Patterns in AAF Software

Barcelona Developers Conference

13 Nov 2001

Jim Trainor, AAF Association

AAF Software Patterns

- Software Patterns are not: Operational Patterns!
- Software Patterns are:
  - recurring solutions to common problems
  - documented
    - UML
    - Code fragments
    - Be pragmatic: what ever works for you
  - given names
  - over-hyped but useful nonetheless
- Also called Design Patterns
- Pattern Name + Pattern Documentation = Vocabulary to Convey Experience
- Promotes reuse.

Pattern References

- The Book
  - Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994) Design Patterns, Elements of Object Oriented Software. Reading, MA. Addison Wesley. ISBN 0-201-63361-2
- MSDN Article:
  - Yair Alan Griver, 1997, “Introduction to Design Patterns”

Property Value Processing

- Problem: Process a property value.
  - IAAFPropertyValue provides type information in the form of an IAAFTypeDef.
  - User must query the IAAFTypeDef to determine the actual value type in order to process it.
  - There are 16 types. IAAFTypeDefString, IAAFTypeDefInt, etc.
  - The type resolution code fragment required to implement value processing for one of this 16 type should occur once a program.

Property Value Processing

- Typical type resolution code fragment:

```
IAAFTypeDef* aTypeDef;
aPropertyValue->GetType( &aTypeDef )
eAAFTypeCategory_t cat = aTypeDef->GetTypeCategory();
switch( cat )
{
        case kAAFTypeCatCharacter:
                IAAFTypeDefCharacter aTypeDefCharacter;
                aTypeDef->QueryInterface( IID_AAFTypeDefCharacter,
                                          &aTypeDefCharacter );
        int aCharacter;
        aTypeDefCharacter->GetCharacter( aPropertyValue, &aCharacter );
        < Process aCharacter value >
        break;

        case kAAFTypeCatString:
        // repeat for all 16 types
}
```

Property Value Processing

- Design Goal: decouple value processing from the type resolution processing.
- Implement reusable type resolution processing.
- Implement the value processing using an abstract protocol.

Property Value Processing

AxProperty
-_sp : IAAFPropertySP
+AxProperty(in : IAAFPropertyValueSP) +GetValue() : IAAFPropertyValueSP

