COM as a Better C++

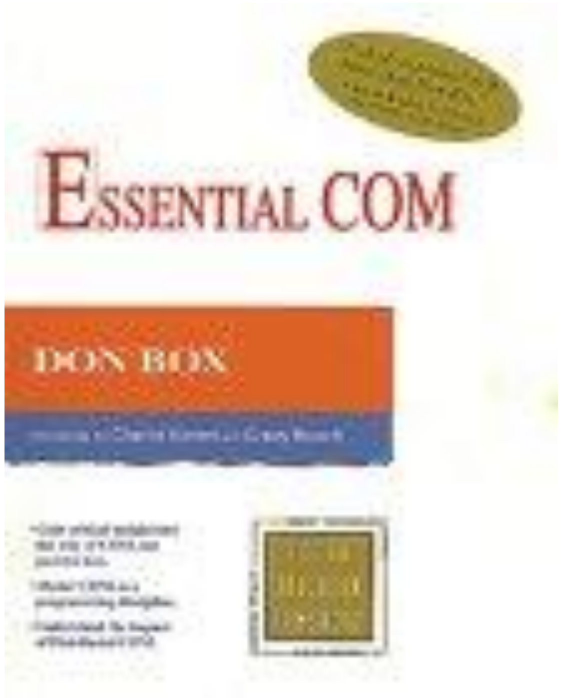
En udviklingshistorie med det formål at forklare COM's grundlæggende arkitektur

(Hvad og Hvorfor)

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

References

- This presentation is based on chapter 1 in Don Box's book "Essential COM" from Addison Wesley.
  This chapter is free for download as a pdf file:
  http://www.awprofessional.com



C++

- Most of the C++ canon was published in the late 1980s and early 1990s.
- During this era, most C++ developers worked on UNIX workstations and built fairly monolithic applications.
- This environment has understandably shaped the mindset of the C++ community, so C++ is not aimed at building distributed component based applications.

Reuse i C++

- One of the principal goals for C++ was to allow programmers to build user-defined types (UDTs) that could be reused outside their original implementation context.
- As a consequence a marketplace for C++ class libraries (normally distributed as source-code) emerged
  - albeit rather slowly
- Factors that work against reuse:
  - NIH (not invented here)
    - Pure hubris
    - Pure documentation
    - Difficult to comprehend others abstractions
  - Library users tend to modify the acquired library's source code!
    - which make it difficult to upgrade to the next version

Software Distribution and C++

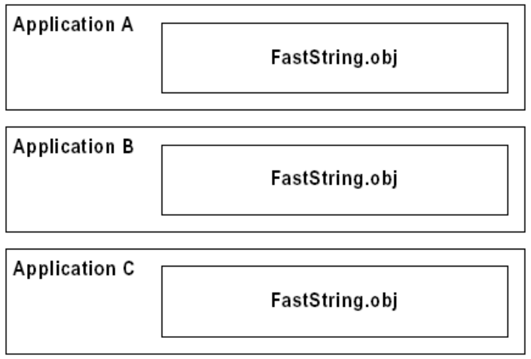
As an example on how to distribute software libraries in C++, we investigate how to distributed a hypothetic fast O(1) string search algorithm.

```
// faststring.h
class FastString {
  char *m_psz;

public:
  FastString(const char *psz);
  ~FastString(void);
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
};
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Distribution in source code form



- Traditionally, C++ libraries have been distributed in source code form.
- The users of a library would be expected to add the implementation source files to their make system and recompile the library sources locally using their C++ compiler.
- Assume that the FastString Obj occupies 16MB of space in the target executable image.
- If three applications use the FastString library, each of the three executables will contain the 16MB code. This means that if an end-user installs all three client applications, the FastString implementation occupies 48MB worth of disk space.
- Worse yet, if the end-user runs the three client applications simultaneously, the FastString code occupies 48MB virtual memory, as the operating system cannot detect the duplicate code that is present in each executable image.

Problems with distribution in source code form

- The application developers can modify the class library source to fit their own needs, producing a “private build”, that may be better suited to their application but it’s no longer the original library.
  This make it difficult to upgrade to the next version of the library.

- If (when) the library vendor finds a defect in the FastString class, there is no way to field-replace the implementation. Once the FastString code is linked into the client application, one can no longer simply replace the FastString code directly at the enduser’s machine.

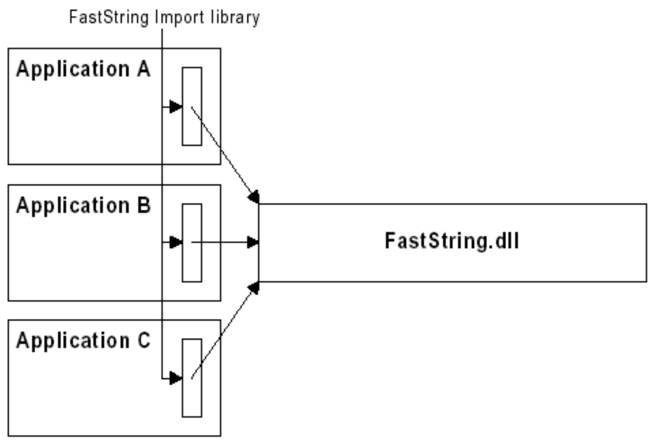
Dynamic Linking and C++

- One technique for solving the problems with source code destribution is to package the FastString class as a Dynamic Link Library (DLL).
- The Microsoft C++ compiler provides the __declspec(dllexport) keyword for just this purpose:

