NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

COM AND XPCOM AS A SOLUTION TO BAMBOO'S VERSIONING PROBLEM

by

Mithat Daglar

March 2000

Thesis Advisor: Michael Zyda
Co-advisor: Michael V. Capps

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE: COM AND XPCOM AS A SOLUTION TO BAMBOO'S VERSIONING PROBLEM.

5. FUNDING NUMBERS

6. AUTHOR Mithat Daglar

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(maximum 200 words)***

Bamboo is a systems toolkit that is primarily concerned with supporting performance-critical applications that must run continuously for extremely long periods of time. Bamboo supports this by managing the loading and unloading of executable code into and out of process memory at runtime. Thus, as application requirements change over time, obsolete code can be replaced without having to restart the application. This technique's flexibility has already been demonstrated, but fails in one critical way. Although the C++ programming language standard defines a consistent syntax, it fails to specify a consistent binary encapsulation. Thus, if the executable code for a C++ base class is dynamically replaced, it is very likely that its in memory layout differs from before and therefore incompatible with whatever derived classes may exist. The only recourse is to recompile and reload the derived classes as well.

Component Object Model (COM) and Cross Platform Object Model (XPCOM) solve C++'s weakness by enforcing a complete separation of a class's interface from its implementation. This thesis demonstrates support for dynamic versioning of Bamboo C++ modules using COM and XPCOM.

14. SUBJECT TERMS The versioning Problem, Component Object Model, Cross Platform Object Model	15. NUMBER OF PAGES 140
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited

COM AND XPCOM AS A SOLUTION TO BAMBOO'S VERSIONING PROBLEM

Mithat Daglar
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1993

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2000

Author: _____
Mithat Daglar

Approved by: _____
Michael Zyda, Thesis Advisor

_____
Michael V. Capps, Thesis Co-advisor

_____
Kent Watsen, Second Reader

_____
Dan Boger, Chairman
Department of Computer Science

ABSTRACT

Bamboo is a systems toolkit that is primarily concerned with supporting performance-critical applications that must run continuously for extremely long periods of time. Bamboo supports this by managing the loading and unloading of executable code into and out of process memory at runtime. Thus, as application requirements change over time, obsolete code can be replaced without having to restart the application. This technique's flexibility has already been demonstrated, but fails in one critical way. Although the C++ programming language standard defines a consistent syntax, it fails to specify a consistent binary encapsulation. Thus, if the executable code for a C++ base class is dynamically replaced, it is very likely that its in memory layout differs from before and therefore incompatible with whatever derived classes may exist. The only recourse is to recompile and reload the derived classes as well.

Component Object Model (COM) and Cross Platform Object Model (XPCOM) solve C++'s weakness by enforcing a complete separation of a class's interface from its implementation. This thesis demonstrates support for dynamic versioning of Bamboo C++ modules using COM and XPCOM.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. MOTIVATION:

C++ supports only a syntactic encapsulation. Since this encapsulation mechanism does not apply at the binary level, C++ classes are subject to what is called the "versioning problem". The versioning problem occurs whenever a class is reused; the class must be recompiled if its binary layout has changed. C++-based dynamically extensible systems, such as Bamboo [Ref. 1], are especially affected by this problem. This thesis will investigate two solutions for the versioning problem, COM and XPCOM, by applying to Bamboo.

B. BACKGROUND:

1. The Versioning Problem:

Creating reusable binary objects is one and maybe the most principal goals of the Object Oriented (OO) Programming. Theoretically software reusability saves time in program development. It encourages the reuse of proven and debugged, high-quality software. However, due to design, development and run-time obstacles, this principle goal is not always easy to reach for C++.

One of the biggest obstacles to building reusable binary components in C++ is related to this language's encapsulation mechanism. C++ supports only syntactic encapsulation via its "private" and "protected" keywords [Ref. 2]. The only thing that

these keywords do is to prevent programmers from accessing the data members or functions directly. But the compiler always has the right to access every data member and make predictions about the layout of the class. This causes the reusing class to think that the reused class has always the same binary layout in the memory even if it has changed. These changes include adding new data members, removing existing data members changing the type or even the declaration order of the data members in the class declaration. Two very simple classes can be taken as an example to illustrate this problem:

```
class A {
private :
      int number;
public :
      void setNumber(int p_number);
      int getNumber();
};
class B : public A {
private :
      char* name;
public :
      void setName(char* p_name);
      char* getName();
};
```

In this example class B tries to reuse class A by inheritance. After compilation of the class B, all the functions, including the constructor and the destructor of class B assume that the member data of class A is located in memory location m. When an instance of class B is created, the creation mechanism first calls the constructor of class A and then constructor of class B is called. Assuming that the class A finishes at the

memory location m+8, class B's constructor allocates eight bytes of memory starting from m+8 for its name data member.

If the implementer of class A makes a change to his class and adds a new data member after class B's compilation:

```
class A {
        int number;
        int number2;
        //other code
};
```

there is no way for class B to know this change unless it recompiles. Class B's constructor still assumes that class A finishes at memory location m+8 and allocates 8 bytes of memory for its data member starting at memory location m+8. But in reality that portion of the memory belongs to class A's newly added data member (number2). In some part of the program a call to getNumber2 function will return the name data member instead of the number2 data member. Likewise a call to setNumber2 will overwrite the name data member with its parameter. This is shown in Figure 1.1. Of course the results of these operations are unpredictable. If in some part of the program setNumber2 function has overwritten the name data member with a value of 0, trying print out the name:

```
cout << my_B.getName();
```

will crash the program.

2. Bamboo and Reusabilty:

Due to its execution speed, strongly typed preprocessor and support for object oriented and generic programming semantics, Bamboo is implemented in C++ [Ref. 3].

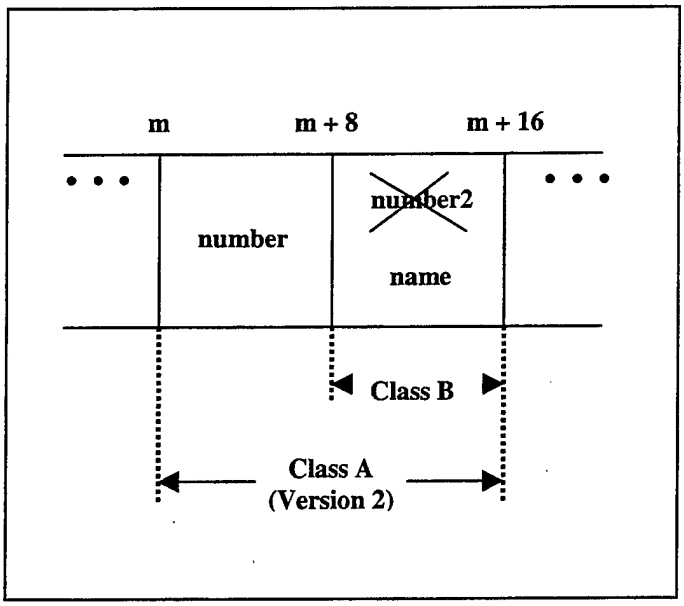

Figure 1.1. Memory Allocation of Class A and Class B: New data member "number" is overwritten by "name"

Modules that are written in C++ can reuse the existing code in the other modules in two ways:

1. Since Bamboo is open source, a module developer can simply copy the header and source files into his module and add these source files to his build.

2. He can specify a dependency on the other module and let Bamboo dynamically load this module.

Of course the first method does not reflect what is meant by "reuse" and has some serious drawbacks. The net result of this procedure is that executable code of the other module is bundled as part of the overall developer's module. If binary code of Person class is 1 megabyte, than each module that uses Person module's source code with above method adds another 1 Megabyte of more virtual memory as shown in Figure 1.2.

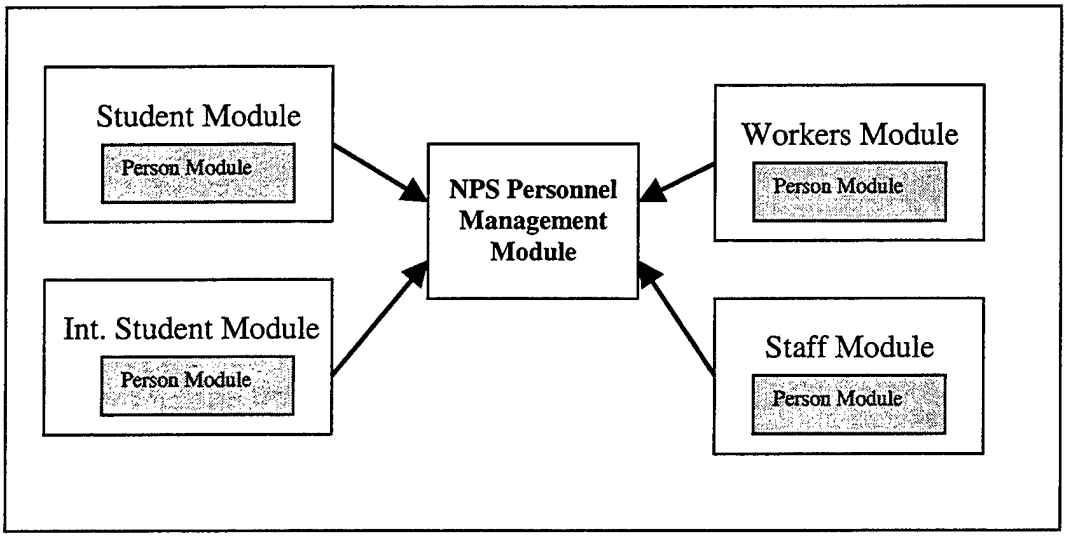

Figure 1.2. Effect of Source Code Bundling [After Ref. 1 and 4]

Another and more serious problem with this method is that when the Person module developer finds an error in one of his functions and fixes it or he changes one of his function implementation to a more efficient one, all depending module developers must recopy the source and rebuild their modules.

