interface IViewObject2
interface IDropSource
interface IDropTarget
interface IEnumOLEVERB

Standard COM Interfaces

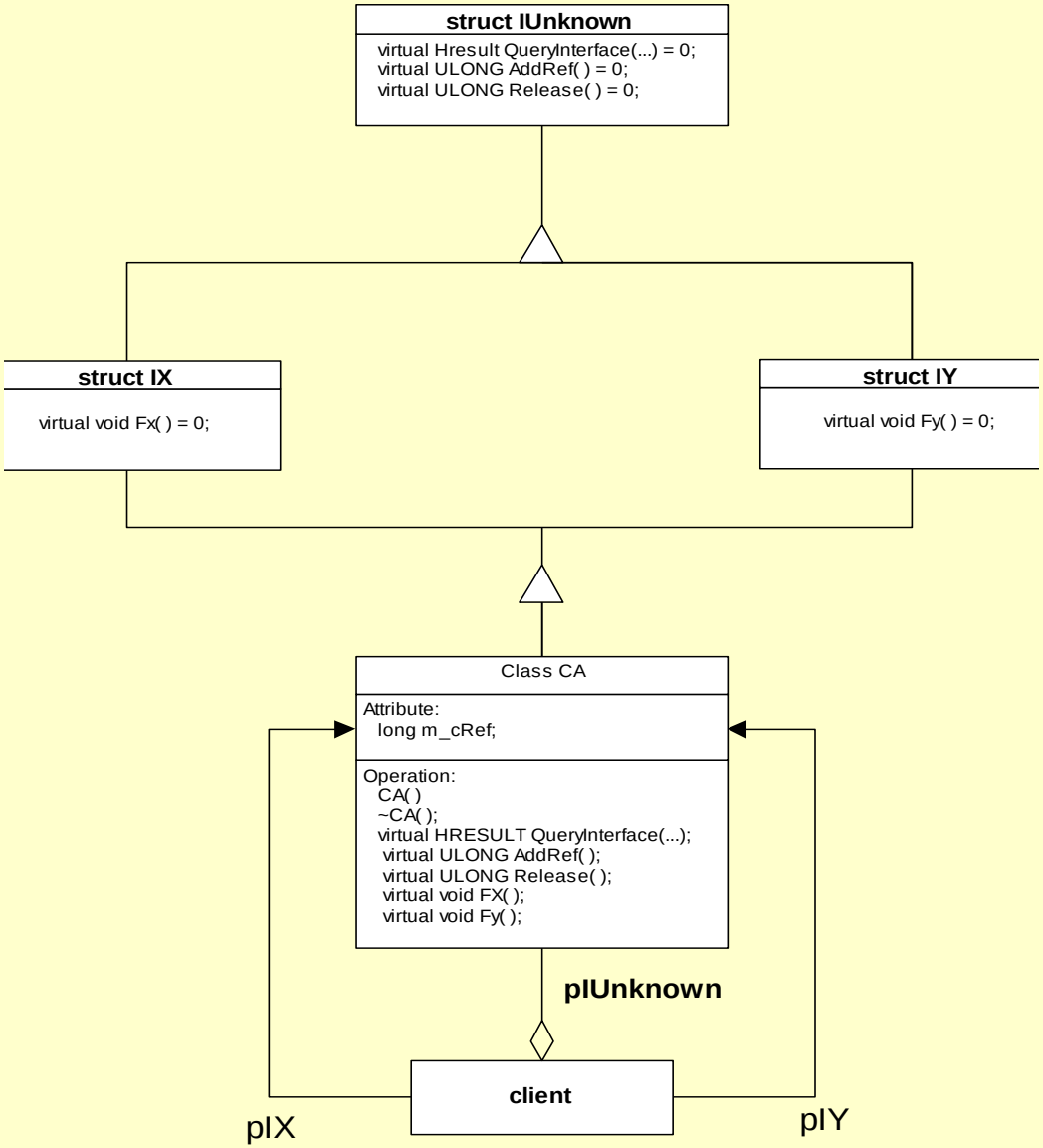
File OCIDL.H: - interfaces used for controls like ListBoxEx and WebBrowser