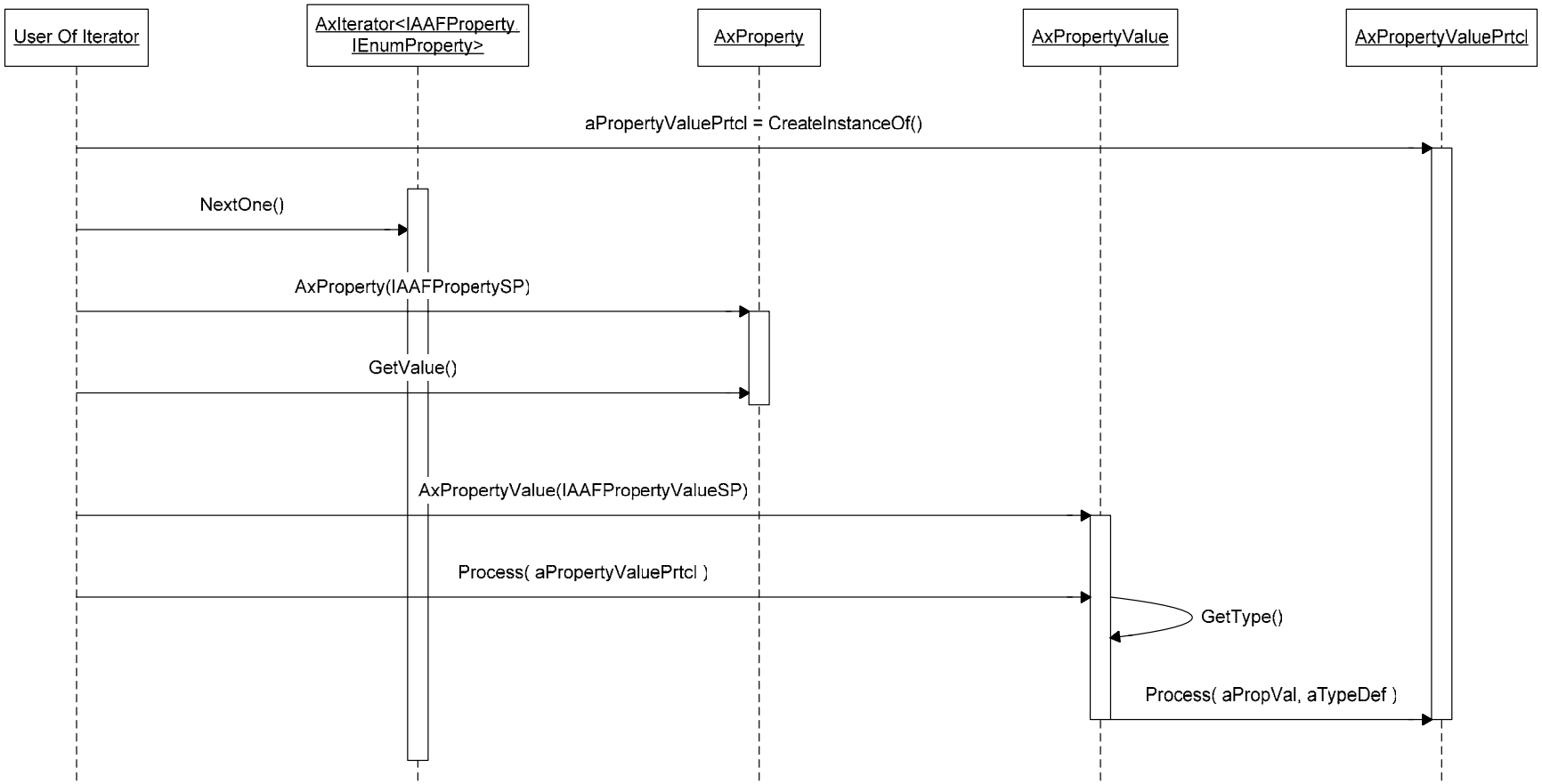
AxPropertyValue
-_sp : IAAFPropertyValueSP
+AxPropertyValue(in : IAAFPropertyValueSP&) +GetType() : IAAFTypeDefSP +Process(in : AxPropertyValuePrtcl&)

- AxPropertyValue::Process() implements the big switch statement once and for all.

AxPropertyValuePrtcl
+Process(in : IAAFTypeDefCharacterSP&) +Process(in : IAAFTypeDefIndirectSP&) +Process(in : IAAFTypeDefIntSP&) +Process(in : IAAFTypeDefRenameSP&) +Process(in : IAAFTypeDefEnumSP&) +Process(in : IAAFTypeDefExtEnumSP&) +Process(in : IAAFTypeDefFixedArraySP&) +Process(in : IAAFTypeDefRecordSP&) +Process(in : IAAFTypeDefSetSP&) +Process(in : IAAFTypeDefStreamSP&) +Process(in : IAAFTypeDefStringsSP&) +Process(in : IAAFTypeDefStringsSP&) +Process(in : IAAFTypeDefStrongObjSP&) +Process(in : IAAFTypeDefWeakObjRefSP&) +Process(in : IAAFTypeDefObjectRefSP&) +Process(in : IAAFTypeDefOpaqueSP&) +Process(in : IAAFTypeDefVariableArraySP&)