These drawbacks are not acceptable by Bamboo whose one of the main goals is dynamic extensibility. Thus the only reasonable reuse mechanism in Bamboo is specifying dependency on other modules. Student module owner indicates this dependency in his module's make file and after this point the only thing that he needs is Person module's header files. He can copy these header files into his source directory or he can specify the absolute path to these header files. A better way to deal with header files is putting them into module's include directory. If Person module is implemented in this way, Student module developer does not need to do any of these. In the compile time, Bamboo first checks if the header file is in this module's source directory, than checks if it is in this module's include directory and finally checks the include directories

of dependent modules. After compilation, C++ linker will resolve all functions of Person class by looking at Person module's dynamic link library. For linker to be able to find these functions in the dynamic link library, Person module must have exported them. Functions can be exported to dynamic link libraries by using Bamboo's export flag:

```
BB_Proper_Export_Flag void setName(char* name);
```

To export all the functions of a class this flag can be put before the name of the class :

```
class BB_Proper_Export_Flag Person { … }
```

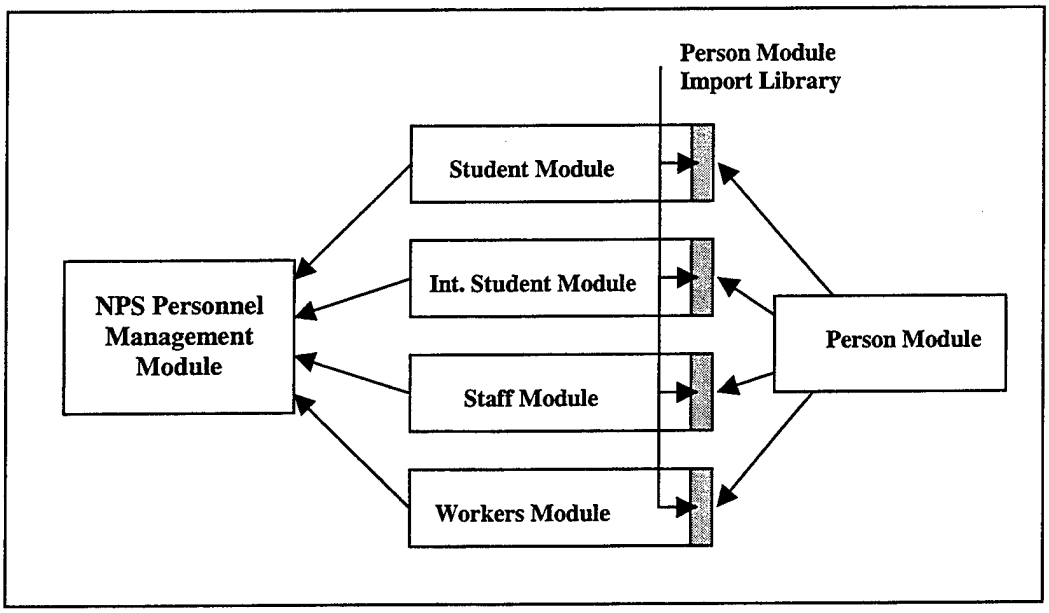

Figure 1.3. Effect of Specifying Dependency on Other Modules: Bamboo loads only one copy of Person module DLL [After Ref. 1 and 4]

Specifying dependency and dynamically linking against the other modules solves the problem of binary code bundling and requirement for rebuild to get the benefits of error fixing or better implementation of functions. As shown in Figure 1.3, for all Person module dependents, Bamboo loads only one copy of Person module dynamic link library. If Person module developer finds an error in one of his function implementation or he

wants to enhance the efficiency of a function he can do this, as long as he does not change the function's signature. Since it dynamically links against the Person module, Student module does not need to be rebuilt.

C. THESIS STATEMENT:

The versioning problem is an important obstacle for Bamboo's extensibility. Any change to the binary layout of a class in a module requires all dependent modules to be recompiled. A proper solution must be found to overcome this problem.

D. CHAPTERS OVERVIEW:

This study is composed of six chapters. Chapter II gives a literature overview. It explains why separating interface from implementation is important for C++ projects and overlooks at briefly C wrappers around C++ API, CORBA and in detail COM, which are the current solutions to the versioning problem.

Chapter III describes the reason why COM and XPCOM are better solutions for Bamboo versioning problem and provides the design issues for application of COM and XPCOM to Bamboo modules.

Chapter IV gives the implementation. It demonstrates an unsafe module that does not use COM or XPCOM, thus subject to the versioning and then shows how to apply COM and XPCOM to this module to make it free from versioning.

Chapter V analyzes the modules implemented in the forth chapter. It runs each module ten times with different configurations and shows that the unsafe module always fails but COM and XPCOM modules are not affected by these changes.

Chapter IV gives conclusions and recommendations for the future work.

II. RELATED WORK

A. INTRODUCTION:

This chapter first explains why separating interface from implementation in C++ applications is important for solving the versioning problem. Then it discusses three different solutions that are currently used to solve the versioning problem: using C Wrappers around C++ API, the Common Object Request Broker Architecture (CORBA), and the Component Object Model (COM). Finally it looks at how Microsoft and Mozilla solve this problem in their applications.

B. SEPARATING INTERFACE AND IMPLEMENTATION:

The versioning problem that was introduced in the previous chapter is the result of the encapsulation mechanism that is used by C++. The concept of encapsulation is based on separating what an object looks like (the interface) from how it actually works. The problem with C++ is that this principle does not apply at binary level, as a C++ class is both interface and implementation simultaneously [Ref. 4]. Because of this, a compiler can see every implementation detail about the class from its header file and makes memory allocations accordingly. But the programmer does not need any implementation detail about the class that he is reusing. He cannot write code that accesses directly to the private and protected data members and functions anyway. The only thing he needs to

know about the class is its functionality and how he can use it. In a C++ class all this information is in the public section of the class.

This weakness of C++ can be addressed by modeling the interface and implementation as two distinct entities. By defining one entity to represent the interface to a data type and another entity as the data type's actual implementation, the class's implementation can be modified while holding the interface constant [Ref. 4]. Clients are given only the interface, which contains only the public functions of the class and a means to create the instances of real implementation on behalf of the client. All mechanisms that try to solve the C++ binary versioning problem use this idea.

C. CURRENT SOLUTIONS FOR THE VERSIONING PROBLEM:

1. C Wrappers Around C++ API:

This approach separates interface from its implementation by wrapping the C++ implementation class with an interface written in C. The interface contains a corresponding "C" function for each public function of in the C++ class. The implementation of the "C" functions in the interface simply delegates the call to the corresponding C++ methods.

Clients of the implementation class are given only the interface, which is simply a header file. Since the client's project is built against this header file, there is no way for compiler to make any predictions about the binary layout of the class. It allocates memory space for a pointer to the class. Since a pointer to any type occupies only an integer size memory space, this memory allocation is always in the same size even if the

binary layout of the class changes. Thus the implementer of the class is free to make any changes to his implementation class provided that he does not change any function signature in the interface.

Even though it solves the versioning problem this method has two serious drawbacks. Especially for the large classes that have tens or even hundreds of functions, writing a new function in the interface that does nothing but forwards this call to the implementation class is burdensome. This also adds more memory requirements for the overall project.

The second disadvantage of this method is that since an interface is just a list of the functions in the implementation class, inheritance cannot be used in the interface. This means that if an implementation class inherits from several classes directly or indirectly, every function in these base classes must also have a corresponding function in the interface. Of course this makes the problem even more burdensome.

2. CORBA (Common Object Request Broker Architecture):

The principle goal of this approach is to overcome network problems resulting from technology changes, platform and operating system differences. CORBA supplies a balanced set of flexible abstractions and concrete services needed to realize practical solutions for the problems associated with distributed heterogeneous computing. It provides platform independent programming interfaces and models for portable distributed object oriented computing applications [Ref. 5]. It is independent from programming languages, computing and networking platforms.