interface IEnumConnections
interface IConnectionPoint
interface IEnumConnectionPoints
interface IConnectionPointContainer
interface IClassFactory2
interface IProvideClassInfo
interface IProvideClassInfo2
interface IProvideMultipleClassInfo
interface IOleControl
interface IOleControlSite
interface IPropertyPage
interface IPropertyPage2
interface IPropertyPageSite
interface IPropertyNotifySink
interface ISpecifyPropertyPages
interface IPersistMemory
interface IPersistStreamInit
interface IPersistPropertyBag
interface ISimpleFrameSite
interface IFont

interface IPicture
interface IFontEventsDisp
interface IFontDisp
interface IPictureDisp
interface IOleInPlaceObjectWindowless
interface IOleInPlaceSiteEx
interface IOleInPlaceSiteWindowless
interface IViewObjectEx
interface IOleUndoUnit
interface IOleParentUndoUnit
interface IEnumOleUndoUnits
interface IOleUndoManager
interface IPointerInactive
interface IObjectWithSite
interface IErrorLog
interface IPropertyBag
interface IPerPropertyBrowsing
interface IPropertyBag2
interface IPersistPropertyBag2
interface IAdviseSinkEx
interface IQuickActivate

A Concrete Example

- In the diagram on the next page, we show a COM component that implements two interfaces, IX and IY.
- The client gets a pointer to IUnknown from COM's CoCreateInstance function. That pointer can only be used to access the three IUnknown functions, QueryInterface, AddRef, and Release.
- The client uses QueryInterface to get a pointer to one of the interfaces, say IX. That pointer can only be used to access the functions exposed by IX, in this case just the function Fx().

