**COM - Component Object Model**

- **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 interfaces 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

```cpp
template <class T> class Iptr
{
  T* p;

  Iptr() : p(0) {};  // constructor
  Iptr(T* q) : p(q) {
    if (p) p->AddRef();  // increment reference count
  }
  Iptr(Iptr<T> q) : p(q.p) {
    if (p) p->AddRef();  // increment reference count
  }

  ~Iptr() {
    if (p) p->Release();  // decrement reference count
  }

  T* operator T*() { return p; }
  T& operator*() { return *p; }  // dereference
  T* operator->() { return p; }  // pointer to this
  T** operator&() {
    assert(p == NULL);
    return &p;  // return address
  }  // not recommended

  // operator = left as exercise
};
```

COM, part 2

FooBar

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

• Smart Pointers
Implementing QueryInterface

```cpp
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 members 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
    - Adapter - 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 handling 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?

Delegation/Forwarding

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 a 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

COM and software components

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, are marshalled
  – like a 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 only kills one process
- OS services can be provided as COM services

Java Beans

- A Java-based Software component technology
  - not the only way to do components in Java
- A Java Bean is a reusable software component that can be manipulated visually in a builder tool

Visual builder tools

- No, not a text editor
- Used to combine and customize existing components, not write from scratch
- What can be customized?
- What can be attached?

Design patterns

- Could allow you to view and change any fields of a component
  - Doable using reflection
  - But a bad idea
    - could make inconsistent changes, change fields that aren’t a part of public interface
- Set of design patterns to define how to customize Java Beans

Design patterns

- Very hot buzz word
  - some actual substance
- A common/standard way of doing something
  - can’t be captured by standard forms of OO reuse

Why reuse design patterns?

- Sometimes, because the pattern is a really cool and wonderful idea
- But mainly, so that when another programmer looks at your code
  - They will instantly see what idea you are trying to implement
Tools and patterns

• If a pattern is simple
  – automatic tools can understand it
  – extract pattern information
  – generate code

Properties

• If a component supports functions:
  – public void setMyValue(int v)
  – public int getMyValue()
• It has a MyValue property of type int
• For boolean types, getter function can be named is<Prop>()

Properties, continued

• Can have read-only, read/write or write-only properties
  – don’t have to define both getter and setter method

Java Bean Event Patterns

• A Bean Event must extend
  – class java.util.EventObject {
    public EventObject(Object src);
    public Object getSource();
  }
• Name should end in Event
  – e.g., tempChangeEvent

Event Listeners

• must implement java.util.EventListener
  – just a marker interface
• have event-Listener methods
  – void <eventName><EventType> e);
• interface TempChangeListener {
  void tempChanged(TempChangedEvent e);
}

Event sources

• Event sources fire events
• Have methods to attach/detach Listeners
  – public void add<ListenerType>(ListenerType ls);
  – public void remove<ListenerType>(ListenerType ls);
## Event Adapters

- Easy to construct event adapters
  - For example, an adapter that receives `temperatureChanged` events, and generates `temperatureIncreased` and `temperatureDecreased` events

## Bound properties

- Can set things up so that changes to bean property are indicated by an event
  - Events are a subtype of `java.beans.PropertyChangeEvent`
  - Listeners implement `PropertyChangeListener`
  - One Listener for all change events on the bean
    - May optionally support listeners for specific properties

## Constrained Properties

- Listeners can veto property changes
  - Listener throws `PropertyVetoException`
  - `set<Property>` method throws ...

## Builder tools

- Example: Sun’s BeanBox
- Can create instances of beans
- Modify their properties
  - Default mechanism
  - Special code for manipulating bean
- Attach Listeners, create adapters, ...

## Serialization and Persistence

- OK, so we can manipulate Java Beans in a builder tool
- Doesn’t help if we can’t distribute the beans
- Serialize the beans
- Application loads beans from Serialized form