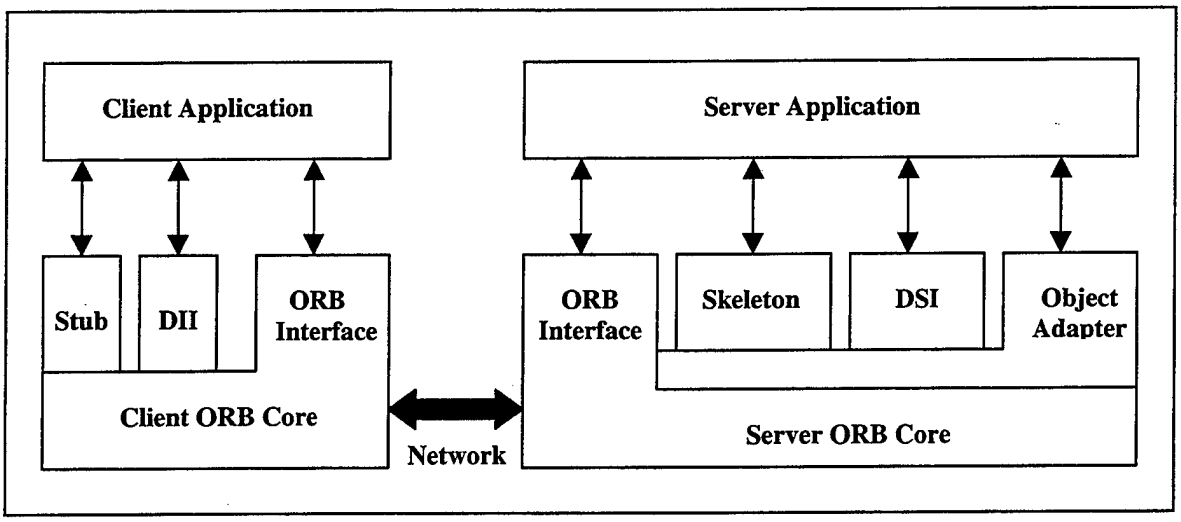

Figure 2.1. Common Object Request Broker Architecture (CORBA) [From Ref. 5]

Figure 2.1 shows the general request flow diagram of CORBA. As shown in the figure there are two running applications in a CORBA design: the server and the client. Server is a pool of objects. The implementer of the server side writes his interface that his object will support by using Object Management Group (OMG) Interface Definition Language (IDL). Compilation of this IDL file with an IDL to target programming language will convert IDL definitions to this language and package them in several files (skeleton, stub, etc). The skeleton produced by IDL compiler will provide a connection between implementer's application written in a programming language and Server ORB Core. After getting this skeleton source file the programmer implements his server with a main loop that accepts client connections.

To be able to make use of the server, clients must be provided the interface written in IDL. Once a client gets this file, like server side, he also compiles it with IDL to his language compiler. This compiler does not necessarily have to be the same with the one that was used in server side. Client can use, for instance, an IDL to Java while

server implementer uses an IDL to C++ compiler. The stub that created by IDL compiler links client's application to underlying Client ORB Core.

Client and server applications can be on the same machine or on different machines on the network. To make a request for an object the client first makes a connection to the server. After connection is established, the client directs requests to the Client ORB Core by using a stub. This request propagates to the server via Server ORB Core and the skeleton. If server supports the requested object, a reference to the object is sent back to the client by using the same path.

As can be seen from this request-replay mechanism, there is no direct talk between client and server applications. The client does not need to know any implementation detail on the server side. Each request and reply is passed via underlying CORBA mechanisms. Since the client application is not built against precise declaration of the object, it is not subject to the versioning problem. The server application can always be modified without the requirement of client-side recompilation.

3. COM (Component Object Model):

a. Definition And Fundamentals:

COM is a component software architecture that allows applications and systems to be built from components supplied by different software vendors. This architecture is used for defining components. Like CORBA, it is not a programming language, thus the implementation of the object is made by using a programming language, like C++ or Java. COM is also the technology that one uses for manipulating components, but in a similar vein, it is not a technology that one uses for client programs.

It can be seen as a shell that one wraps around components, and that defines how communication is passes between clients and the component object [Ref. 7]. This idea is shown in Figure 2.2.

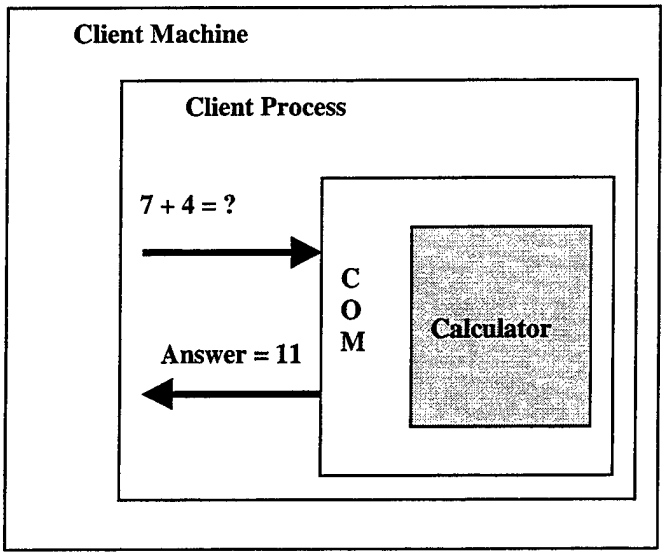

Figure 2.2. COM Logic [After Ref. 7]

COM defines several fundamental concepts that provide the model's structural underpinnings [Ref. 2]. These include:

- A binary standard for function calling between components by using virtual functions and virtual function tables.
- A provision for strongly typed groupings of functions into interfaces by using Interface Definition Language (IDL).
- A base interface providing runtime type discovery and reference counting.
- A mechanism to uniquely identify components and their interfaces.
- A component loader to set up component interactions.

Following sections will explain each of these concepts in detail.

b. Binary Standard For Function Calling:

Like the versioning problem, name mangling is another serious problem with C++ that makes it harder to create portable binary components with this language. Name mangling is simply the difference between function naming conventions among the C++ compiler vendors.

To support function overloading, C++ compilers convert a function's name to a compiler-understandable form. The problem results from the fact that there is no standard between the C++ compilers about how this new name will be given. So every compiler uses its individual naming convention and exports the functions with different names. For instance, the function

```
void setNumber(int p_number);
```

will be named by Microsoft Visual C++ 6.0 as

```
?setNumber@@YAXH@Z
```

whereas Borland C++ 5.0 compiler exports it as

```
@setNumber$qi
```

On a platform where the compiler being used is the same with the one that was used to implement a dynamic link library, the name mangling problem will not occur. But if these two compilers are different, then the client of the dynamic link library will not be able to call the functions in the library.

COM solves this problem by not calling the functions with their names. Instead, it uses a virtual function table and virtual table pointer to invoke the functions on a COM object. As it will be explained in the next section, one of the basic COM requirements is that the interface must be composed of only pure virtual functions and nothing else [Ref.

4]. The implementation of the object will inherit from this interface and implement its functions.

Virtual functions are special in C++ and treated differently by the compiler. If a class has virtual function(s) or it is inheriting from a class that has virtual function(s), the compilers creates a table of virtual functions named "virtual function table" and keeps the names and the corresponding implementation location of the virtual functions in this table. It also adds a new data member, named "virtual table pointer" to the class definition. This pointer is transparent to the programmer and points to the first entry in the virtual function table [Ref. 4]. For example;

```
Class my_interface {
   virtual void setNumber(int p_number) = 0;
   virtual int getNumber(void) = 0;
};
class my_implementation: public my_interface {
   void setNumber(int p_number);
   int getNumber(void);
   virtual void printNumber(void);
};
```

When my_implementation class is compiled, its virtual function table will look like Figure 2.3.

The order of the functions in the virtual function table is standard for every C++ compiler and the rule is as follows: if the class has a base class that has virtual function(s), the first declared virtual function in the base class goes into the first entry in the table, and the second virtual function goes into the second entry and so on. After the base class functions, the first virtual function in the class declaration goes into the next

entry and the other virtual functions follow it according to their declaration order. If the class does not have a base class that has a virtual function, the first virtual function in the

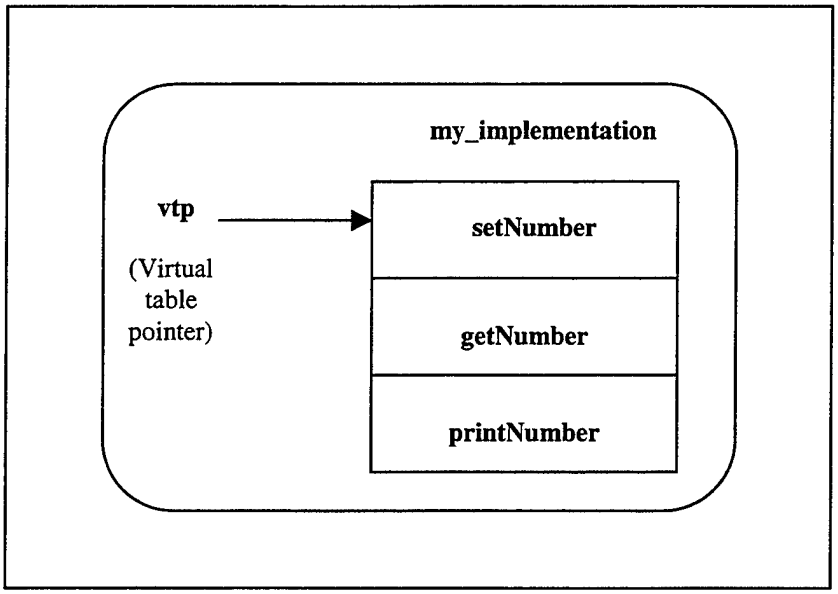

Figure 2.3. Virtual Function Table of "my_implementation"

class goes into the first entry in the table and the other virtual functions follow it according to their declaration order.

When a function on a COM object is called, COM de-references virtual table pointer to get the first entry address of the objet's virtual function table. Then it retrieves the declaration order of the function by looking at the interface. To find the function's entry in the table it adds this number to the first entry's address. Once it gets the function's entry it invokes it by de-referencing the pointer in this entry.