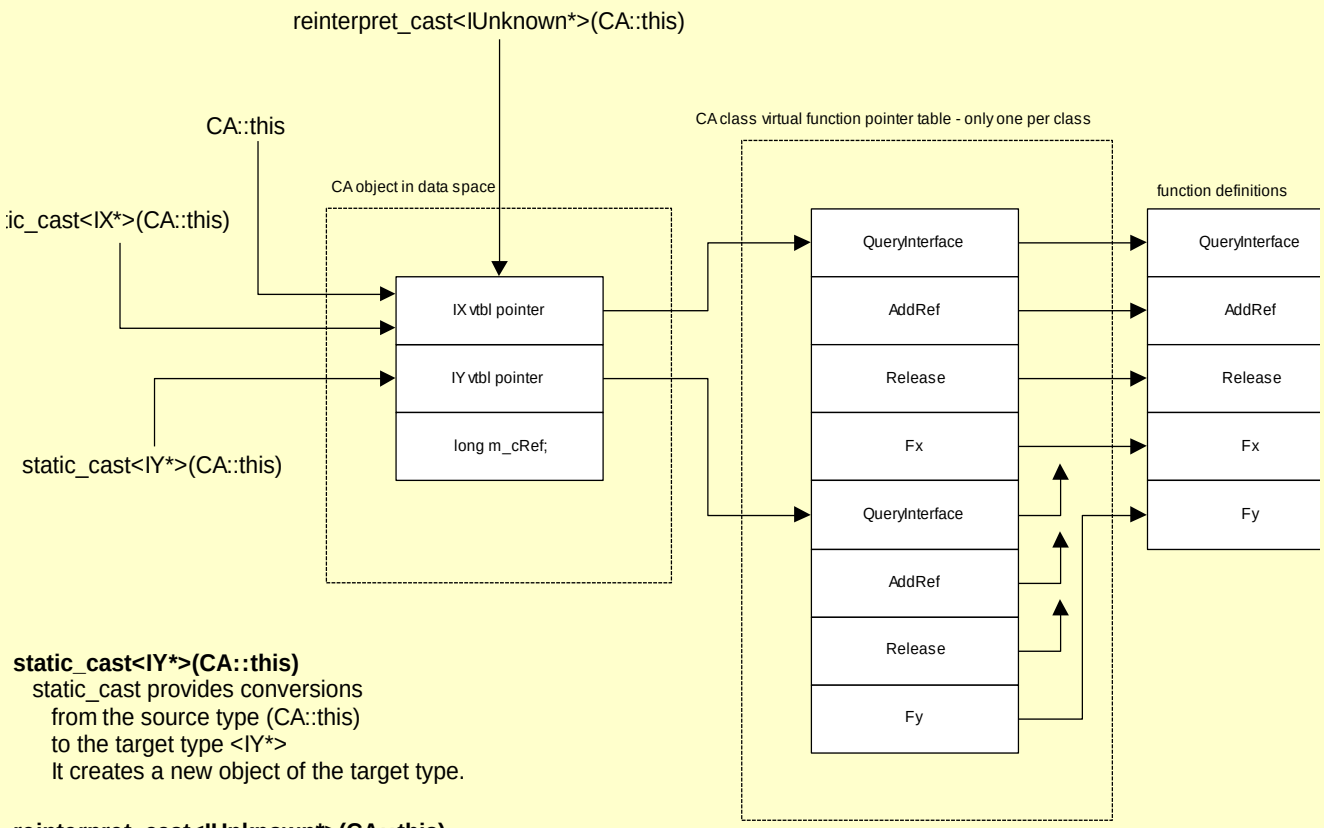


pointers pIunknown, pIX, and pIY
can be declared using only
eclarations for Iunknown, IX, and IY

Vtable Layout

- On the next page, you see a diagram that illustrates how the C++ interfaces connect to the code that implements them.
- The IUnknown pointer points to a table of function pointers called the vtable (for virtual function pointer table). It can only access the QueryInterface, AddRef, and Release functions.
- QueryInterface returns, using the casts shown in the diagram, a pointer to the requested interface, either IX or IY, or NULL if the request can't be satisfied.

Vtable



static_cast<IY*>(CA::this)
static_cast provides conversions
from the source type (CA::this)
to the target type <IY*>
It creates a new object of the target type.

reinterpret_cast<IUnknown*>(CA::this)
reinterpret_cast interprets the same
bit pattern as belonging to a different type
No new object is created.

Reusing Implementations

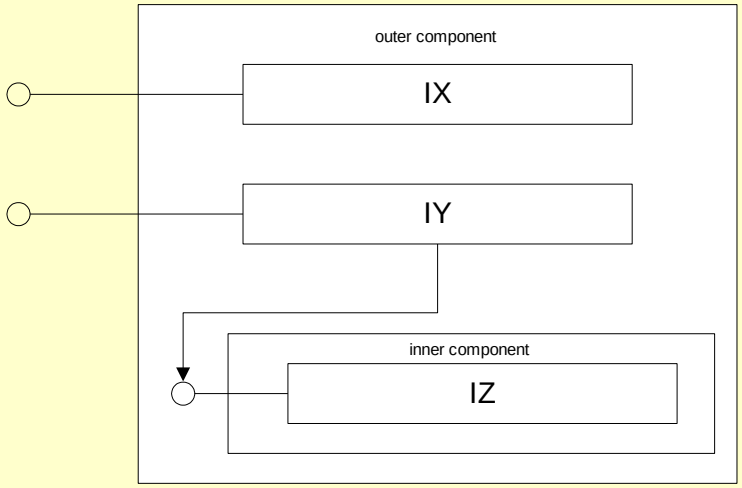
- There are a lot of existing interfaces that we want to use without having to re-implement them. How do we do that?

- C++ has four mechanisms for reuse:
  - inheritance of implementation (ruled out by COM due to concerns about software reliability)
  - composition, e.g, using objects of existing classes as data members of the class being implemented
  - Aggregation, e.g., create and use object of existing classes in a member function.
  - templates, shown by the Standard C++ Library to be very powerful

COM's Reuse Mechanisms: Containment

- COM defines component containment which has semantics of C++ aggregation but can be composed at run time.

- With containment the reusing COM object loads an existing component and implements part of its own interface by delegating calls to the contained component.



