Fundamentals Of COM(+) (Part 1)

Don Box
Cofounder
DevelopMentor
http://www.develop.com/dbox

[redacted]

11-203

COM – The Idea

- COM is based on three fundamental ideas
- Clients program in terms of interfaces, not classes
- Implementation code is not statically linked, but rather loaded on-demand at runtime
- Object implementors declare their runtime requirements and the system ensures that these requirements are met
- The former two are the core of classic COM
- The latter is the core of MTS and COM+

Tale Of Two COMs

- COM is used primarily for two tasks
- Task 1: Gluing together multiple components inside a process
  - Class loading, type information, etc
- Task 2: Inter-process/Inter-host communications
  - Object-based Remote Procedure Calls (ORPC)
- Pros: Same programming model and APIs used for both tasks
- Cons: Same programming model and APIs used for both tasks
- Design around the task at hand

Motivation

- We want to build dynamically composable systems
  - Not all parts of application are statically linked
- We want to minimize coupling within the system
  - One change propagates to entire source code tree
- We want plug-and-play replaceablity and extensibility
  - New pieces should be indistinguishable from old, known parts
- We want freedom from file/path dependencies
  - xcopy /s *.dll C:\winnt\system32 not a solution
- We want components with different runtime requirements to live peaceably together
  - Need to mix heterogeneous objects in a single process

A Solution – Components

- Circa-1980's style object-orientation based on classes and objects
  - Classes used for object implementation
  - Classes also used for consumer/client type hierarchy
- Using class-based OO introduces non-trivial coupling between client and object
  - Client assumes complete knowledge of public interface
  - Client may know even more under certain languages (e.g., C++)
- Circa-1990's object orientation separates client-visible type system from object-visible implementation
  - Allows client to program in terms of abstract types
  - When done properly, completely hides implementation class from client

Recall: Class-Based oo

- The object implementor defines a class that…
  - Is used to produce new objects
  - Is used by the client to instantiate and invoke methods

```
// faststring.h – seen by client and object implementor
class FastString {
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString();
  int Length() const;
  int Find(const char *pszSearchString) const;
};
```

```
// faststring.cpp – seen by object implementor only
FastString::FastString(const char *psz)
  :    :    :
```

Recall: Class-Based oo

- Client expected to import full definition of class
  - Includes complete public signature at time of compilation
  - Also includes size/offset information under C++

```
// client.cpp
// import type definitions to use object
#include "faststring.h"
int FindTheOffset( ) {
  int i = -1;
  FastString *pfs = new FastString("Hello, World!");
  if (pfs) {
    i = pfs->Find("o, W");
    delete pfs;
  }
  return i;
}
```

Class-Based OO Pitfalls

- Classes not so bad when the world is statically linked
  - Changes to class and client happen simultaneously
  - Problematic if existing public interface changes…
- Most environments do a poor job at distinguishing changes to public interface from private details
  - Touching private members usually triggers cascading rebuild
- Static linking has many drawbacks
  - Code size bigger
  - Can't replace class code independently
- Open Question: Can classes be dynamically linked?

Classes Versus Dynamic Linking

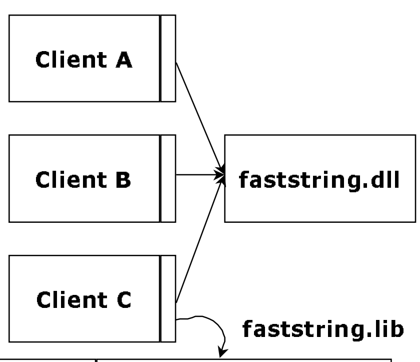
- Most compilers offer a compiler keyword or directive to export all class members from DLL
  - Results in mechanical change at build/run-time
  - Requires zero change to source code (except introducing the directive)

```
// faststring.h

class __declspec(dllexport) FastString {
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString();
  int Length() const;
  int Find(const char *pszSearchString) const;
};
```