As it can be seen from this function invocation mechanism, name of the function is not important to invoke it anymore. It is just an entry in virtual function table. So an object compiled with a C++ compiler can be used by another C++ compiler without worrying about the name mangling problem.

c. *Interfaces And Interface Definition Language:*

Besides providing compiler independence, creating language independent binary components is another main goals of COM. Like CORBA, COM suggests that a binary component must be able to be implemented and used by any language. To achieve this, it makes another separation: separation of the language for defining interfaces from the one for defining the implementation. For defining interfaces, COM uses Microsoft Interface Definition Language (IDL) [Ref. 4].

Before implementing an object, the implementer creates an IDL file and writes the interface of his object into this file by following the IDL rules [for these rules see Ref. 2]. This file is a contract between him and the clients of the object. He can never change this file once he issues it. These changes include almost every thing, such as function signatures, function declaration orders, and object's physical name. But, as it will be described later in this chapter he can add new functions at the end of the interface. He compiles this interface with Microsoft IDL compiler (midl). This compilation creates necessary files for communication between implementer's application and COM. He implements his COM object by using a programming language and issues his IDL file and the dynamic link library (or application) to his clients.

When a client gets the IDL file, he also compiles it with midl. This compilation will produce necessary files to communicate with COM, create instances of the object and invoke the functions on it. Again, the client does not need to know anything about the implementation details or binary layout of the object. Since he compiles against the interface, the object is free from versioning.

d. Run-time Type Discovery:

COM provides another very useful mechanism for the object implementers: it allows them to add new interfaces to their objects after they issued their IDL files and dynamic link libraries.

This mechanism is very useful in that, most of the time an implementer cannot see the future requirements very well. After the objet's delivery new requirements can make him to wish that he had some other functionality on his object. Thanks to COM's this mechanism he can add this new functionality into his object for his new clients but at the same time, without loosing his old clients.

The mechanism works as follows: the implementer writes the functions in a new interface. This interface inherits from IUnknown, which is the base interface for all COM interfaces and will be discussed later in this section. In the implementation class, he inherits from this interface by writing its name in the inheritance list *after* the base classes he is currently inheriting from. He implements these new functions in the same file with the old ones, then recompiles his object. As discussed previously, since the new interface is the last entity in the inheritance list, its function entries go into the end of the virtual function table. Since the current configuration of the virtual function table does not change with this compilation, old clients are not affected from this versioning. On the other hand, since they compile against the new version, new clients can make use of the new functions.

Serving different kinds of clients (old and new) may cause a problem here. How can a client understand whether the object is in a specific version, if he does not have

documentation? COM solves this problem by giving the clients the chance of interrogating the object about its version at run-time. It uses a special interface for this purpose: IUnknown.

IUnknown is the base interface for all COM interfaces. Every interface must explicitly inherit from it and every COM implementation class must implement its functions as well. IUnknown has three pure virtual functions:

```
HRESULT QueryInterface([in] REFIID riid, [out] void** ppv);
ULONG AddRef();
ULONG Release();
```

QueryInterface is used for run-time type discovery. It gives the semantics of dynamic cast operator in C++ [Ref. 4]. By using this function, clients can switch from one interface to another one on the same object. If the object does not support a requested interface, this means that the version of the object is different from the one the client is asking for, thus the client cannot use those functions with this version.

Usage of QueryInterface function can cause a confusion for the programmers: they may assume that they are creating a new object each time they call this function. But in reality, they are not creating a new object but changing their viewpoint to the object and looking at it from a different angle. As a result of this misunderstanding, they may delete a pointer to the object after they are done with it, without thinking that this deletion voids all the other pointers to the same object too. After this point, if they try to de-reference a pointer to the object (which they misleadingly believe that it is still there), this will crash the program [Ref. 4]. So the clients must be very careful about this issue. They must always keep track of which pointer is pointing to which object. Especially for

large programs, this is not so easy and that is the reason for the existence of AddRef and Release functions in IUnknown.

AddRef and Release functions are used for reference counting. When the client gets a new pointer to the object, he calls AddRef function to increase the reference count. Like wise, when he is done with a pointer, he calls Release function to decrement reference count. If the reference count reaches to zero, this means that nobody is using the object anymore and its time for the object to destroy itself. The implementation of Release function will call the object's destructor if it is so.

The responsibility of calling AddRef and Release functions at appropriate time is on the client. Even though these functions rescue him from keeping track of each pointer in the program, still he must be careful about when to call these functions. When he fails to use AddRef function appropriately, the reference count does not increment, so the object can be destroyed at an unexpected time. When he fails to use Release function appropriately, this will make the reference count inaccurate too, so the object will never have a chance to call its destructor, thus memory leakage occurs. [For common situations that require calls to these functions see Ref. 4]

e. Component and Interface Identifications:

Another mechanism that is provided by COM is giving universally unique names to the interfaces and the binary components. Not having the same name with another interface or component in the world is crucial in that even if it is very small, there is always a chance that these two interfaces or components may cooperate in a particular application. If this is the case, since there is no other way for COM to create an object

and invoke its functions other than by using the physical names, it will not be able to distinguish these two interfaces or objects. So it will fail to create the object or call the function of the interface.

COM provides this uniqueness with one of its API: GUIDGEN.exe. This program uses the network interface card number of the machine and the current time [Ref. 2] to produce a 32-digit hexadecimal number. This is called "the physical name" of the object. Since this is almost impossible for the programmer to use this number every time he wants to refer to the object, COM allows him to give human-readable aliases to this physical name and use it throughout his program.

f. Object Registration and Creation:

To be able to create instances of objects, COM keeps a database of class identities and corresponding dynamic link libraries under HKEY_CLASSES\CLSID directory of Windows Registry. After creating a COM object, implementer must register it with Windows registry so that the clients can use it. A registration creates a new entry under HKEY_CLASSES\CLSID directory with the physical name of the object with a value of absolute path to the dynamic link library that implements the object. An example of such an entry is shown in Figure 2.4.

Clients can create a COM object by using COM API function CoCreateInstance. When a call is made to this function, COM searches Windows registry for the requested object. Once it finds the entry to the object in the registry, retrieves the path to the corresponding dynamic link library. By using this path, it loads the library into the

memory if it is not already there and invokes the object's factory to create an instance of the object. If all these steps are successful, the client gets a pointer to the object.

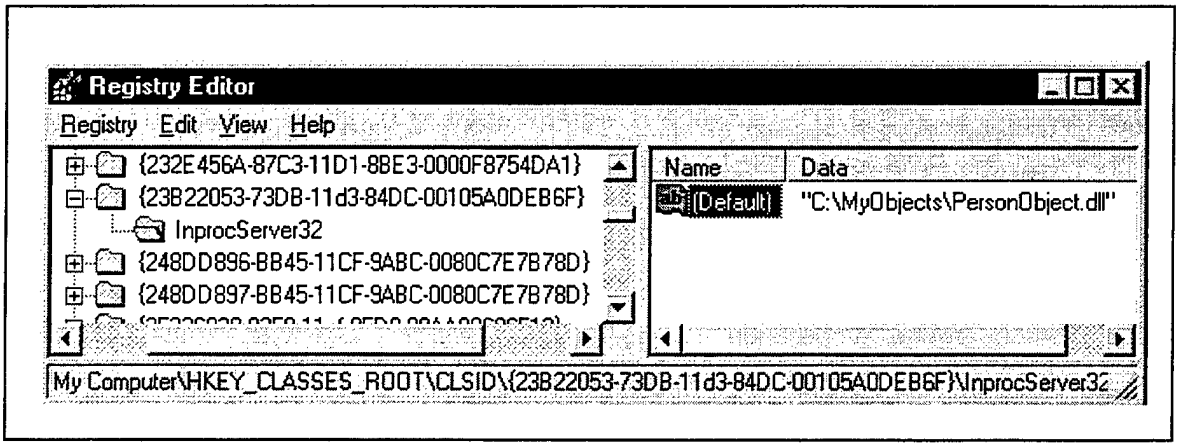

Figure 2.4. Person Object's Entry in the Window's Registry

g. Reusability:

Since what all known about a COM object is its interface, a COM object implementer cannot simply inherit from another COM object to reuse it. Instead he must use containment or aggregation for this purpose [Ref. 4].

In the case of containment, COM object implementer keeps a pointer to the reused object and writes a corresponding function into his implementation for all functions in the reused object. Implementations of these functions are very straightforward and simply forward the calls to the reused object via its pointer. Thus the reusing object uses the reused object's services to implement itself. Containment is shown in Figure 2.5.

Like it was in the case of C Wrappers around C++ API, the requirement for the reused object to have a corresponding function for each function in the reused object is the disadvantages of this method.