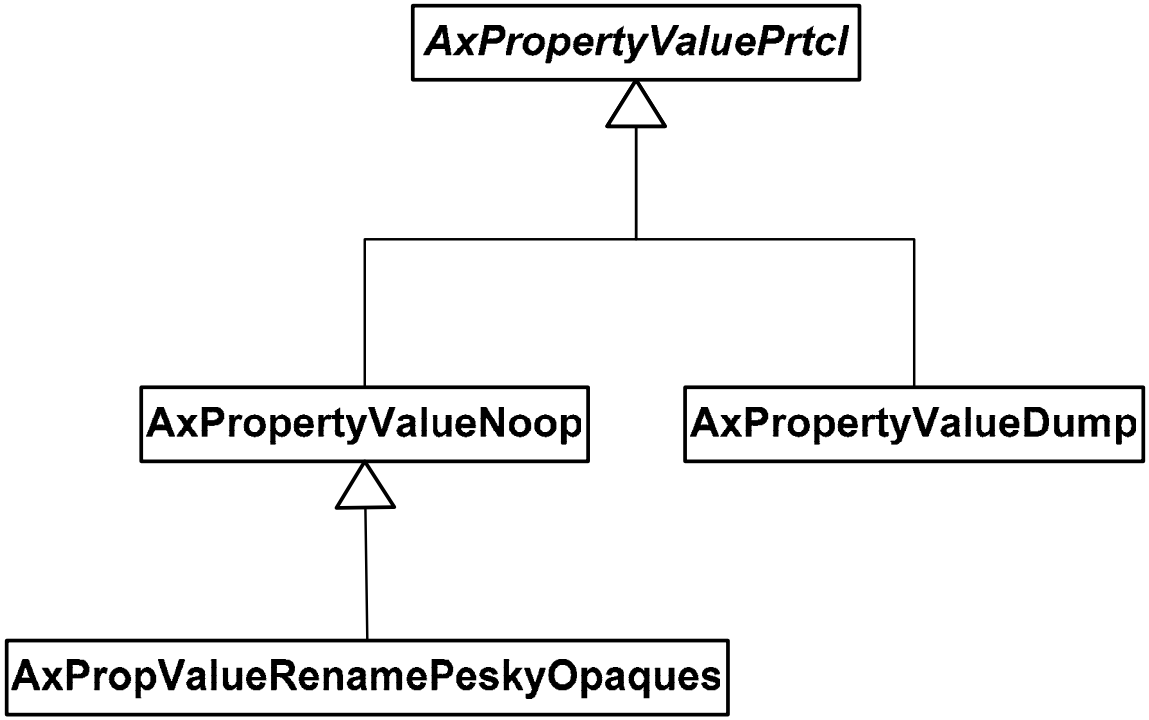
** Note, all Process() methods have an IAAFPropertyValueSP& argument. Not shown for clarity.*

Property Value Processing



Property Value Processing

- In the new (“Ax”) examples you will find:



ID Resolution

- Problem: Automate mapping of C++ types to their matching UUIDs.
  - Programmers have to provide type/id pairs that must agree. Example: QueryInterface(IID_AAFMob, &anIaafMobPtr)
  - Type info appears twice: once as an IID, once as a language type.
  - Eliminate redundancy.

ID Resolution

```
// Small template class that is intended to be specialized for each type for which
// automated ID Resolution is required.  An attempt to resolve the ID of types without the
// required specialization throws a “bad implementation” exception (run time error).

template <class Type>
inline const IID& IIDResolve( Type* )
{
        throw BadImp( L"IDResolve()" );
}

// Specialize for all types of interest.
template<> inline const IID& IIDResolve<IAAFMob>( IAAFMob* )
        { return IID_AAFMob; }
template<> inline const IID& IIDResolve<IAAFComponent>( IAAFComponent* )
        { return IID_AAFComponent; }
        .
        .
        .
```

Type Safe QueryInterface

- Problem: QueryInterface() is not type safe.
  - The need to use a void** as the second argument to QueryInterfaces() means the compiler cannot enforce type safeness.
  - How can type safeness be restored?

Type Safe QueryInterface

- Use the ID Resolution pattern to map a language type to an IID.
- Parameterize the type by implementing a templated QueryInterface wrapper.