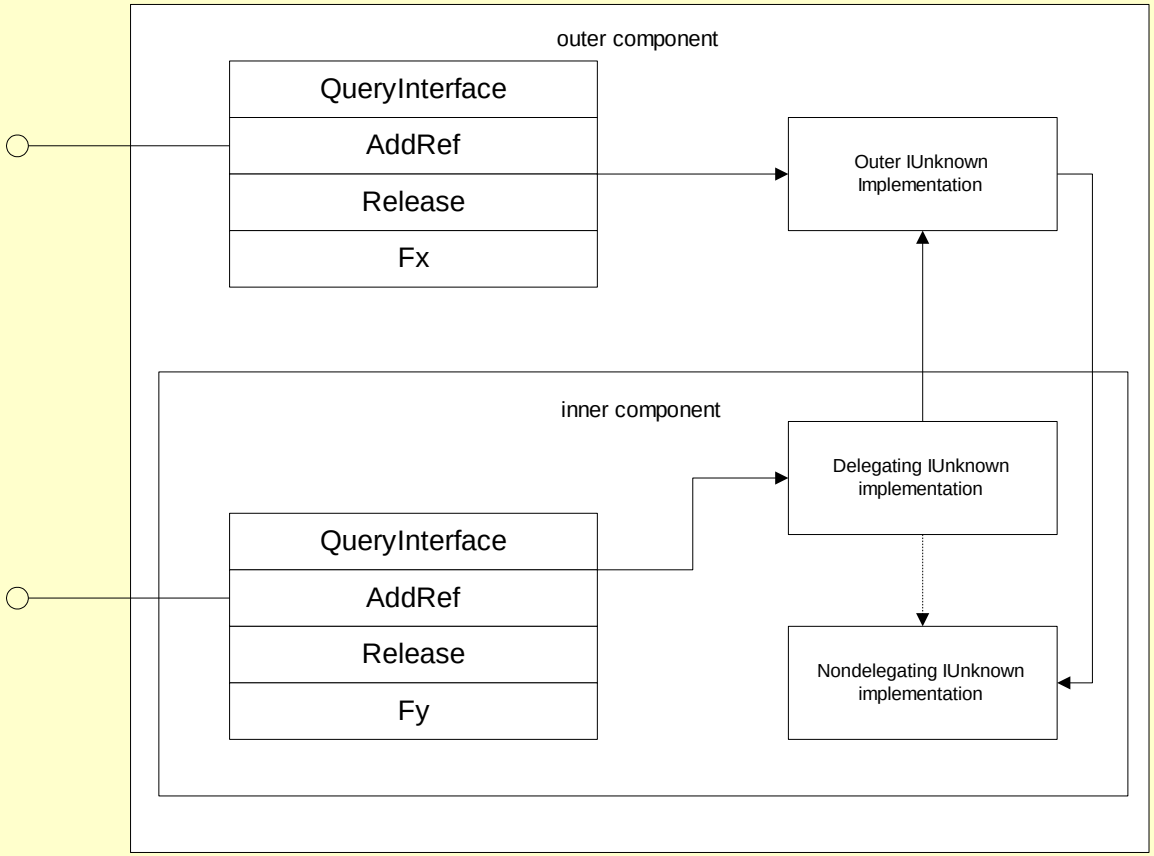
Implementing Containment

- Containing component class:
  - provides an interface matching the contained classes interface and delegates calls to the inner interface (optional).
  - provides an init( ) function which calls CoCreateInstance(…) on the contained component.
  - Declares a pointer member to hold the pointer to inner interface returned by CoCreateInstance(…).
  - Outer component's class factory calls init( ) in CreateInstance(…) function.
- Client:
  - no special provisions.
- Inner Component:
  - no special provisions

COM's Reuse Mechanisms: Aggregation

- What COM chose to define as aggregation is unfortunately quite different than C++ aggregation.
- With COM aggregation the aggregating class forwards interface of the reused existing class to the client by delivering a pointer to the aggregated interface.
  - This complicates implementation of the inner IUnknown since the usual COM policy for interfaces must still be carried out.`
  - The result is that, in order to be aggregate-able a component must implement two IUnknown interfaces

COM Aggregation



Implementing (COM) Aggregation

- Signaling aggregation:
  - CoCreateInstance(…) and IClassFactory::CreateInstance(…) both have a parameter: Iunknown* pUnknownOuter. If this pointer is null the created object will not be aggregated.
  - If An outer component wants to aggregate an inner component it passes its own IUnknown interface pointer to the inner.
- Implementing IUnknown:
  - If an aggregatable component is not being aggregated it uses its non-delegating IUnknown implementation in the usual way.
  - If it is being aggregated it uses its delegating IUnknown to forward requests for IUnknown or outer interface to the outer component. Clients never get a pointer to the inner non-delegating IUnknown. When they ask for IUnknown they get a pointer to the outer IUknown.