Classes Versus Dynamic Linking

- Clients statically link to import library
  - Maps symbolic name to DLL and entry name
- Client imports resolved at load time
- Note: C++ compilers non-standard wrt DLLs
  - DLL and clients must be built using same compiler/linker



import name	*file name*	*export name*
??@3fFastString_6Length	*faststring.dll*	*??@3fFastString_6Length*
??@3fFastString_4Find	*faststring.dll*	*??@3fFastString_4Find*
??@3fFastString_ctor@sz2	*faststring.dll*	*??@3fFastString_ctor@sz2*
??@3fFastString_dtor	*faststring.dll*	*??@3fFastString_dtor*

Classes Versus Dynamic Linking: Evolution

- Challenge: Improve the performance of Length!
  - Do not change public interface and break encapsulation

```
// faststring.h
class FastString {
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString();
  int Length() const;
  int Find(const char *pszSearchString) const;
};
```

```
// faststring.cpp
#include "faststring.h"
#include <string.h>

int FastString::Length() const {
  return strlen(m_psz);
}
```

Classes Versus Dynamic Linking: Evolution

- Solution: Speed up FastString::Length by caching length as data member

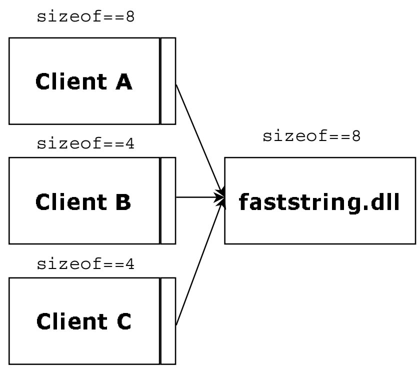
```
class __declspec(dllexport) FastString
{
  char *m_psz;
  int   m_cch;
public:
  FastString(const char
  ~FastString();
  int Length() const;
  int Find(const char *p
};
```

```
FastString::FastString(const char *sz)
: m_psz(new char[strlen(sz)+1]),
  m_cch(strlen(sz)) {
  strcpy(m_psz, sz);
}

int FastString::Length() const {
  return m_cch;
}
```

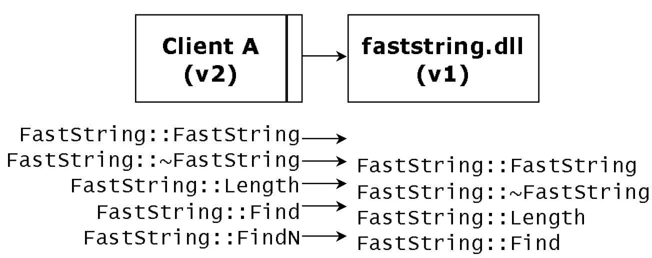
Classes Versus Dynamic Linking: Evolution

- New DLL assumes sizeof(FastString) is 8
- Existing Clients assume sizeof(FastString) is 4
- Clients that want new functionality recompile
- Old Clients break!
- This is an inherent limitation of virtually all C++ environments



Classes Versus Dynamic Linking: Interface Evolution

- Adding new public methods OK when statically linked
  - Class and client code inseparable
- Adding public methods to a DLL-based class dangerous!
  - New client expects method to be there
  - Old DLLs have never heard of this method!!



Conclusions

- Cannot change definition of a data type without massive rebuild/redeployment of client/object
- If clients program in terms of classes, then classes cannot change in any meaningful way
- Classes must change because we can't get it right the first time
- Solution: Clients must not program in terms of classes

Interface-Based Programming

- Key to solving the replaceable component problem is to split the world into two
- The types the client programs against can never change
  - Since classes need to change, these better not be classes!
- Solution based on defining alternative type system based on abstract types called interfaces
- Allowing client to only see interfaces insulates clients from changes to underlying class hierarchy
- Most common C++ technique for bridging interfaces and classes is to use abstract base classes as interfaces

Abstract Bases As Interfaces

- A class can be designated as abstract by making (at least) one method pure virtual

```
struct IFastString {
  virtual int Length( ) const = 0;
  virtual int Find(const char *) const = 0;
};
```