```
class __declspec(dllexport) FastString {
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString(void);
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
};
```

Instead of containing the actual code, the import library simply contains references to the file name of the DLL and the names of the exported symbols.

Benefits from use of a DLL



- When compiling the library to a DLL, the FastString machine code needs to exist only once on the user's hard disk.
- When multiple clients access the code for the library, the operating system's loader is smart enough to share the physical memory pages containing FastString's read-only executable code between all client programs.
- If the library vendor finds a defect in the source code, it is possible to build a new DLL and ship it to the end-users.
- **Moving the FastString library into a DLL is an important step toward turning the original C++ class into a field-replaceable and efficient reusable component.**

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

C++ and Portability

- Once the decision is made to distribute a C++ class as a DLL, one is faced with one of the fundamental weaknesses of C++:
  - **There is no standardization at the binary level!**
- C++ compilers have different implementations of function overloading
  - implements name mangling different.
- The classic technique of using extern "C" to disable symbolic mangling would not help in this case, as the DLL is exporting member functions.
- Exceptions are another example of a language feature often implemented in different ways.
  A C++ exception thrown from a function compiled with the Microsoft compiler cannot be caught reliably by a client program compiled with the Watcom compiler.

Problems with C++ and DLL’s

- The lack of a C++ binary standard limits what language features can be used across DLL boundaries.
- To create a vendor-independent component simply exporting C++ member functions from DLLs is not enough.

Encapsulation and C++

Assume the library developer builds an enhanced version of the library where the length function also is a O(1) algorithm.

```
// faststring.h
class FastString {
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString(void);
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
};
```