Implementing Aggregation

- The delegating IUnknown forwards QueryInterface, AddRef, and Release calls to the outer IUnknown.
- When a client requests an inner interface from an outer interface pointer the outer delegates the query to the inner non-delegating QueryInterface.
- When CoCreateInstance is called by the outer component it passes its IUnknown pointer to the inner and gets back a pointer to the inner IUnknown. This happens in an init( ) function called by the outer's class factory in its CreateInstance function.

COM Architectural Features

- Program to Interfaces - create objects with class factories	- Break compilation dependencies
- implement using dynamic link libraries	- reuse binary code - update without rebuilding
- use registry to locate components - identify components using GUIDS	- clients need no knowledge of where components reside - avoid name clashes with other components
- delegate activation to the OS	- allows components with different threading models to interoperate
- use Remote Procedure Call (RPC) communication and marshalling between processes and machines	- support for distributed architectures, e.g., from OLE linking and embedding to enterprise computing
- use Interface Definition Language (IDL) to describe component's interfaces	- hides some of the ugly code required to handle RPCs

Appendix
In-Process Components

COM component management

Logical and physical program structure

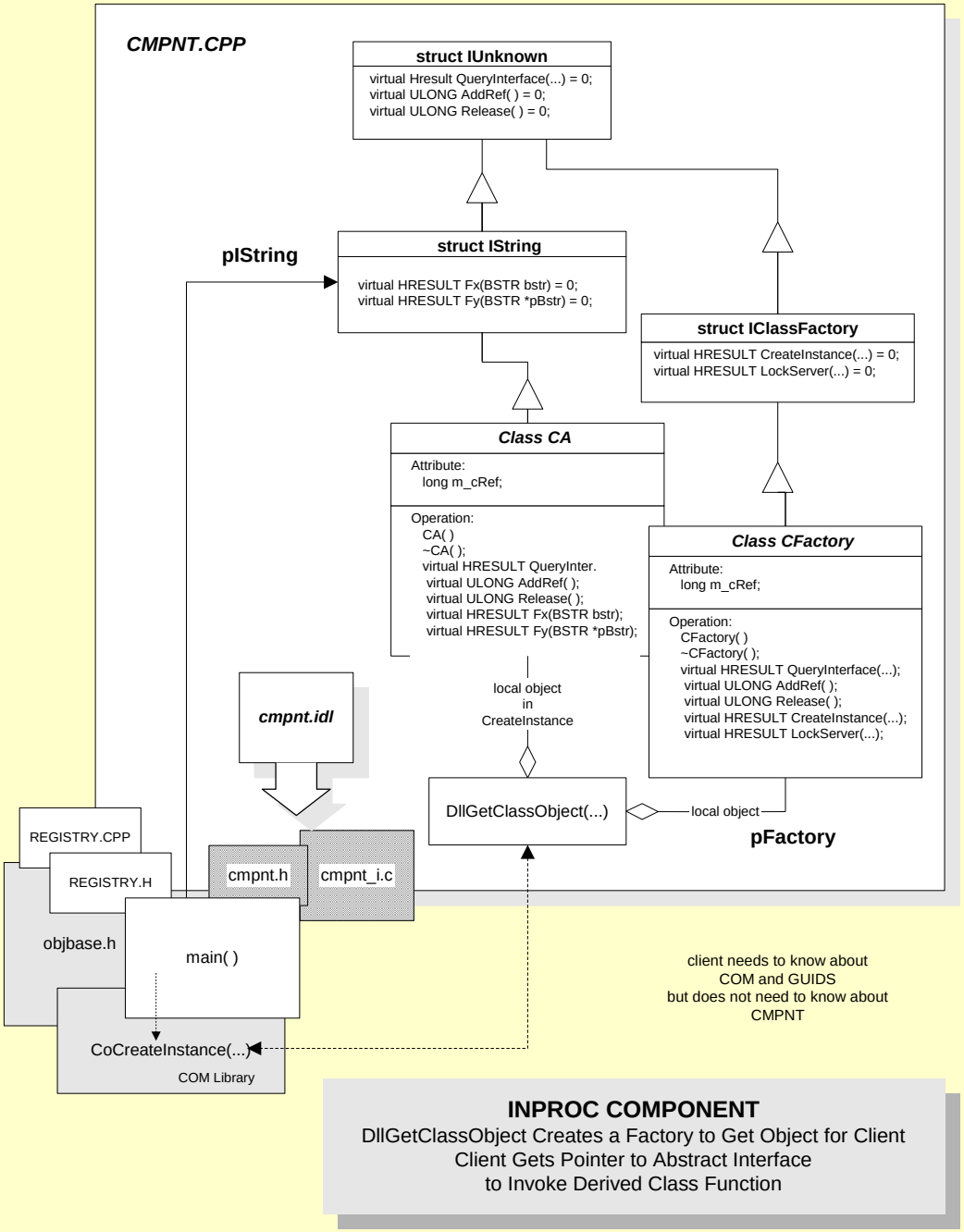
A little code

In-Process Components