- Cannot instantiate abstract base
  - Can declare pointers or references to abstract bases
- Must instead derive concrete type that implements each pure virtual function
- Classes with only pure virtual functions (no data members, no implementation code) often called pure abstract bases, protocol classes or interfaces

Interfaces And Implementations

- Given an abstract interface, the most common way to associate an implementation with it is through inheritance
  - Class FastString : public IFastString {...};
- Implementation type must provide concrete implementations of each interface method
- Some mechanism needed to create instances of the implementation type without exposing layout
  - Usually takes the form of a creator or factory function
- Must provide client with a way to delete object
  - Since the new operator is not used by the client, it cannot call the delete operator

Exporting Via Abstract Bases

```
// faststringclient.h – common header between client/class

// here's the DLL-friendly abstract interface:
struct IFastString {
  virtual void Delete() = 0;
  virtual int Length() const = 0;
  virtual int Find(const char *sz) const = 0;
};

// and here's the DLL-friendly factory function:
extern "C" bool
CreateInstance(const char *pszClassName,  // which class?
               const char *psz,           // ctor args
               IFastString **ppfs);       // the objref
```

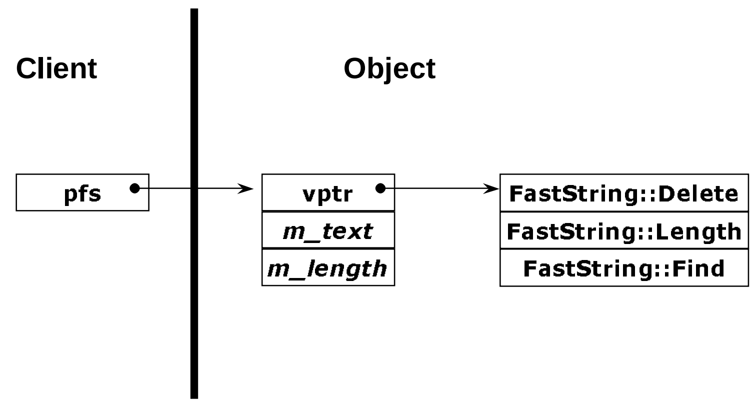
Exporting Via Abstract Bases

```
// faststring.h – private source file of class
#include "faststringclient.h"
class FastString : public IFastString {
// normal prototype of FastString class + Delete
  void Delete() { delete this; }
};
```

```
// component.cpp – private source file for entire DLL
#include "faststring.h"   // import FastString
#include "fasterstring.h" // import FasterString (another class)

bool CreateInstance(const char *pszClassName,
                    const char *psz, IFastString **ppfs) {
  *ppfs = 0;
  if (strcmp(pszClassName, "FastString") == 0)
    *ppfs = static_cast<IFastString*>(new FastString(sz));
  else if (strcmp(pszClassName, "FasterString") == 0)
    *ppfs = static_cast<IFastString*>(new FasterString(sz));
  return *ppfs != 0;
}
```

Exporting Using Abstract Bases



Interfaces And Plug-compatibility

- **Note that a particular DLL can supply multiple implementations of same interface**
  - **CreateInstance(“SlowString”, “Hello!!”, &pfs);**
- **Due to simplicity of model, runtime selection of implementation trivial**
  - **Explicitly load DLL and bind function address**

```
bool LoadAndCreate(const char *szDLL, const char *sz,
                   IFastString **ppfs){
  HINSTANCE h = LoadLibrary(szDLL);
  bool (*fp)(const char*, const char*, IFastString**);
  *((FARPROC*)&fp) = GetProcAddress(h, “CreateInstance”);
  return fp(“FastString”, sz, ppfs);
}
```

Interfaces And Evolution

- Previous slides alluded to interface remaining constant across versions
- Interface-based development mandates that new functionality be exposed using additional interface
  - Extended functionality provided by deriving from existing interface
  - Orthogonal functionality provided by creating new sibling interface
- Some technique needed for dynamically interrogating an object for interface support
  - Most languages support some sort of runtime cast operation (e.g., C++'s dynamic_cast)