Type Safe QueryInterface

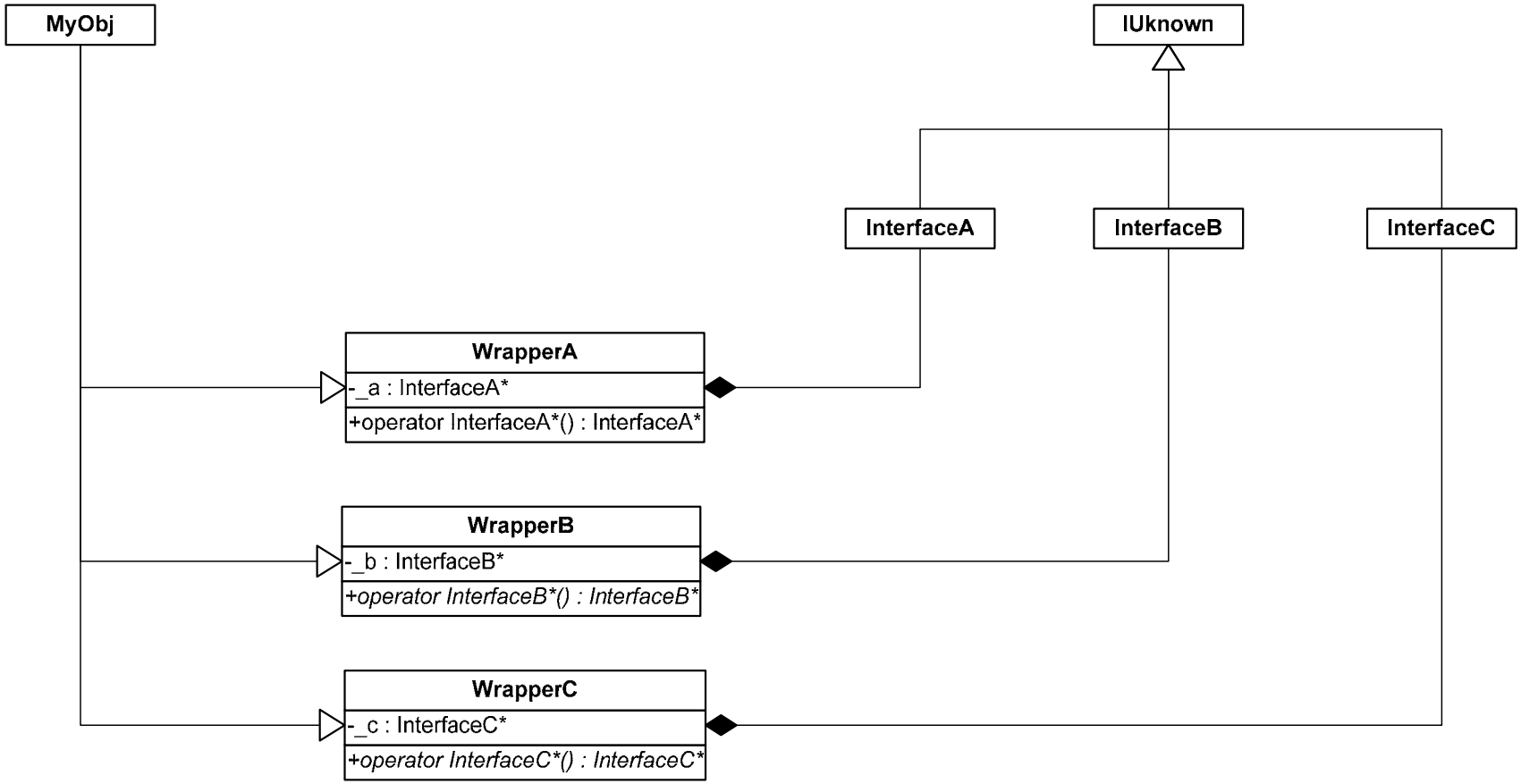
```
template <DstType, SrcType>
TypeSafeQueryInterface( SrcType* srcPtr, DstType* dstPtr  )
{
  HRESULT hr;
  hr = srcPtr->QueryInterface( IDResolve(dstPtr),
                               reinterpret_cast<void**>(&dstPtr) )

  if ( hr != AAFRESULT_SUCCESS )
          throw AnError;
}
```

- You only need to resolve the type ID once – when you specialize the IDResolve template.
- Automated type safety enforcement!
- Dangerous reinterpret_cast<> only needs to appear once.

