Component Object Model

- Language independent
- OS independent (in theory)
- Way to allow components to be designed, deployed, upgraded
  - Need to interact with code written after you were deployed

Immutable interfaces

- Interact through interfaces
  - No direct access to fields
  - Interfaces must never be changed
  - Interfaces assigned a GUID
    - avoid name clashes
    - allow versioning by assigning a new GUID

COM

- A binary compatibility standard
  - interface pointers

Multiple interfaces

- Components can implement multiple interfaces
- Different interfaces may correspond to different entry points to object
  - C++ multiple inheritance
  - adaptors
Interfaces in COM

• Similar to interfaces in Java
  – no variables
• Interfaces have a 128 bit Unique ID
  – immutable, never changed, no collisions
• In writing COM code, always use interface pointers/references

Reference counting

• COM objects are reference counted
  – each object keeps track of the number of pointers to it
• When ref count goes to zero, element deletes itself
• Cycles can be a problem
• Remembering where to put all increments and decrements can be a problem

Each interface counted separately

• Each entry point/interface to a COM object is ref counted separately
  – allows an adaptor to be garbage collected

IUnknown

• All COM interfaces must extend IUnknown
  – HRESULT QueryInterface(const IID& iid, void ** ppv)
  – ULONG AddRef() // inc ref count
  – ULONG Release() // dec ref count

Query interface

• Like a C++ dynamic cast
  – Do you support this interface?
  – If so, give me back a pointer of that kind
    • incrementing the ref count for that interface
  – Else, signal failure

QueryInterface rules

• You always get the same IUnknown
• You can get an interface if you got it before
• You can get the interface you have
• You can get back to where you started
HRESULT
• COM doesn’t understand exceptions
• So almost all methods return an HRESULT
  – numerical indication of success or a specific error

Creating objects in COM
• Each component has a CLSID (class ID)
• Can call CoCreateInstance
• Can get Class Factory, then create instances directly
• Each DLL has a function that can return class factories for all classes that can be created by that DLL

Smart Pointers
• Automatically take care of reference counting
  – Some versions of COM smart pointers automatically perform dynamic casting (via calls to QueryInterface)
  – Not recommended

Smart Pointers
template <class T> class Iptr
{
  T* p;
  Iptr() : p(0) {}
  Iptr(T* q) : p(q) {
    if (p) p->AddRef();
  }
  Iptr(Iptr<T> q) : p(q.p) {
    if (p) p->AddRef();
  }
  ~Iptr() {
    if (p) p->Release();
  }
  T* operator T*() { return p; }
  T& operator*() { return *p; }
  T** operator&() {
    assert(p == NULL);
    return &p;
  }
  // operator = left as exercise

Administrivia
• Project 6 spec posted, due Monday May 14
  – initial discussion today, more on Thursday
• Project 5 out for commentary?
  – also due May 14
• Project 4 grades out soon, with grading scripts

CMSC 433, Spring 2001 COM, Part 2
Alan Sussman
May 1, 2001
Last time

- COM features
  - all accesses through interfaces
  - can create new ones, but can’t change existing ones
  - use GUIDs for uniqueness
  - all objects ref counted – each interface separately
  - since objects can have multiple interfaces
  - all objects extend IUnknown, return HRESULT
    * must implement QueryInterface(), AddRef(), Release()
  - use CoCreateInstance or class factory to create objects

FooBar

- interface IFoo : public IUnknown { … }
- interface IBar : public IUnknown { … }
- class Foo : public IFoo { … }
- class Bar : public IBar { … }
- class FooBar : public Foo, public Bar { … }

Implementing QueryInterface

```c
STDMETHODIMP QueryInterface(const IID& iid, void **ppv)
{
if (iid == IID_IUnknown || iid == IID_IFoo) {
    *ppv = static_cast<Foo*>(this);
} else if (iid == IID_IBar) {
    *ppv = static_cast<Bar*>(this);
} else {
    *ppv = null; return E_NOINTERFACE;
}
reinterpret_cast<IUnknown*>(*ppv)->AddRef();
return S_OK;
}
```