Example: Adding Extended Functionality

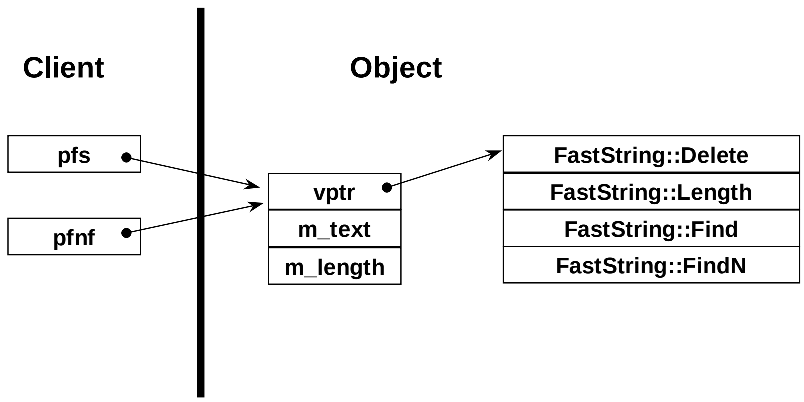
- Add method to find the nth instance of sz

```
// faststringclient.h
struct IFastNFind : public IFastString {
  virtual int FindN(const char *sz, int n) const = 0;
};
```

```
// faststringclient.cxx

int Find10thInstanceOfFoo(IFastString *pfs) {
  IFastNFind *pfnf = 0;
  if (pfnf = dynamic_cast<IFastNFind *>(pfs)) {
    return pfnf->FindN("Foo", 10);
  }
  else
    // implement by hand...
}
```

Example: Adding Extended Functionality



Example: Adding Orthogonal Functionality

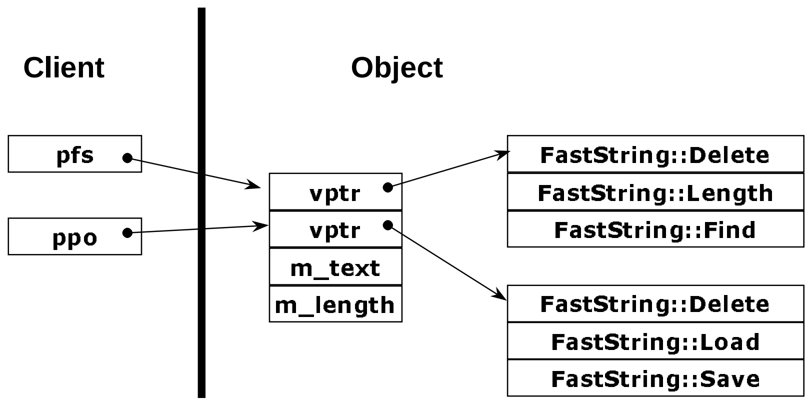
- Add support for generic persistence

```
// faststringclient.h
struct IPersistentObject {
  virtual void Delete(void) = 0;
   virtual bool Load(const char *sz) = 0;
   virtual bool Save(const char *sz) const = 0;
};
```

```
// faststringclient.cxx

bool SaveString(IFastString *pfs) {
  IPersistentObject *ppo = 0;
  if (ppo = dynamic_cast<IPersistentObject*>(pfs)) {
    return ppo->Save("Autoexec.bat");
  }
  else
    return false; // cannot save...
}
```

Example: Adding Orthogonal Functionality



Fixing Interface-Based Programming In C++

- The dynamic_cast operator has several problems that must be addressed
  - 1) Its implementation is non-standard across compilers
  - 2) There is no standard runtime representation for the typename
  - 3) Two parties may choose colliding typenames
- Can solve #1 by adding yet another well-known abstract method to each interface (a la Delete)
- #2 and #3 solved by using a well-known namespace/type format for identifying interfaces
  - UUIDs from OSF DCE are compact (128 bit), efficient and guarantee uniqueness
  - UUIDs are basically big, unique integers!