Implicit Casting in COM Wrappers

- Problem: COM wrappers should operate seamlessly with code that expects a “bare” COM interface pointer.
  - Easy to mix code that uses wrapped COM interface pointers with code that uses “bare” interface pointers.
  - Use of wrappers is not “all or nothing”.
  - Especially useful when COM wrappers inherit the interface of other wrappers. Relieves need to access a single COM object via multiple pointers.

Implicit Casting in COM Wrappers



- Implement a C++ type cast operator for each wrapper to ensure wrapper can be used where a bare pointer is required.

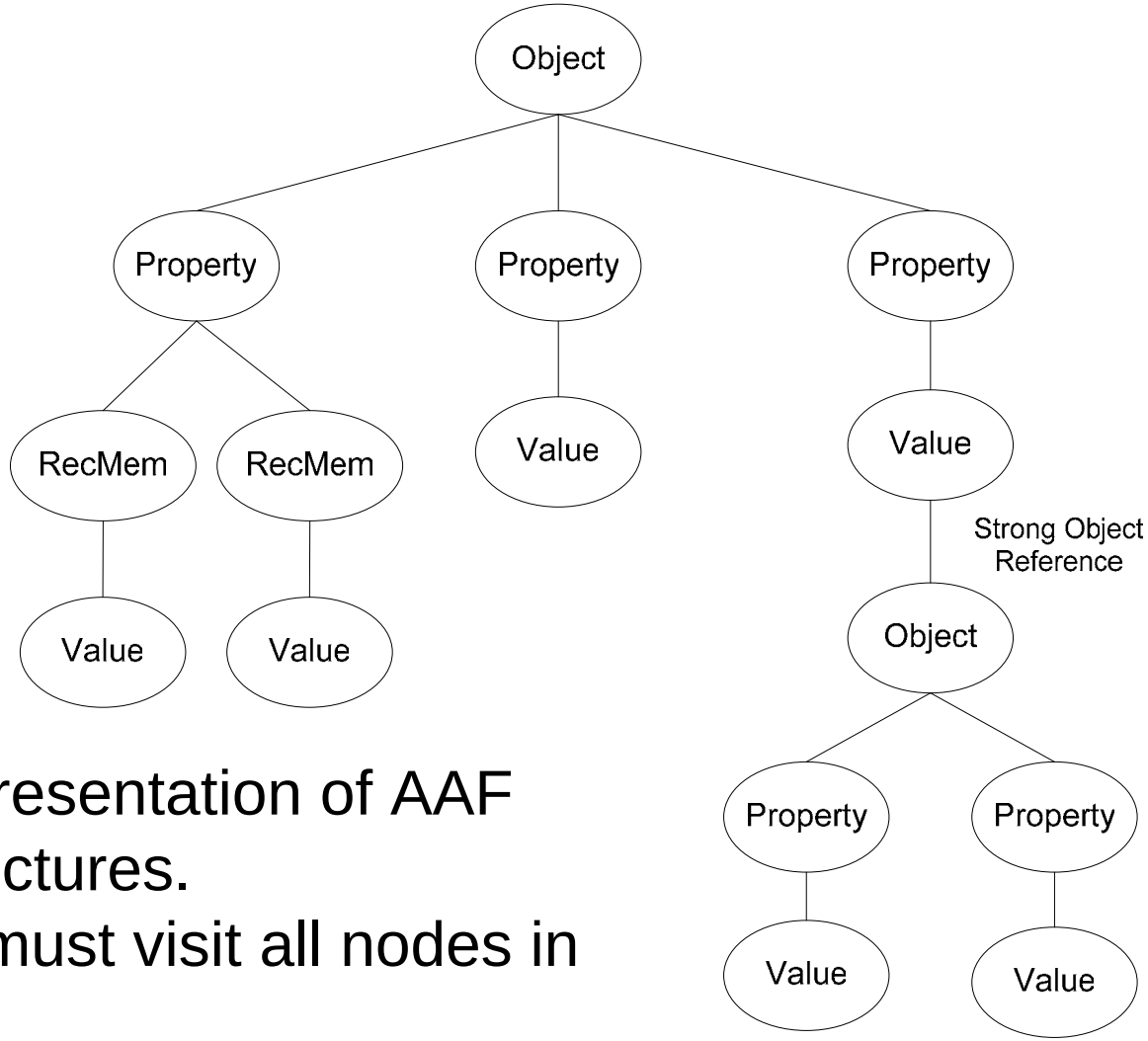
Implicit Casting in COM Wrappers

```
// Some functions that require bare COM interface pointers.
DoSomethingWithA( InterfaceA* );
DoSomethingWithB( InterfaceB* );
DoSomethingWithC( InterfaceC* );

//  Create some COM interfaces
InterfaceA* a = CreateA();
InterfaceB* b = CreateB();
InterfaceC* c = CreateC();

// A Class that wraps the COM interfaces
AggregateInterface wrapper( a, b, c );

// Wrappers can be used where bare pointers are required.
DoSomethingWithA( wrappers );  // Compilers figures out which
DoSomethingWithB( wrappers );  // type cast operator to call to
DoSomethingWithC( wrappers );  // recover the bare pointer.
```