```
// faststring.h version 2.0
class __declspec(dllexport) FastString {
  const int m_cch; // count of characters
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString(void);
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
};
```

Data members are private, and no change of public member functions, so we can distribute the new version safely … ?

New DLL to Developers

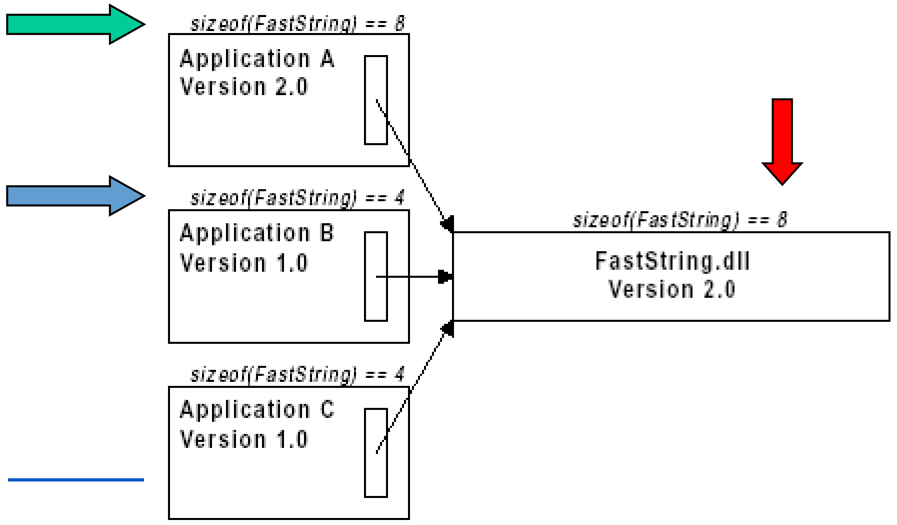
- When the application developers receive the updated FastString, they integrate the new class definition and DLL into their source code control system and fire off a build to test the new and improved FastString.
- As expected no source code modifications are required to take advantage of the new version of Length().
- Encouraged by this experience, the development team convinces management to add the new DLL to the final “golden master” CD that is about to go to press.
- Like most installation programs, the setup script for the application developers’s product is designed to silently overwrite any older versions of the FastString DLL that may be present on the end-user’s machine.
- This seems innocent enough, as the modifications did not affect the public interface of the class, so the silent machine-wide upgrading to FastString version 2.0 should only enhance any existing client applications that had been previously installed.

New DLL to end-users

- End-users finally receive their copies of the application developers highly anticipated product.
- Each end-user immediately drops everything and installs the new application on their machine to try it out.
- After the thrill of being able to perform extremely fast text searches wears off, the enduser goes back to his or her normal life and …
- launches a previously installed application that also happens to use the FastString DLL.
- For the first few minutes all is well.
- Then, suddenly, a dialog appears indicating that an exception has occurred and that all of the end user’s work has been lost. ☹
- The end-user reinstalls the operating system and all applications, but even this does not keep the exception from reoccurring.
- What happened?

New DLL to end-users 2

- What happened was that the library vendor was lulled into believing that C++ supported encapsulation.
- C++ does support syntactic encapsulation via its private and protected keywords,
- **BUT the C++ standard has no notion of binary encapsulation.**
- This is because the compilation model of C++ requires the client's compiler to have access to all information regarding object layout in order to instantiate an instance of a class or to make non-virtual method calls.
- This includes information about the size and order of the object's private and protected data members.



Separating Interface from Implementation

- The concept of encapsulation is based on separating what an object looks like (its interface) from how it actually works (its implementation).
- The problem with C++ is that this principle does not apply at a binary level, as a C++ class is both interface and implementation simultaneously.
- This weakness can be addressed by modeling the two abstractions as two distinct C++ classes.
- By defining one C++ class to represent the interface to a data type and another C++ class as the data type's actual implementation, the object implementor can theoretically modify the implementation class's details while holding the interface class constant.

Separating Interface from Implementation 2

One simple approach would be to use a handle class as the interface.
The handle class would simply contain an opaque pointer whose type would never be fully defined in the client's scope.

```
// faststringitf.h
class __declspec(dllexport) FastStringItf {
  class FastString; // introduce name of impl. class
  FastString *m_pThis; // opaque pointer
  // (size remains constant)
public:
  FastStringItf(const char *psz);
  ~FastStringItf(void);
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
};
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

The implementations of the interface class

The implementations of the interface class's methods simply forward the method calls on to the actual implementation class:

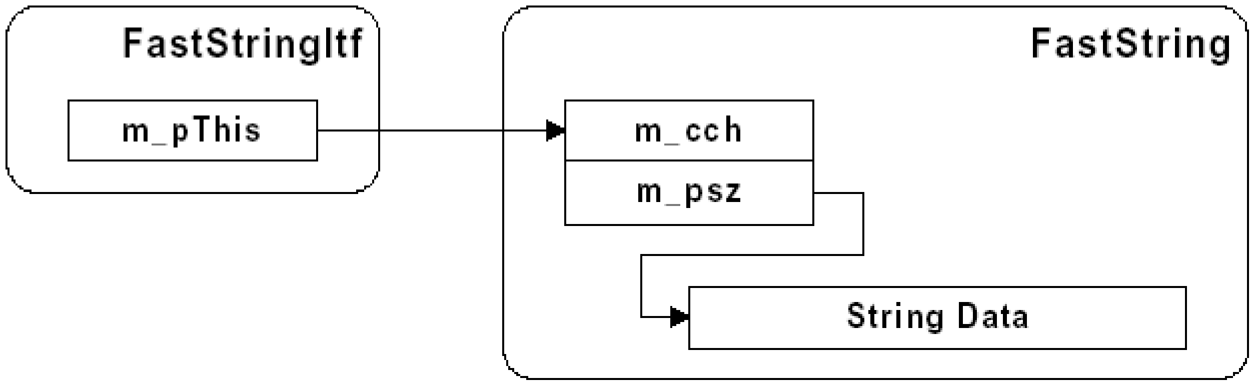
```
// faststringitf.cpp/// (part of DLL, not client) //
#include "faststring.h"
#include "faststringitf.h"
FastStringItf::FastStringItf(const char *psz)
  : m_pThis(new FastString(psz)) {
  assert(m_pThis != 0);
}
FastStringItf::~FastStringItf(void) {
  delete m_pThis;
}
int FastStringItf::LengthItf(void) const {
  return m_pThis->Length();
}
int FastStringItf::Find(const char *psz) const {
  return m_pThis->Find(psz);
}
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

The implementations of the interface class 2

- These forwarding methods would be compiled as part of the FastString DLL, so as the layout of the C++ implementation class FastString changes, the call to the new operator in FastStringItf's constructor will be recompiled simultaneously, ensuring that enough memory is always allocated.
- The client never includes the class definition of the C++ implementation class FastString.
- This affords the FastString implementor the flexibility to evolve the implementation over time without breaking existing clients.

Runtime model of using handle classes



- Note that the level of indirection introduced by the interface class imposes a binary firewall between the client and the object implementation.
- This binary firewall is a very precise contract describing how the client can communicate with the implementation.
- All client-object communications take place through the interface class, which imposes a very simple binary protocol for entering the domain of the object's implementation.
- This protocol does not rely on any details of the C++ implementation class.

Abstract Bases as Binary Interfaces

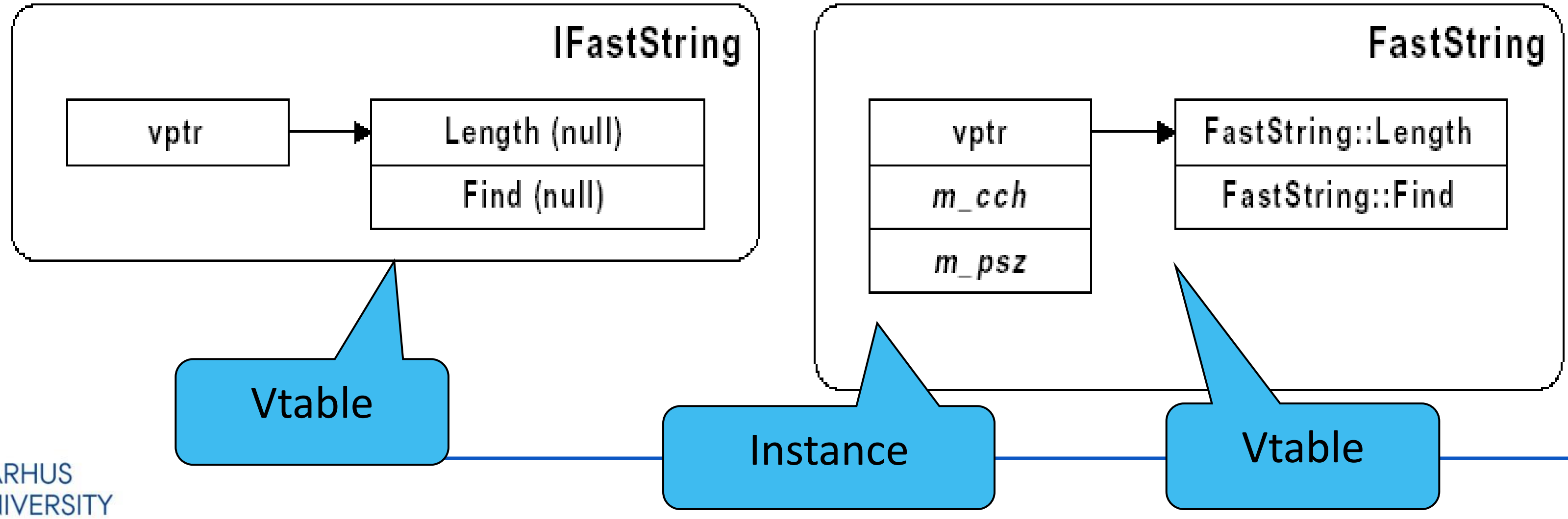
- Use of abstact base classes as interface classes, make it possible to avoid all the "dummy" forwarding methods.
- But it relies one one assumption:
  - All C++ compilers on a given platform must implement the virtual function call mechanism equivalently.
  - Looky for us – they do.

Abstract Bases as Binary Interfaces 2