Things to note

- If you support 100 interfaces, cascaded if statements are going to get expensive
  - can’t use case statements (UIID’s aren’t ints)
  - could use custom hashtable
- Separate ref counts
  - could put call to AddRef in each branch
  - would eliminate reinterpret_cast
  - but would increase code size

I want everything

- Why can’t I ask
  - what is the list of all of the interfaces you support?
- What would you do with the list of all interfaces a component supports?
- Can use component categories

Component categories

- Assigned a GUID
- Corresponds to a set of interfaces
  - If a component is registered as member of a category
  - instances of that component support all of those interfaces
  - will still need to use QueryInterface to move between interfaces
Categories in Java

- Just define a Mega-interface
  - An interface that extends all of the interfaces in the category
  - Ask if class/component implements that
  - Can use reference of Megainterface type to invoke all methods from any interface in category
  - No casting needed

Component reuse

- How to reuse components?
  - Base class (implementation inheritance)
  - Containment (have as a member)
    - Delegation - Some methods get directly forwarded
    - Adaptor - Some methods get translated
  - Aggregation

Aggregation

- Say I have a component Bar
  - which uses a component Foo
- Foo implements the IFoo interface
- Bar also implements the IFoo interface
  - by handing things off to its Foo
- Could handle by delegation
  - but that adds an additional level of indirection

Using Aggregation

- When someone asks a Bar for its IFoo interface
  - just hand them a reference to your Foo
  - handles all IFoo function calls
- But what if you invoke QueryInterface on the IFoo reference and ask for an IBar interface?
Supporting aggregation

- You must be able to be told that you have an outer component
- Calls to QueryInterface should be routed to your outer component
- Reference counts are a little tricky
  - cycle could prevent stuff from being collected

Automation/IDispatch interfaces

- Can ask an interface which methods it supports, and invoke those methods
- Visual Basic example
  Dim Bullwinkle As Object
  Set Bullwinkle = CreateObject(“TalkingMoose”)
  Bullwinkle.PullFromHat 1, “Rabbit”
- Look for method “PullFromHat”
  - guess that it takes a LONG and a BSTR

Automation interfaces are a pain

- All argument types must be one of a predefined VARIANT list
  - primitive types
    - how do you pass a 1?
      - long, byte, short, ushort, ulong, int, uint
    - IUnknown and IDispatch types
  - No method overloading
  - No way to ask the types of a method

Dual interfaces

- Support both Dispatch invocation and regular method invocation
- Code for dispatch invocation can be built automatically
  - if you limit yourself to VARIANT argument types

Reflection in Java

- Allows you to ask a class or interfaces questions such as
  - which methods do you support?
    - what type are there arguments?
    - what fields do you have?
  - On an object
    - invoke a method
    - get or set a field

Reflection, continued

- All classes/objects support reflection
  - accessing private fields/methods needs permission from the security manager
Topics

• Marshalling
• Threads

Marshalling/Serialization

• COM allows objects to be marshalled
  – same as Serialization in Java
• Need to give extra data in IDL file
  – IDL = interface definition language

Example IDL for Marshalling

```csharp
interface IY : IUnknown {
    HRESULT fCount([out] int* sizeArray);
    HRESULT fArrayIn([in] int sizeIn,
                     [in, size_is(sizeIn)] int arrayIn[]);
    HRESULT fArrayOut([in] int maxSize,
                       [in, size_is(maxSize)] int arrayOut[],
                       [out] int* sizeOut);
}
```

COM Threads

• Free threads
  – similar to Java threads, must use explicit synchronization
• Apartment threads
  – COM objects can be grouped into an apartment
  – Each apartment has a designated thread

Apartment threads

• Single thread for entire apartment
• Call from a free thread, or from a different apartment, is marshalled
  – like an RMI call
  – Apartment thread must have a message loop to receive and dispatch calls
Apartment threads

• Simple synchronization model
  – backwards compatible with WIN32?
• Similar to having a single synchronization object for an entire set of components
• Still have potential problems such as deadlock

Servers in COM

• COM objects don’t have to be local
  – can make a remote call (like Java RMI)
• A COM object can be
  – in process
  – in process, different apartment
  – same machine, separate process
  – different machine

Advantages of COM servers

• Seg fault (illegal pointer dereference) only kills one process
• OS services can be provided as COM services