- An inproc component is implemented as a dll which the client loads into its own address space, e.g., becomes part of the client process.
- The inproc component provides a class factory so COM can create an instance on behalf of the client. Thus the client has no compilation dependency on the component.
- The component also declares its interfaces, IX and IY, and implements them with the class CA.
- The component also provides four functions in the dll to support its activities:
  - Dllmain() simply saves a handle to the process
  - DLLRegisterServer() calls function in Registry module to register comp.
  - DllUnRegisterServer() calls function in Registry module to unregister comp.
  - DllCanUnloadNow() tells come that client is done with dll
  - DllGetClassObject() called by COM to get a pointer to the class factory instance.

Inproc Component Architecture

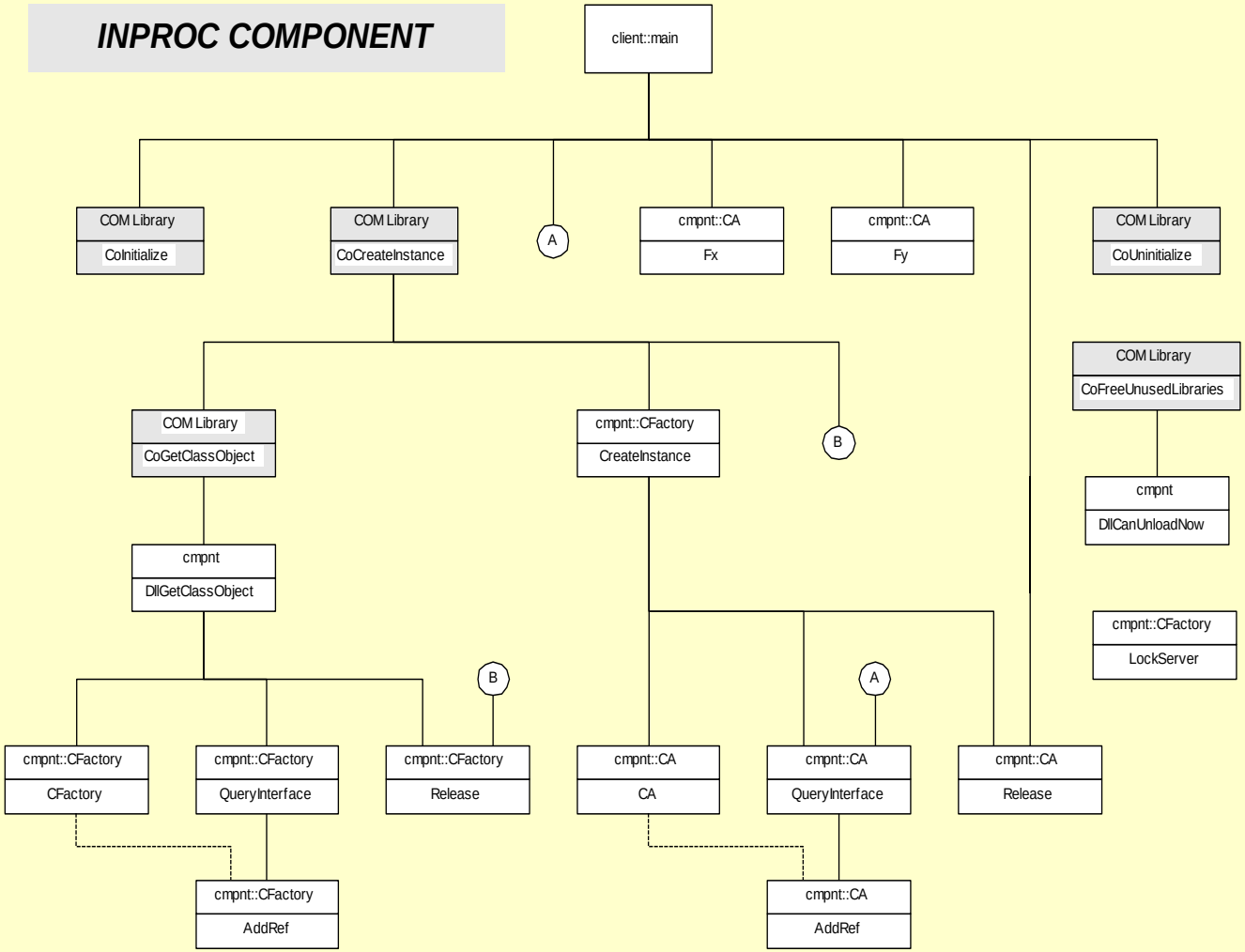
- The structure of the inproc component is shown by the architectural diagram on the next page. The diagram shows:
  - Interfaces, IX and IY, declared by the component
  - class factory and class that implements the interfaces
  - DllGetClassFactory function
  - Registry module (no details) that is responsible for writing the path to the component into the registry.
  - declarations of the interfaces in IFACE.H (no implementation details) used by both client and component.
  - Definitions of constant GUIDs in IFACE.CPP used by both client and component.
  - COM library, exposed by the declarations in objbase.h