```
// ifaststring.h
class IFastString {
public:
  virtual int Length(void) const = 0;
  virtual int Find(const char *psz) const = 0;
};
```

```
// faststring.h
class FastString : public IFastString {
  const int m_cch;
  char *m_psz;

public:
  FastString(const char *psz);
  ~FastString(void);
  int Length(void) const;
  int Find(const char *psz) const;
};
```



Abstract Bases as Binary Interfaces 3

- It's not possible to instantiate a class with methods marked as pure virtual, so how can the client create an instance without access to the implementation class' header file?
- The library implementor must supply a global function in the DLL that would call new:

```
// faststring.cpp (part of DLL)
IFastString *CreateFastString (const char *psz) {
  return new FastString(psz);
}
```

Abstract Bases as Binary Interfaces 4

- The last remaining barrier to overcome is related to object destruction.
- How can we guarantee the correct destructor being executed no matter what C++ compiler the client use?
- One workable solution is to add an explicit Delete method to the interface as another pure virtual function and have the implementation class delete itself in its implementation of this method.

Abstract Bases as Binary Interfaces 5

The updated version of the interface header file looks like this:

```
// ifaststring.h
class IFastString {
public:
  virtual void Delete(void) = 0;
  virtual int Length(void) const = 0;
  virtual int Find(const char *psz) const = 0;
};

extern "C“ IFastString *CreateFastString (const char *psz);
```

Example Client Code

To use the FastString data type, clients simply need to include the interface definition file and call CreateFastString to begin working:

```
#include "ifaststring.h"
int f(void) {
  int n = -1;
  IFastString *pfs = CreateFastString("Hi Bob!");
  if (pfs) {
    n = pfs->Find("ob");
    pfs->Delete();
  }
  return n;
}
```

Runtime Polymorphism

Deploying class implementations using abstract base classes as interfaces opens up a new world of possibilities in terms of what can happen at runtime: polymorphism and late binding.
Note that the FastString DLL exports only one symbol, CreateFastString.
This makes it fairly trivial for the client to load the DLL dynamically on demand using LoadLibrary and resolve that one entry point using GetProcAddress:

```
IFastString *CallCreateFastString(const char *psz) {
  static IFastString * (*pfn)(const char *) = 0;
  if (!pfn) { // init ptr 1st time through
    const TCHAR szDll[] = __TEXT("FastString.DLL");
    const char szFn[] = "CreateFastString";
    HINSTANCE h = LoadLibrary(szDll);
    if (h)
      *(FARPROC*)&pfn = GetProcAddress(h, szFn);
  }
  return pfn ? pfn(psz) : 0;
}
```

Object Extensibility

- Separating the interface from the implementation makes it possible to safely change the implementation code and redistribute the DLL to both clients and end-users.
- But what about changing the interface?
- Changing the interface definition completely violates the encapsulation of the object (its public interface has changed) and would break all existing clients.
- **This implies that interfaces are immutable binary and semantic contracts that must never change!**
- The solution to this problem is to allow an implementation class to expose more than one interface. ☺

Object Extensibility 2

Consider the simple case of an interface extending another interface:

```
class IFastString2 : public IFastString {
public:
  // real version 2.0
  virtual int FindN(const char *psz, int n) = 0;
};
```

Clients can interrogate the object reliably at runtime to discover whether or not the object is IFastString2 compatible by using C++'s dynamic_cast operator:

```
int Find10thBob(IFastString *pfs) {
  IFastString2 *pfs2=dynamic_cast<IFastString2*>(pfs);
if (pfs2) // the object derives from IFastString2
  return pfs2->FindN("Bob", 10);
else { // object doesn't derive from IFastString2
  error("Cannot find 10th occurrence of Bob");
  return -1;
  }
}
```

OE 3: Multiple inheritance of interfaces

```
class IPersistentObject {
public:
  virtual void Delete(void) = 0;
  virtual bool Load(const char *pszFileName) = 0;
  virtual bool Save(const char *pszFileName) = 0;
};
```

```
class FastString : public IFastString, public IPersistentObject
{
  int m_cch; // count of characters
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString(void);
// Common methods
  void Delete(void); // deletes this instance
// IFastString methods
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
// IPersistentObject methods
  bool Load(const char *pszFileName);
  bool Save(const char *pszFileName);
};
```

Problems With Object Extensibility

- Both examples shown demand the client to use RTTI (dynamic_cast)
- This is bad because RTTI is a very compiler-dependent feature.
- The solution is to implement the type cast in the interface class.

```
virtual void *Dynamic_Cast(const char *pszType) =0;
```

Adding Dynamic_Cast

```
class IPersistentObject {
public:
  virtual void *Dynamic_Cast(const char *pszType) =0;
  virtual void Delete(void) = 0;
  virtual bool Load(const char *pszFileName) = 0;
  virtual bool Save(const char *pszFileName) = 0;
};


class IFastString {
public:
  virtual void *Dynamic_Cast(const char *pszType) =0;
  virtual void Delete(void) = 0;
  virtual int Length(void) = 0;
  virtual int Find(const char *psz) = 0;
};
```

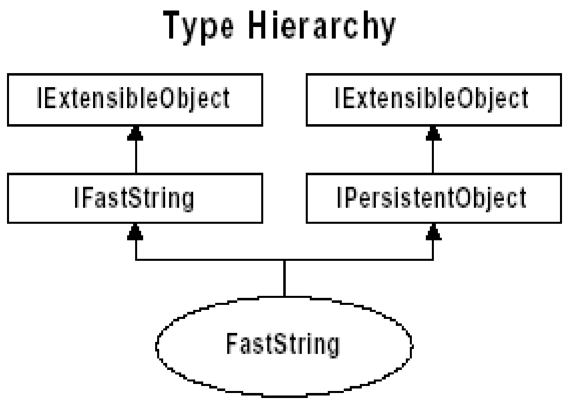
IExtensibleObject Extract

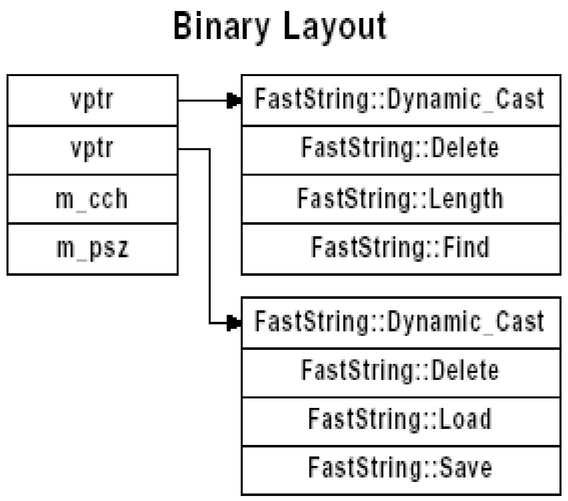
```
class IExtensibleObject {
public:
  virtual void *Dynamic_Cast(const char* pszType) =0;
  virtual void Delete(void) = 0;
};