QueryInterface

- COM programmers use the well-known abstract method (QueryInterface) in lieu of dynamic_cast

```
virtual HRESULT _stdcall
QueryInterface(REFIID riid,// the requested UUID
void **ppv // the resultant objref
) = 0;
```

- Returns status code indicating success (S_OK) or failure (E_NOINTERFACE)
- UUID is integral part of interface definition
  - Defined as a variable with IID_ prefixed to type name
  - VC-specific __declspec(uuid) conjoins COM/C++ names

QueryInterface As A Better Dynamic Cast

```
void UseAsTelephone(ICalculator *pCalc) {
  ITelephone *pPhone = 0;
  pPhone = dynamic_cast<ITelephone*>(pCalc);
  if (pPhone) {
    // use pPhone
   :     :    :
```

```
void UseAsTelephone(ICalculator *pCalc) {
  ITelephone *pPhone = 0;
  HRESULT hr = pCalc->QueryInterface(IID_ITelephone,
                                   (void**)&pPhone);
  if (hr == S_OK) {
    // use pPhone
    :     :    :
```

Fixing Interface-Based Programming In C++

- Previous examples used a “Delete” method to allow client to destroy object
  - Requires client to remember which references point to which objects to ensure each object deleted exactly once

```
ICalculator *pCalc1 = CreateCalc();
ITelephone *pPhone1 = CreatePhone();
ICalculator *pCalc2 = dynamic_cast<ICalculator*>(pPhone1);
ICalculator *pCalc3 = CreateCalc();

pPhone1->Dial(pCalc1->Add(pCalc2->Add(pCalc3->Add(2))));

pCalc1->Delete();  // assume interfaces have Delete
pCalc2->Delete();  // per earlier discussion
pPhone1->Delete();
```

Fixing Interface-Based Programming In C++

- COM solves the “Delete” problem with reference counting
  - Clients blindly “Delete” each reference, not each object
- Objects can track number of extant references and auto-delete when count reaches zero
  - Requires 100% compliance with ref. counting rules
- All operations that return interface pointers must increment the interface pointer’s reference count
  - QueryInterface, CreateInstance, etc.
- Clients must inform object that a particular interface pointer has been destroyed using well-known method
  - Virtual ULONG _stdcall Release( ) = 0;

Reference Counting Basics

```
ICalculator *pCalc1 = CreateCalc();
ITelephone *pPhone1 = CreatePhone();
ICalculator *pCalc2 = 0;
ICalculator *pCalc3 = CreateCalc();
ITelephone  *pPhone2 = 0;
ICalculator *pCalc4 = 0;

pPhone1->QueryInterface(IID_ICalculator,(void**)&pCalc2);
pCalc3->QueryInterface(IID_ITelephone,(void**)&pPhone2);
pCalc1->QueryInterface(IID_ICalculator, (void**)&pCalc4);

pPhone1->Dial(pCalc1->Add(pCalc2->Add(pCalc3->Add(2))));

pCalc1->Release(); pCalc4->Release();
pCalc2->Release(); pPhone1->Release();
pCalc3->Release(); pPhone2->Release();
```

IUnknown

- The three core abstract operations (QueryInterface, AddRef, and Release) comprise the core interface of COM, IUnknown
- All COM interfaces must extend IUnknown
- All COM objects must implement IUnknown

```
extern const IID IID_IUnknown;
struct IUnknown {
  virtual HRESULT STDMETHODCALLTYPE QueryInterface(
                  const IID& riid, void **ppv) = 0;
  virtual ULONG STDMETHODCALLTYPE AddRef( ) = 0;
  virtual ULONG STDMETHODCALLTYPE Release( ) = 0;
};
```