Recursive Iterator

- Problem: A iterator that can visit every data structure in an AAF File.
  - Dump programs must “visit” all data structures in a file. Can the “visiting” code be factored out and made reusable.
  - Re-used to implement dump programs, browsers, file structure “verifier”, other “bulk processing” operations.

Recursive Iterator



- Tree representation of AAF data structures.
- Iterator must visit all nodes in the tree.

Recursive Iterator

- Require a similar representation of each element in the tree.
- Require a means to visit each node in the tree.
- A protocol that the visitor uses to expose the tree level (topology), and the underlying AAF data structure to the user of the iterator.

Recursive Iterator

- AxLib attempts to provide this.
- Somewhat crude at the moment. Requires refinement.
- Used to implement axDump.
- Could be re-used to implement browser.
- See: AxIntrChgObjRecIter.{cpp,h}
- Recursive Iterator is not necessarily a good name.

Recursive Iterator

- Better Names:
  - AAF Entity Tree Visitor
  - OPVR Tree Visitor (Object, Property, Value, Record)
  - Eviscerator (Oliver's suggestion)

CreateInstance Type Safety

- Problem: Automate translation of language type to ID's when creating objects as means of enhancing type safety.
  - Related to TypeSafeQueryInterface()
  - Uses IDResolution pattern.

CreateInstance Type Safety

- First, a variation of IDResolution to resolve AUIDs:

```
template <class Type>
inline const IID& AUIDResolve( Type* )
{
      throw BadImp( L“AUIDResolve()" );
}

// Specialize for all types of interest.
template<> inline const aafUID_t& AUIDDResolve<IAAFMob>( IAAFMob* )
                { return AUID_AAFMob; }

template<> inline const aafUID_t& AUIDDResolve<IAAFComponent>( IAAFComponent*
                 { return AUID_AAFComponent; }
```

CreateInstance Type Safety

```
template <Type>
TypeSafeCreateInstance( IAAFDictionarySP dictionary,
                                    Type* ptr  )
{
  HRESULT hr;
  hr = dictionary->CreateInstance( AUIDResolve(ptr),
                                   IIDResolve(ptr),
                                   reinterpret_cast<IUnkown**>(&ptr) )

  if ( hr != AAFRESULT_SUCCESS )
            throw AnError;
}
```

Going Further

- Ideas:
  - Attempt to generate compile time type errors rather than runtime “bad implementation” exceptions.
  - See boost.org for ideas to implement this.
  - Also see: Modern C++ Design, Addison Wesley for more ideas.