class IPersistentObject : public IExtensibleObject {
public:
  virtual bool Load(const char *pszFileName) = 0;
  virtual bool Save(const char *pszFileName) = 0;
};

class IFastString : public IExtensibleObject {
public:
  virtual int Length(void) = 0;
  virtual int Find(const char *psz) = 0;
};
```

Implementation of Dynamic_Cast





```
void *FastString::Dynamic_Cast(const char *pszType) {
  if (strcmp(pszType, "IFastString") == 0)
    return static_cast<IFastString*>(this);
  else if (strcmp(pszType, "IPersistentObject") == 0)
    return static_cast<IPersistentObject*>(this);
  else if (strcmp(pszType, "IExtensibleObject") == 0)
    return static_cast<IFastString*>(this);
  else
    return 0; // request for unsupported interface
}
```

Client use of Dynamic_Cast

```
bool SaveString(IFastString *pfs, const char *pszFN){
  bool bResult = false;
  IPersistentObject *ppo = (IPersistentObject*)
                  pfs->Dynamic_Cast("IPersistentObject");
  if (ppo)
    bResult = ppo->Save(pszFN);
  return bResult;
}
```

Resource Management 1

```
void f(void) {
  IFastString *pfs = 0;
  IPersistentObject *ppo = 0;
  pfs = CreateFastString("Feed BOB");
  if (pfs) {
    ppo = (IPersistentObject *) =
          pfs->Dynamic_Cast("IPersistentObject");
    if (!ppo)
      pfs->Delete();
    else {
      ppo->Save("C:\\SomeFile.txt");
      ppo->Delete();
    }
  }
}
```

- The client has to keep track of which pointers are associated with which objects and call Delete only once per object.

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Resource Management 2

- One way to simplify the client's task is to push the responsibility for managing the lifetime of the object down to the implementation.
- One simple solution to this problem is to have each object maintain a reference count that is incremented when an interface pointer is duplicated and decremented when an interface pointer is destroyed.

Resource Management 3

```
class IExtensibleObject {
public:
  virtual void *Dynamic_Cast(const char* pszType) =0;
  virtual void DuplicatePointer(void) = 0;
  virtual void DestroyPointer(void) = 0;
};
```

- All users of IExtensibleObject must now adhere to the following two mandates:
  1. When an interface pointer is duplicated, a call to DuplicatePointer is required.
  2. When an interface pointer is no longer in use, a call to DestroyPointer is required.

Resource Management 4

- These methods could be implemented in each object simply by noting the number of live pointers and destroying the object when no outstanding pointers remain:

```
class FastString : public IFastString, public IPersistentObject {
  int m_cPtrs; // count of outstanding ptrs
  …
public:
  // initialize pointer count to zero
  FastString(const char *psz) : m_cPtrs(0) {}
  void DuplicatePointer(void) {
    // note duplication of pointer
    ++m_cPtrs;
  }
  void DestroyPointer(void) {
    // destroy object when last pointer destroyed
    if (--m_cPtrs == 0)
      delete this;
  }
  …
};
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Summary

- To build a reusable binary component we:
  - deploy the class as a Dynamic Link Library (DLL) to decouple the physical package of the class from the packaging of its clients
  - use the separation of interfaces and implementations to encapsulate the implementation details of the data type behind a binary firewall
  - use the abstract base class approach to define interfaces, this firewall took the form of a vptr and vtbl
  - use an RTTI-like construct for dynamically interrogating an object to discover whether it in fact implements a desired interface. This construct give us a technique for extending existing versions of an interface as well as exposing multiple unrelated interfaces from a single object
  - use DuplicatePointer and DestroyPointer to implement Resource Management
- **In short:**

we have just engineered the

Component Object Model

COM