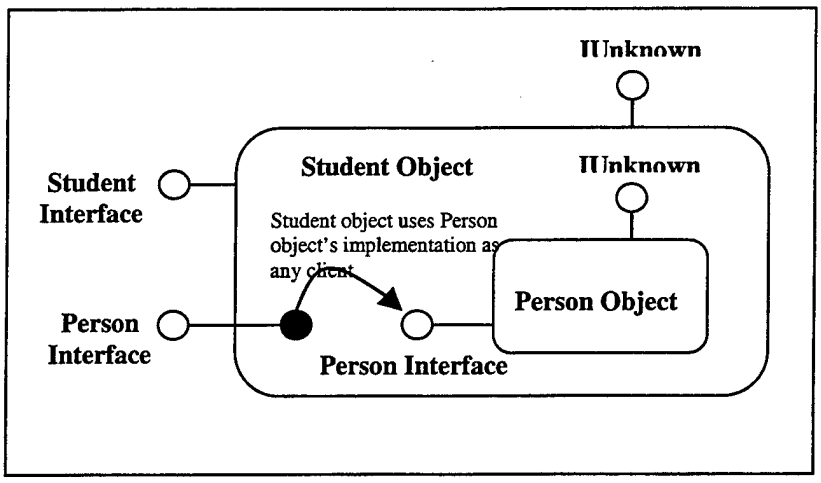

Figure 2.5. Containment [After Ref. 6]

The second and more powerful reuse mechanism is called "aggregation". In this method, the reusing object exposes interface from the reused object as if they were implemented on the reusing object itself. It is actually a specialized case of containment and is available as a convenience to avoid extra implementation overhead in the outer object [Ref. 2]. This method creates a "is-a" relationship between two objects. Aggregation is shown in Figure 2.6.

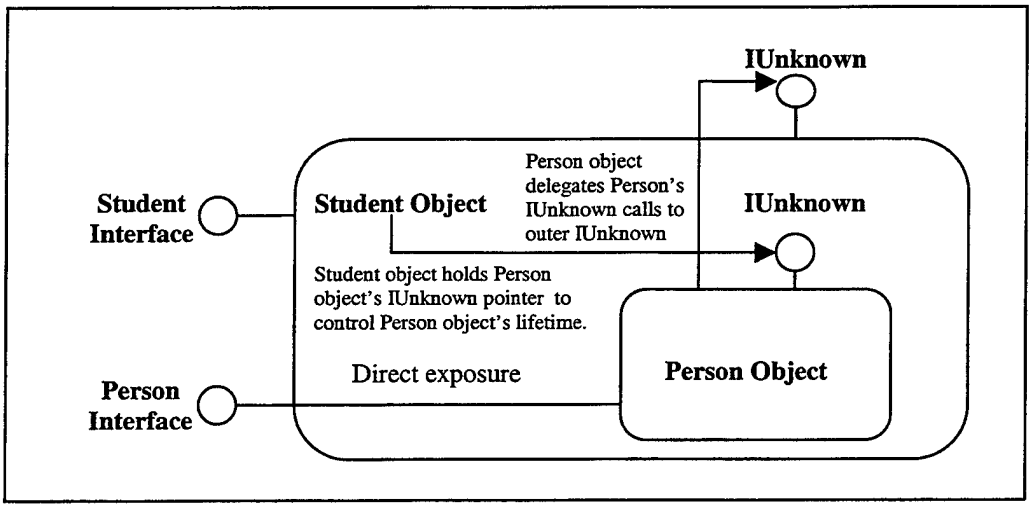

Figure 2.6. Aggregation Relationship Between Student and Person Objects [After Ref. 6]

To be able to use aggregation, the reused object must have been implemented as an "aggregatable" object. Implementing an aggregatable object requires some extra work. But it is always beneficial to do this extra work and make COM object aggregatable. [For the rules of how to create an aggregatable object see Ref. 4]

D. MICROSOFT AND THE VERSIONING PROBLEM:

Microsoft has frequently suffered from the versioning problem [Ref. 8]. In 1980s, C++ libraries were distributed as source code form. The users of the library would be expected to add the implementation source files to their make system and recompile the library sources locally using their C++ compiler. But, as it was discussed in Chapter I, the net effect of this procedure was that the executable code of the library would be bundled as part of the overall client application [Ref. 4]. Furthermore if a defect was found in the implementation and was fixed, the only way for the clients to get use of it was to get the new source code and recompile their application.

To solve these problems Microsoft stated to package the implementation into dynamic link libraries (DLLs). With this technique all of the methods of a class were added to the list of the corresponding DLL, allowing runtime resolution of each method name to its address in memory. Instead of containing the actual code, the import library simply contained references to the file name of the DLL and the names of exported symbols. Clients needed to add only this fairly small import library to their build. They did not need to bundle the real implementation (which was packed in the DLL and would be linked dynamically) to their application anymore. Furthermore, if an error was fixed

in the implementation, the only thing that a client had to do was to get the new version of the DLL and delete the old one [Ref. 4].

Even though this approach solved the disadvantages of distributing the implementation in source code form, it brought its own drawbacks. Once the decision was made to distribute the C++ classes in DLLs, Microsoft was faced with the versioning problem. The rule with DLLs was "...if two or more DLLs export the same functions (immutability), you can link either. However, a single application cannot use both DLLs, nor can they both reside on the same computer..." [Ref. 9].

If a new application had some DLLs that were used by existing applications on the same platform, the installation process would override the old DLLs with the new versions. Then, all the applications that used these DLLs would link against the new versions. Even a single change to the binary layout of the class would make existing applications unable to run, because of the reasons that were discussed in Chapter I.

To overcome this problem Microsoft implemented and started using COM in C++ class implementations. COM alleviated the drawback of using DLLs by allowing different applications to use more than one COM servers with identical interfaces (therefore identical methods). Even though these COM servers "looked like" the same, in reality, due to their different class IDs (CLSIDs) they were different binaries, thus could reside on the same platform [Ref. 9].

E. MOZILLA AND THE VERSIONING PROBLEM:

Mozilla is an open source web browser, designed for standards compliance, performance and portability. Mozilla was the original code name for the product that came to be as Netscape Navigator and later, Netscape Communicator. Now it is intended to be used as the generic term referring to the browser derived from the source code of Netscape Navigator [Ref. 10].

The Mozilla browser consists of pluggable components implemented in C++ and packaged into modules. These components can be dynamically changed. This means that the architecture is scalable [Ref. 11] and dynamically extensible. Like every dynamically extensible system Mozilla is subject to the versioning problem. When a C++ class is changed in one of the modules, all depending modules must also be recompiled. Of course this is a very serious threat for its extensibility.

To overcome this problem, Mozilla uses Cross Platform Component Object Model (XPCOM) to tie its modules together [Ref. 11]. XPCOM is a simplified version of COM and implemented by Mozilla Organization. The main differences between COM and XPCOM are, the latter supports only C++ and is portable. It comes as a stand-alone module in Mozilla architecture. Every module owner in Mozilla uses XPCOM rules to implement C++ classes in his module. The modules that were implemented before XPCOM module are being converted to XPCOM compliant modules [Ref. 12].

F. SUMMARY:

This chapter explained the importance of separating interface from implementation for C++ applications. It emphasized that the reason for the versioning problem stemmed from giving implementation details to the clients. Then it looked at three solutions (C wrappers, CORBA and COM) for the versioning problem that separate interface from implementation.

III. DESIGN

A. INTRODUCTION:

This chapter will explain why COM and Cross Platform Component Object Model (XPCOM) are better solutions than using C Wrappers around C++ API and CORBA for Bamboo's versioning problem. Then design issues for there new Bamboo modules, NonCOMModule, COMModule and XPCOMModule will be discussed.

B. PROPER SOLUTION FOR BAMBOO'S VERSIONING PROBLEM:

As was discussed in the previous chapter, currently there are three different solutions for the versioning problem. While choosing the best option for Bamboo, the following criteria were used:

1) The solution must be appropriate for object oriented programming design techniques.
2) Bamboo modules package the implementation into dynamic link libraries. Thus the solution must support in-process interoperations.
3) There must be a standard between different versions of the solution and they must not conflict with each other.
4) The problem with Bamboo is the C++ binary versioning problem. Anything else that is supported by the solution is unnecessary for Bamboo.
5) The solution must be easily integrated into Bamboo with minimum overhead.

6) The solution must be easy to use.

7) The solution must be portable.

1. Using C Wrappers Around C++ API:

Even though it is currently used in Bamboo's kernel, one can easily say that using C Wrappers around C++ API is not a good solution for Bamboo's modules. It voids most important object oriented techniques like inheritance and function overloading.

With this approach, every function in the implementation class requires four steps of programming:

1) Declare the function in a C++ header file.
2) Implement the function in a C++ source file.
3) Declare a corresponding function in a C header file.
4) Implement this function in a C source file to call its corresponding function in the implementation class.

These steps add too much programming overhead for the programmer. As was mentioned in the previous chapter, every public function in the implementation class must have a corresponding function in the interface. Especially for the large classes that have tens or maybe hundreds of functions, the programmer must explicitly write and implement them for the interface too. Worse than this, since the interface is just a list of public functions in the implementation class, the programmer cannot use inheritance in the interface. He must also explicitly declare and implement all the functions that the implementation C++ class is inhering from its base classes.

Another disadvantage of this method is that since C style linkage is used, the functions cannot be overloaded in the interface. This means that if an implementation class has two functions with the same name but differ in signature, such as:

```
void setNumber(int p_number);
void setNumber(float p_number);
```

the programmer must use different names for these functions. For example:

```
void setIntegerNumber(int p_number);
void setFloatNumber(float p_number);//New name
```

Yet another drawback with this method is that clients of the implementation class loose some of their C++ syntax. For example they cannot use the following very natural piece of C++ code anymore as it is:

```
ClassA* my_ClassA();
my_ClassA→setNumber(10);
```

Instead they must use the object as a parameter in setNumber function:

```
ClassA* my_ClassA = Create_ClassA();
setNumber(my_classA, 10);
```

Of course this is not a well-known and good style for C++ programmers.