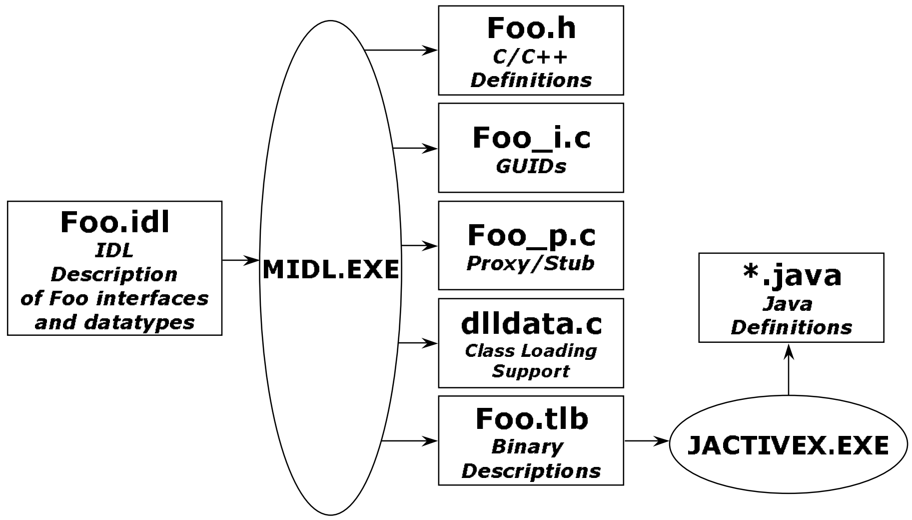
Com Interfaces In Nature

- Represented as pure abstract base classes in C++
  - All methods are pure virtual
  - Never any code, only signature
  - Format of C++ vtable/vptr defines expected stack frame
- Represented directly as interfaces in Java
- Represented as Non-Creatable classes in Visual Basic
- Uniform binary representation independent of how you built the object
- Identified uniquely by a 128-bit Interface ID (IID)

Com Interfaces In Nature

- COM interfaces are described first in COM IDL
- COM IDL is an extension to DCE IDL
  - Support for objects + various wire optimizations
- IDL compiler directly emits C/C++ interface definitions as source code
- IDL compiler emits tokenized type library containing (most) of original contents in an easily parsed format
- Java™/Visual Basic® pick up mappings from type library

COM IDL



COM IDL

- All elements in an IDL file can have attributes
  - Appear in [ ] prior to subject of attributes
- Interfaces are defined at global scope
  - Required by MIDL to emit networking code
- Must refer to exported types inside library block
  - Required by MIDL to emit type library definition
- Can import std interface suite
  - WTYPES.IDL - basic data types
  - UNKNWN.IDL - core type interfaces
  - OBJIDL.IDL - core infrastructure itfs
  - OLEIDL.IDL - OLE itfs
  - OAIDL.IDL - Automation itfs
  - OCIDL.IDL - ActiveX Control itfs

COM IDL

CalcTypes.idl

```
[ uuid(DEFACED1-0229-2552-1D11-ABBADABBAD00), object ]
interface ICalculator : IDesktopDevice {
    import "dd.idl"; // bring in IDesktopDevice
    HRESULT Clear(void);
    HRESULT Add([in] short n);       //   n sent to object
    HRESULT GetSum([out] short *pn); // *pn sent to caller
}
[
  uuid(DEFACED2-0229-2552-1D11-ABBADABBAD00),
  helpstring("My Datatypes")
]
library CalcTypes {
  importlib("stdole32.tlb"); // required
  interface ICalculator;     // cause TLB inclusion
}
```

COM IDL - C++ Mapping

CalcTypes.h

```
#include "dd.h"
extern const IID IID_ICalculator;
struct
__declspec(uuid("DEFACED1-0229-2552-1D11-ABBADABBAD00"))
ICalculator : public IDesktopDevice {
   virtual HRESULT STDMETHODCALLTYPE Clear(void) = 0;
   virtual HRESULT STDMETHODCALLTYPE Add(short n) = 0;
   virtual HRESULT STDMETHODCALLTYPE GetSum(short *pn) = 0;
};
extern const GUID LIBID_CalcTypes;
```