CMPNT.CPP
struct IUnknown
virtual Hresult QueryInterface(...) = 0;
virtual ULONG AddRef( ) = 0;
virtual ULONG Release( ) = 0;
pIString
struct IString
virtual HRESULT Fx(BSTR bstr) = 0;
virtual HRESULT Fy(BSTR *pBstr) = 0;
struct IClassFactory
virtual HRESULT CreateInstance(...) = 0;
virtual HRESULT LockServer(...) = 0;
Class CA
Attribute:
long m_cRef;
Operation:
CA( )
~CA( );
virtual HRESULT QueryInter.
virtual ULONG AddRef( );
virtual ULONG Release( );
virtual HRESULT Fx(BSTR bstr);
virtual HRESULT Fy(BSTR *pBstr);
Class CFactory
Attribute:
long m_cRef;
Operation:
CFactory( )
~CFactory( );
virtual HRESULT QueryInterface(...);
virtual ULONG AddRef( );
virtual ULONG Release( );
virtual HRESULT CreateInstance(...);
virtual HRESULT LockServer(...);
cmpnt.idl
local object
in
CreateInstance
DllGetClassObject(...)
local object
pFactory
REGISTRY.CPP
REGISTRY.H
cmpnt.h
cmpnt_i.c
objbase.h
main( )
CoCreateInstance(...)
COM Library
client needs to know about
COM and GUIDS
but does not need to know about
CMPNT
INPROC COMPONENT
DllGetClassObject Creates a Factory to Get Object for Client
Client Gets Pointer to Abstract Interface
to Invoke Derived Class Function

Inproc Structure Chart

- The diagram on the next page is a structure chart. It shows calling relationships between functions in the client, server, and COM.
- In this diagram, callers are always above callees.
- The diagram shows clearly how the client calls COM to get a pointer to an interface.
- COM calls DllGetClassObject to create the class factory, then uses the pointer it gets to create an instance of the CA class that implements the interfaces.
- COM then passes the pointer to the interface back to the client for its use.
- The client finally calls release to tell COM its done with the component.