With these serious drawbacks, using C Wrappers around C++ API fails in the first fifth and sixth criteria: It is not appropriate for object oriented programming techniques, it requires a lot of overhead to integrate into Bamboo and it is not easy to use. This approach can not be a good solution for Bamboo.

2. COM and CORBA:

After eliminating C Wrappers around C++ API, a decision must be made between two remaining options: COM and CORBA. These two models are competitor for each other and there is a big war in the market. Each one tries to find the flaws and weaknesses in the other and show that its solution is better. There are lots of papers, debates, articles about COM versus CORBA and most of them are subjective. Deciding which one is really better is not an easy issue by looking at these references. It needs deep understanding of both models and maybe several years of experience with each of them. Comparing COM and CORBA is not in scope of this study. But still we can decide which one is *better for Bamboo* by looking at them through our needs and criteria.

1) The solution must be appropriate for object oriented programming design techniques: Both COM and CORBA and their IDL tools (midl and ORB IDL to C++ compiler) are appropriate for object oriented programming techniques.
2) The solution must support in-process interoperations: COM began as a way to let client programs linked to object implementations dynamically; i.e., at execution, incorporating them into a single address space. The implementations were packaged in dynamic link libraries [Ref. 13]. It still has this property and writing in-process services does not need too much effort. When an object activated in process, the dynamic link library that implements the object's methods is loaded into client's process and all of the object's data members reside in the client's address space. This makes method invocation extremely efficient, as no process switch is required [Ref. 4].

In contrast to COM, CORBA was designed specifically for distributed systems [Ref. 13]. It packages the objects in out-of-process services and uses client-server architecture.

3) There must be a standard between different versions of the solution and they must not conflict with each other: CORBA is implemented by different vendors and there are variations in each vendor's implementation [Ref. 13]. In contrast, one of the COM's greatest strengths has been that it is defined by single-vendor-implementation (Microsoft).

4) The problem with Bamboo is the versioning problem. Anything else that is supported by the solution is unnecessary for Bamboo: The principle goal of COM is to provide a standard binary layout for the objects. CORBA, on the other hand, tries to solve the problems associated with distributed heterogeneous computing [Ref. 5]. According to Roger Session, who is the author of "COM and DCOM" and an experienced programmer on both COM and CORBA, comparing COM and CORBA is not quite right: "... the word that I use to describe Microsoft architecture is MDCA, for Microsoft Distributed Component Architecture. There are six fundamental technologies that are part of that. Obviously COM and DCOM are two of them. ... So to compare COM and DCOM to CORBA is faulty. If you are to compare these technologies you need to compare these six technologies that make up MDCA to the basic ORB and object services that CORBA provides. ..." [Ref. 14]

As a result for this criterion, CORBA offers too more than Bamboo needs.

5) The solution must be easily integrated into Bamboo with minimum overhead: Currently Bamboo uses Microsoft Visual C++ 6.0 to compile its modules on Windows platforms. Since COM comes with Visual C++, it is already in the Bamboo system, no extra work is needed to integrate it for Windows platforms. On the other hand, all necessary software must be installed on the working platform to make use of CORBA with Bamboo.

6) The solution must be easy to use: One of the biggest disadvantages of CORBA over COM is its complexity [Ref. 13].

7) The solution must be portable: CORBA is platform-independent but COM works only on Windows platforms.

As it can be seen from these results, if it were platform-independent, COM would definitely be a better choice for Bamboo's versioning problem. Especially its in-process activation nature fits very well in Bamboo system. But this disadvantage voids its all other advantages, since portability is one of the principle goals of Bamboo.

At this point a simplified version of COM, which is named as Cross Platform Component Object Model (XPCOM), alleviates COM's this drawback by adding it portability. XPCOM is developed by Mozilla Organization and used in Netscape modules for the versioning problem. Pretty much every thing about COM applies to XPCOM [Ref. 15], except that XPCOM uses its own registry for object registration instead of Windows registry. In contrast to COM, XPCOM supports only C++ [Ref. 16]. This simplification makes it even better for Bamboo, because it directly addresses Bamboo's problem with only C++. Another important difference between COM and

XPCOM is that XPCOM uses XPIDL, which is closely related with OMG IDL, for its interface definitions.

C. UNSAFE MODULES IN BAMBOO:

To be able to show that COM and XPCOM can solve Bamboo's versioning problem, the first thing that must be done is to prove that there is such a problem in Bamboo. This proof is relatively easy, since it does not need any special extra work other than writing and reusing a class in Bamboo. In fact, since no Bamboo module other than the Kernel uses any countermeasures for this problem, all of the modules are currently unsafe in this manner. Showing that these modules are unsafe requires only a counterexample that shows they are affected by the versioning problem.

This study showed that Bamboo modules are subject to the versioning problem by creating three new modules in Bamboo. As it was the reason for their existence, these modules did not use any countermeasures for the versioning. The directory structure of this example under bbKernelModule is shown in Figure 3.1. As can be seen from the figure, NonCOMModule encapsulates these three modules. NonCOMPersonClassModule declares a Person class and implements its functions to manipulate a person database. All the data members of this class were deliberately declared as private or protected, so that no client module could write code that had direct access to them. This declaration also shows that "private" and "protected" keywords in C++ have no effect at binary level.

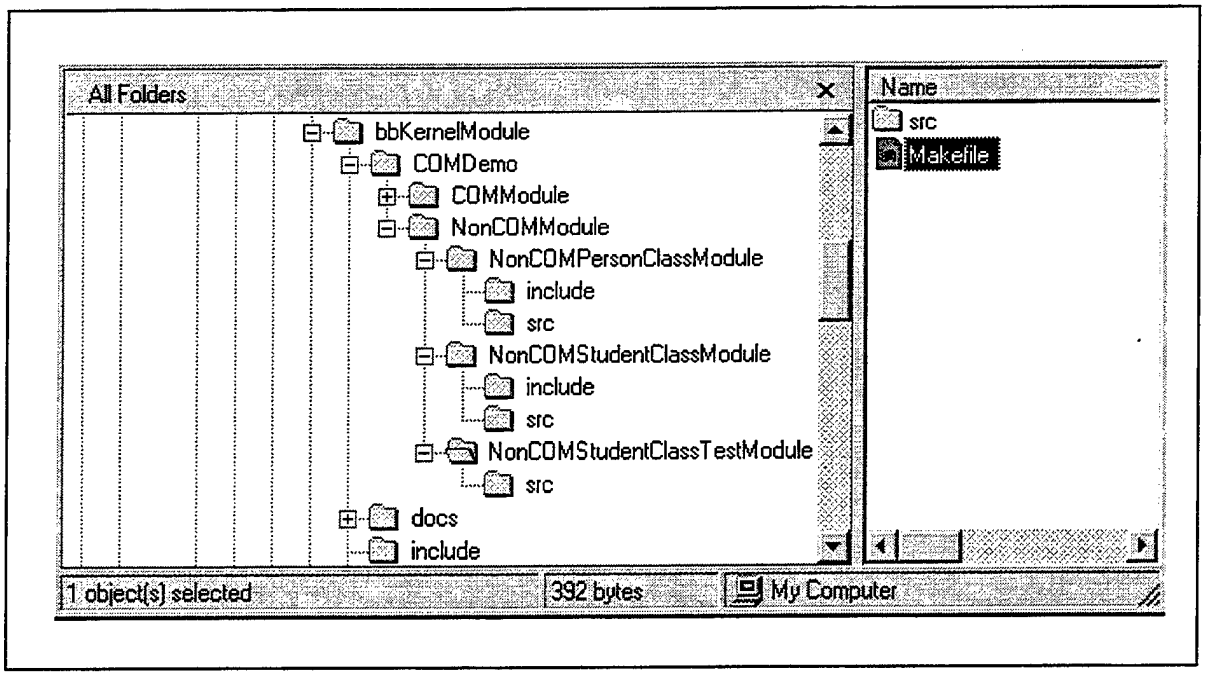

Figure 3.1. NonCOMModule Directory Structure

NonCOMStudentClassModule declares a Student class and implements its functions to manipulate a student database. By depending on NonCOMPersonClassModule, this module reuses the Person class with classical C++ inheritance.

The last module, NonCOMTestModule, tests the Student class by depending on NonCOMStudentClassModule. The first execution of this module had to work without any problems, since NonCOMStudentClassModule was built against the last version of Person class, the binary layout of the Person class was correctly known by the Student class constructor, thus all the memory allocations were as they had to be.

After getting the correct result from the first test, Person class implementer demonstrated a module owner who wanted to make some changes to his class's implementation. He added or removed some data members to or from his class

declaration and changed the function implementations according to these modifications. But he never made any signature changes to his functions, so that his clients could invoke the functions in his class.

With these changes to Person class were in place, test module was executed again without recompiling NonCOMStudentClassModule. Since Student class constructor did not know anything about the new binary layout of Person class, it made memory allocations according to its current knowledge, which is the old version of Person class and not correct anymore. So this run had to prove the existence of the versioning problem in Bamboo by giving wrong results or, according to the location of the change in the declaration, hopefully crashing the program with an error message indicating that some memory location can not be read or written.

Then NonCOMStudentClassModule was recompiled, so that it could get the changes in Person class and test module was re-executed. Because the memory allocations were correct, this time execution had not to encounter any problems.