CalcTypes_i.c

```
const IID IID_ICalculator = {0xDEFACED1, 0x0229, 0x2552,
 { 0x1D, 0x11, 0xAB, 0xBA, 0xDA, 0xBB, 0xAD, 0x00 } };
const GUID LIBID_CalcTypes = {0xDEFACED2, 0x0229, 0x2552,
 { 0x1D, 0x11, 0xAB, 0xBA, 0xDA, 0xBB, 0xAD, 0x00 } };
```

COM IDL – Java/VB Mapping

CalcTypes.java

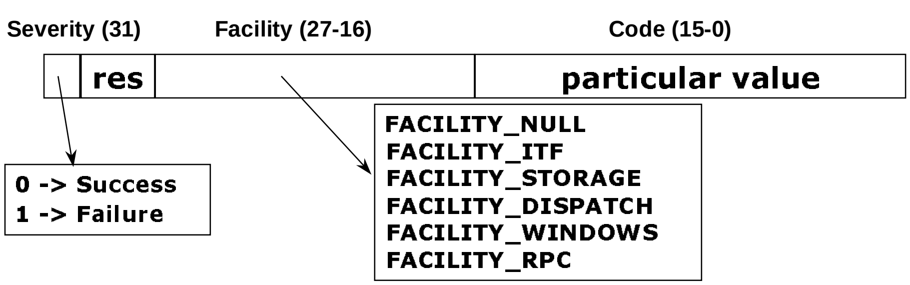
```
package CalcTypes; // library name
/**@com.interface(iid=DEFACED1-0229-2552-1D11-ABBADABBAD00)*/
interface ICalculator extends IDesktopDevice {
    public void Clear( );
    public void Add(short n);
    public void GetSum(short [] pn); // array of length 1
    public static com.ms.com._Guid iid =
      new com.ms.com._Guid(0xDEFACED1, 0x0229, 0x2552,
                           0x1D, 0x11, 0xAB, 0xBA,
                           0xDA, 0xBB, 0xAD, 0x00);
}
```

CalcTypes.cls

```
Public Sub Clear( )
Public Sub Add(ByVal n As Integer)
Public Sub GetSum(ByRef pn As Integer)
```

COM And Error Handling

- COM (today) doesn't support typed C++ or Java-style exceptions
- All (remotable) methods must return a standard 32-bit error code called an HRESULT
  - Mapped to exception in higher-level languages
  - Overloaded to indicate invocation errors from proxies



HRESULTs

- HRESULT names indicate severity and facility
  - <FACILITY>_<SEVERITY>_<CODE>
  - DISP_E_EXCEPTION
  - STG_S_CONVERTED
- FACILITY_NULL codes are implicit
  - <SEVERITY>_<CODE>
  - S_OK
  - S_FALSE
  - E_FAIL
  - E_NOTIMPL
  - E_OUTOFMEMORY
  - E_INVALIDARG
  - E_UNEXPECTED
- Can use FormatMessage API to lookup human-readable description at runtime

COM Data Types

IDL	C++	Java	Visual Basic	Script
small	char	byte	N/A	No
short	short	short	Integer	Yes
long	long	int	Long	Yes
hyper	__int64	long	N/A	No
unsigned small	unsigned char	byte	*Byte*	No
unsigned short	unsigned short	short	N/A	No
unsigned long	unsigned long	int	N/A	No
unsigned hyper	unsigned __int64	long	N/A	No
float	float	float	Single	Yes
double	double	double	Double	Yes
char	char	char	N/A	No
unsigned char	unsigned char	byte	Byte	Yes
wchar_t	wchar_t	char	*Integer*	No

COM Data Types

IDL	C++	Java	Visual Basic	Script
byte	unsigned char	*char*	N/A	No
BYTE	unsigned char	byte	Byte	Yes
boolean	long	int	Long	No
VARIANT_BOOL	VARIANT_BOOL	boolean	Boolean	Yes
BSTR	BSTR	java.lang.String	String	Yes
VARIANT	VARIANT	com.ms.com.Variant	Variant	Yes
CY	long	int	Currency	Yes
DATE	double	double	Date	Yes
enum	enum	int	Enum	Yes
Typed ObjRef	IFoo *	interface IFoo	IFoo	Yes
struct	struct	final class	Type	No
union	union	N/A	N/A	No
C-style Array	array	array	N/A	No