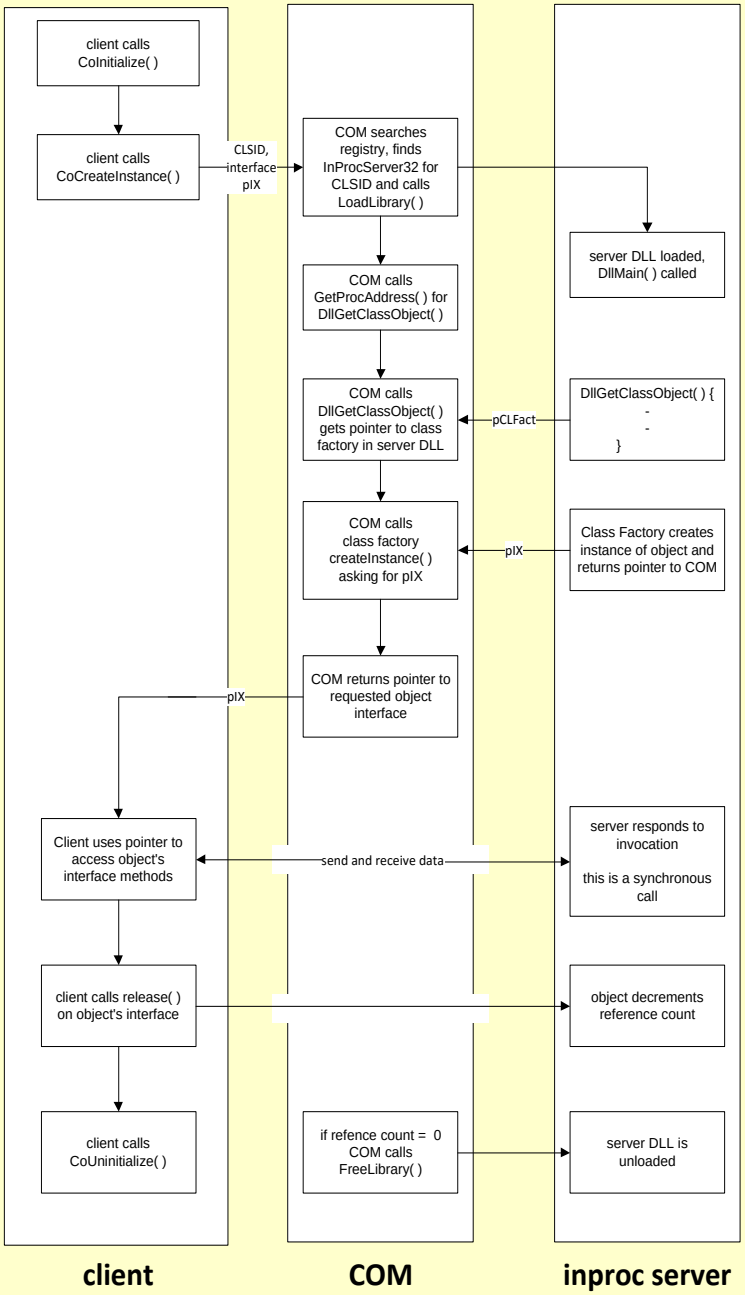
Structure chart



Note: calling sequence goes left to right in this diagram

Activation Diagram

- The diagram on the next page is an elaborated event trace diagram.
- It shows the separate, cooperating actions of the client, COM, and the Component to:
  - Create the component instance
  - Use it
  - Shut it down


client calls CoInitialize( )
client calls CoCreateInstance( )
CLSID, interface pIX
COM searches registry, finds InProcServer32 for CLSID and calls LoadLibrary( )
server DLL loaded, DllMain( ) called
COM calls GetProcAddress( ) for DllGetClassObject( )
COM calls DllGetClassObject( ) gets pointer to class factory in server DLL
pCLFact
DllGetClassObject( ) { - - }
COM calls class factory createInstance( ) asking for pIX
pIX
Class Factory creates instance of object and returns pointer to COM
COM returns pointer to requested object interface
pIX
Client uses pointer to access object's interface methods
send and receive data
server responds to invocation
this is a synchronous call
client calls release( ) on object's interface
object decrements reference count
client calls CoUninitialize( )
if refence count = 0 COM calls FreeLibrary( )
server DLL is unloaded
client
COM
inproc server

Code Samples

- You will find a sample of an inproc COM component, written in C++ without using the MFC or ATL libraries in the CSE775/code/inproc_Ex1 directory.
- This code is illustrated by the Class Diagram, Structure Chart, and Activation diagram on the previous slides.
- Looking carefully at this code, with these three diagrams close at hand will help you understand the details of how an in-process COM component works.