Even though these tests were enough to be a good proof for the fact that the versioning problem affects Bamboo modules, to show an unsafe module always fails to keep up with changes in the modules that it depends on, several tests were performed with different modifications to the Person class.

D. APPLYING COM AND XPCOM TO BAMBOO:

To show that COM and XPCOM are applicable to Bamboo modules for overcoming the versioning problem, three new modules were created for each model:

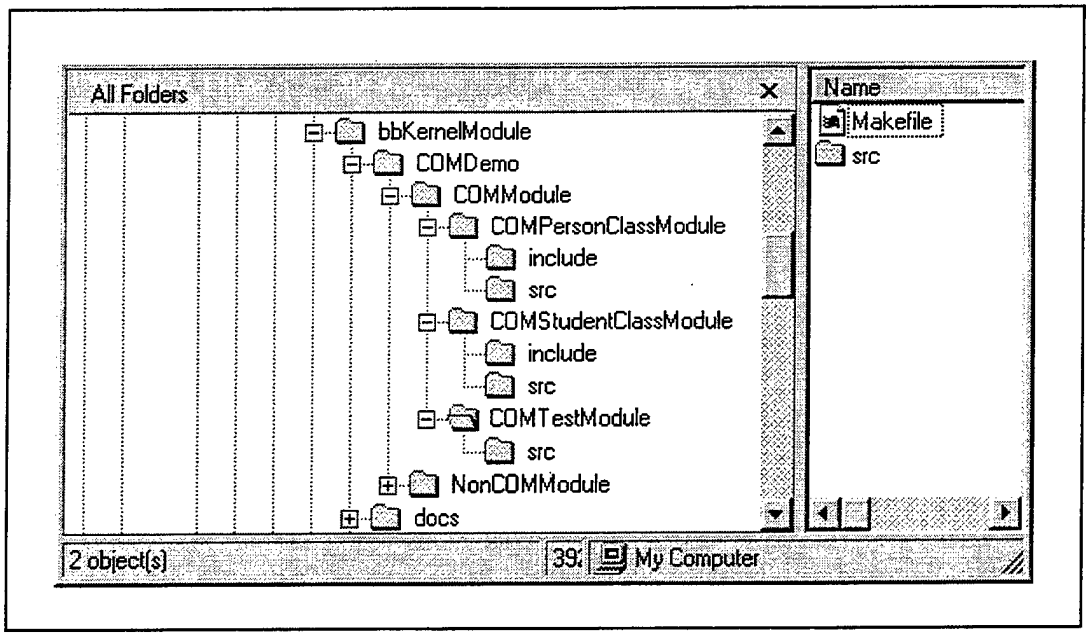

Figure 3.2. Directory Structure of COMModule

[XP]COMPersonClassModule, [XP]COMStudentClassModule and [XP]COMTestModule. The directory structure of COMModule is shown in Figure 3.2.

[XP]COMPersonClassModule implements a [XP]COM Person object that has exactly the same data members and functions with Person class that was implemented in the previous section. Person objet was implemented as aggregatable, so that the clients could get the benefits of "is-a" relationship when they wanted to reuse it.

[XP]COMStudentClassModule implements a [XP]COM Student object that has exactly the same data members and functions with the Student class that was implemented in the previous section. Since classical inheritance in C++ can not be used with [XP]COM, Student object aggregates Person object and reuse it as if it was implemented in itself. This gives the semantics of inheritance and the clients of the Student object are able use them as if Student and Person objects were a single object.

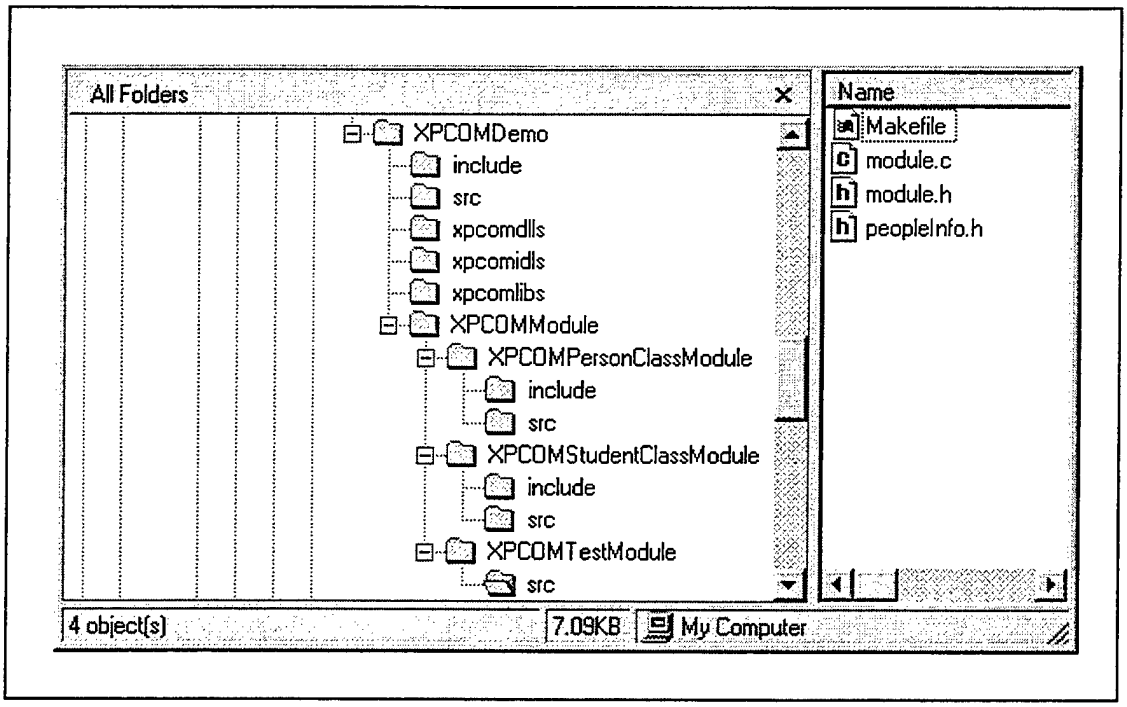

Figure 3.3. XPCOMDemo Directory Structure

Finally [XP]COMTestModule implements a test program for Student object. It creates instances of Student object with [XP]COM creation methods and use them. Test module was be executed several times with exactly the same modifications to Person object that was made in the previous section. These modifications had not to affect the execution of the module anymore, since [XP]COM was used in both Person and Student classes for the versioning problem.

Since it comes with Visual C++, COM is already in Bamboo system for Windows platforms. No special effort is needed to make use of it, except for including the necessary header and library files in the project. How to use these header and library files will be explained in the next chapter. However this is not the case with XPCOM. It is not yet integrated into Bamboo system. So some preparations must be made before implementing XPCOMModule to make use of XPCOM. As a temporary solution, a new

module, which is named as XPCOMDemo was created as a base module for all modules that want to use XPCOM. As can be seen in Figure 3.3, this module includes all necessary header, IDL and library files. All three modules in XPCOMModule specify dependency on this module so that they can use XPCOM header files. "PATH" environment variable was modified so that Bamboo.exe could see XPCOM dynamic link libraries. Appendix A gives a detailed description of this temporary solution.

E. SUMMARY:

This chapter looked at three current solutions for the versioning problem from Bamboo's viewpoint. Due to their disadvantages, using C Wrappers around C++ API and CORBA are not good solutions for Bamboo's versioning problem. COM fits in Bamboo system very well, except for its non-portable nature. Since it gives every thing that we expect from COM, XPCOM is the best solution due to its portability.

To prove that COM and XPCOM can solve this problem in Bamboo modules, first its existence was shown by creating a new module that failed every time when the base class changed and then COM and XPCOM were applied to the same module.

IV. IMPLEMENTATION

A. INTRODUCTION:

This chapter will give the implementation details of NONCOMModule and [XP]COMModule. Source codes of both modules are provided in Appendix C and Apendix D, so only the pieces that need an explanation will be included. Since COM is very similar syntactically to XPCOM, only COMModule's code will be discussed in this chapter and important differences in XPCOM will be addressed.

B. NONCOMMODULE IMPLEMENTATION:

This module demonstrates an unsafe module in Bamboo, which is subject to the versioning problem. It is composed of there sub-modules: NonCOMPersonClassModule, NonCOMStudentClassModule and NonCOMTestModule.

1. NonCOMPersonClassModule:

This module implements a Person class for manipulating a simple person database. To allow the clients of this module to be able to use Person class's header file without copying it into their modules or specifying the path to it, Person.h is put into module's "include" directory. This header file includes the declaration of Person class and Date data type. As it can be seen from this file [see Appendix B], all the functions of the class are exported to dynamic link library by using "BB_Proper_Export_Flag" macro.

All the data members are declared as private to prevent direct accesses. The data member "age" is declared in a "if andif" directive block and is not included in the first version of the Person class. The location of this member variable is deliberately chosen to be forth in the declaration and the reason will be discussed in the next chapter.

The functions of Person class are declared as public, so if a class inherits from Person class it automatically gets these functions too. There is nothing special with the implementation of these functions, except for setBirthDate and getAge. These two functions have two different versions and "#define VERSION2" directive in the beginning of the file defines which version to use. In the first version of Person class setBirthDate function simply sets the birth date of the person with its parameters. In the second version of Person class, it also sets the member variable "age" by subtracting the birth year of the person from current year. If "VERSION2" is not defined, getAge function returns the result of subtracting the person's birth year from year 2000. In the second version of the class, this function simply returns member variable "age", which is added into class definition with this version.