Example

```
struct MESSAGE { VARIANT_BOOL b; long n; };
[ uuid(03C20B33-C942-11d1-926D-006008026FEA), object ]
interface IAnsweringMachine : IUnknown {
  HRESULT TakeAMessage([in] struct MESSAGE *pmsg);
  [propput] HRESULT OutboundMessage([in] long msg);
  [propget] HRESULT OutboundMessage([out, retval] long *p);
}
```

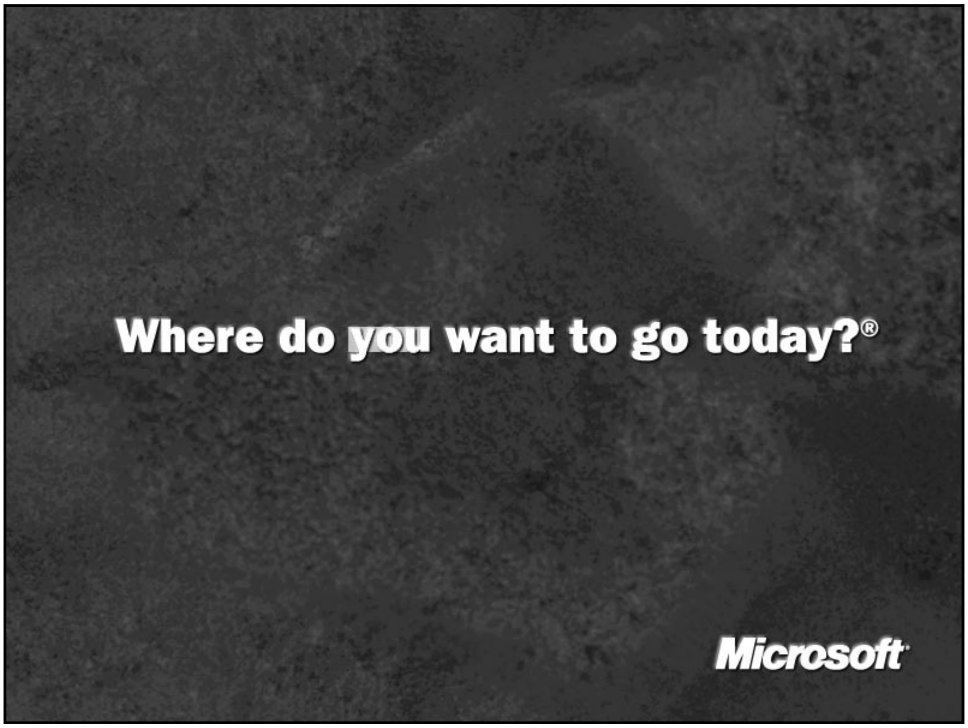
```
public final class MESSAGE {
  public boolean b; public int n;
}
public interface IAnsweringMachine extends IUnknown
{
  public void  TakeAMessage(MESSAGE msg);
  public void putOutboundMessage(int);
  public int getOutboundMessage();
}
```

Where Are We?

- Clients program in terms of abstract data types called interfaces
- Clients can load method code dynamically without concern for C++ compiler incompatibilities
- Clients interrogate objects for extended functionality via RTTI-like constructs
- Clients notify objects when references are duplicated or destroyed
- Welcome to the Component Object Model!

References

- Programming Dist Apps With Visual Basic and COM
  - Ted Pattison, Microsoft Press
- Inside COM
  - Dale Rogerson, Microsoft Press
- Essential COM(+), 2nd Edition (the book)
  - Don Box, Addison Wesley Longman (4Q99)
- Essential COM(+) Short Course, DevelopMentor
  - http://www.develop.com
- DCOM Mailing List
  - http://discuss.microsoft.com


Where do you want to go today?®
Microsoft