2. NonCOMStudentClassModule:

This module implements a Student class for manipulating a simple student database. It depends on NonCOMPersonClassModule by specifying this in its make file. For the same reason with Person.h, Student.h is put into module's include directory. The class publicly inherits from Person class and gets all the functions in the Person class but not the data members, since they are declared as private. The constructor requires all

necessary information to define a person as well to define a student. All the function implementations are straightforward and there is nothing special to explain.

3. NonCOMTestModule:

This module tests the Student and Person classes. It depends on NonCOMStudentClassModule and NonCOMPersonClassModule. The header file "PersonInfo.h" is used as a simple student database that includes information about several students. "Module.c" file creates instances of students by using this database and displays the student information on the screen.

C. [XP]COMMODULE IMPLEMENTATION:

This module applies [XP]COM to Person and Student classes introduced in the previous section. It is composed of three sub-modules: [XP]COMPersonClassModule, [XP]COMStudentClassModule and [XP]COMTestModule. Since the implementation of Person and Student [XP]COM objects are almost identical, creation of a [XP]COM object in Bamboo will be explained in detail by using Person object and only the differences in Student object will be addressed.

1. [XP]COMPersonClassModule:

This module implements an aggregatable Person [XP]COM object. Person object has exactly the same functionality with Person class introduced in the previous section. It

also has exactly the same data members declared as private but adds some new members that are necessary for implementation of a [XP]COM object.

Creating a [XP]COM object requires the following steps to be taken:

1) Write the interface into an IDL file.
2) Produce necessary header and source files by compiling the IDL file.
3) Declare the implementation class in a header file by inheriting from the interface.
4) Implement all the functions declared in the implementation class (usually only constructor and destructor), in the interface and in IUnknown.
5) Declare a factory class and implement it.
6) Implement global functions necessary for registration, un-registration and creation of the object.
7) Write a make file and compile the module.
8) Register the object.

Even though there seems to be eight steps, most of them are standard and needs only slight modifications to an existent [XP]COM object's implementation. If a programmer has such a draft, he can easily implement his [XP]COM object without too much extra work. Remainder of section will go through these steps one by one and give a guidance for how to implement an [XP]COM object in Bamboo.

a. Writing the Person Object's Interface:

This will be usually the first step in the process, because the implementation class will use the files that are created by IDL compiler form this IDL

file. As a convention, all interfaces have a name with an "I" prefix indication that this is an interface [Ref. 4]. So the interface for Person object will be IPerson.

A requirement for all [XP]COM interfaces is that they must have a 32-digit hexadecimal physical names. This number can be created by typing "uuidgen.exe" on a command prompt or using "guidgen.exe". Both tools come with Visual C++. As a good practice, all numbers for related objects can be produced in one session, so when the objects are registered with Windows or XPCOM registry, these names go successively and make it easy to find them.

For both COM and XPCOM interface declarations start with the keyword "interface" followed by the name of the interface and a *single* base interface. This base interface is usually IUnknown for COM and nsISupports for XPCOM. Since it uses OMG IDL, the body of the interface is a little bit different in XPCOM than that of COM. It uses attributes, which are the names of the member variables in the implementation class. Attributes are not data members that require memory allocations. When the interface is compiled they are converted to "get" and "set" functions with appropriate return types and parameters. For example, the line

```
attribute string name;
```

in Person.idl of XPCOM module, will be converted to

```
NS_IMETHOD GetName(char * *aName) = 0;
NS_IMETHOD SetName(const char * aName) = 0;
```

functions in IPerson.h. However, in COM interfaces these "get" and "set" functions must also be declared explicitly by using appropriate COM data types along with the other functions.

Another difference between COM and XPCOM IDL is that in COM, objects always appear in the context of a library definition [Ref. 4].

```
[ uuid(23B22052-73DB-11d3-84DC-00105A0DEB6F)]
library PersonLib {
      importlib("stdole32.tlb");
      importlib("stdole2.tlb");
      [ uuid(23B22053-73DB-11d3-84DC-00105A0DEB6F)]
      coclass PersonObject {
            interface IPerson;
      };//end PersonObject
};//end PersonLib
```

These definitions are similar to "module" in OMG IDL and used for grouping a collection of data types (e.g. interfaces, COM objects and type definitions) into a logical unit or namespace. Libraries are also given a physical name by using the same tools described above. Each library must import two standard libraries,"stdole32.tlb" and "stdole2.tlb", and then list the objects that they expose. Objects in the library start with a physical name (uuid), and then comes a logical name that is valid only throughout the implementation code. The keyword "coclass" indicates that this is a COM object. The body of the coclass is simply a list of all interfaces that the implementation will support. Since Person class has only one interface, "PersonObject" has only one interface name in its body.

XPCOM IDL simplifies these steps by requiring only the interface declaration and than a physical and a logical name for the object:

```
[scriptable, uuid(162641E1-8F17-11d3-84EA-00105A0DEB6F)]
interface IPerson : nsISupports {
    //Attributes and functions
};//end IPerson
```

```
%{ C++
//Physical and logical names of the object.
// {710E2253-8F57-11d3-8896-F2A82A000000}
#define PERSONOBJECT_CID \
{ 0x710e2253, 0x8f57, 0x11d3, { 0x88, 0x96, 0xf2, 0xa8,
0x2a, 0x0, 0x0, 0x0 } };
%}
```

b. Compiling the IPerson.idl File:

After wring the interface in the IDL file, this file must be compiled in order to get the necessary header and source files for the implementation. COM IDL files are compiled with midl.exe, whereas XPCOM IDL files are compiled with xpidl.exe. To compile IPerson.idl file, the following lines are typed on a command prompt :

```
midl IPerson.idl → For COM IDL
xpidl IPerson.idl → For XPCOM IDL
```

XPIDL will produce only a header file (IPerson.h) and this contains all necessary information for the implementation to communicate with XPCOM. On the other hand, since it supports multiple languages, MIDL will produce five new files. Among them only IPerson.h and IPerson_i.c are necessary for Bamboo implementations, the other files can be deleted. These header and source files are put into "include" directory so that the depending modules can use them.

c. Declaration of the Person Class:

Being a [XP]COM object gives a little bit complexity to the declaration of the class. Especially for the "aggregatable" objects like Person class, declaration and implementation require a little extra work.

Since the depending modules will never need the header file of an implementation class, Person.h does not have to be put into "include" directory. In fact, doing so may cause confusion for the clients, so it is better to keep it in the source director (src). Person.h includes the header file that was produced by IDL (IPerson.h) compiler, so it has every piece of necessary information it needs, such as how to communicate with [XP]COM and how to refer to the Person object's physical name. The Person class publicly inherits prom the interface (IPerson). The private portion of the declaration includes all data members. All the variables, except the last two, are the same with those of Person class discussed in the previous section. "refCount" is one of the newly added variables and will hold the number of the pointers that uses the Person object. XPCOM does not require such a variable, since it gives two very useful macros, NS_ADDREF and NS_RELEASE, that do the reference counting automatically. The other new data member is "m_pUnkOuter" ("m_pnsISupsOuter" in XPCOM). This data member is called controlling IUnknown and is used to delegate IUnknown function calls (QueryInterface, AddRef and Release) to aggregating object if the Person object is being aggregated.

Since the Person class is implemented as an aggregatable object, it must have a constructor that takes an IUnknown pointer as a parameter and this constructor can be seen in the public section of the class. For COM object, after constructor and destructor the functions of IUnknown and IPerson are explicitly declared. But this is not necessary for XPCOM; when a XPCOM IDL file is compiled, declaration of all functions in the interfaces are packaged into 'NS_DECL_XXX" macro. So the lines

```
NS_DECL_IPERSON
NS_DECL_ISUPPORTS
```

in XPCOM Person.h automatically declare all functions of IPerson and nsISupports.

The InternalQueryInterface, InternalAddRef, InternalRelease functions and the inner class XNDUnknown are also required for only aggregatable objects. Implementation of aggregation can be done in a number of ways [See Ref. 2 for some alternative ways]. This study uses the one that is provided in Ref. 4. This approach uses an instance of "XNDUnknown" inner class to delegate IUnknown methods to appropriate implementation. As it can be seen from the definition of XNDUnknown class, it also has a set of IUnknown functions and they simply call inner functions. When a call to an IUnknown function is made through a Person object pointer, if Person class is not being aggregated, "This" function declared in XNDUnknown class will return a pointer to the Person object and corresponding IUnknown function is called in the Person object. If it is being aggregated, this call will be delegated to aggregating object with another mechanism and this will be discussed in the next step.

d. Implementation of the Person Class:

The implementation of Person class can be seen in Person.c file. An important point here is that COM module's Person.c must include IPreson_i.c file that was created by MIDL compiler. This file includes class and interface identifications and must be compiled before any other implementation file such as Person.c.

The Person class must implement four sets of functions: the functions that are declared in the class itself (constructor and destructor), the functions of IUnknown, the functions of internal IUnknown and the functions of IPerson interface.

The implementation of constructor initializes the reference count to zero and then assigns its parameter to the controller IUnknown (m_pUnkOuter).

```
Person::Person(IUnknown *pUnkOuter) : refCount(0) {
      if(pUnkOuter)
            m_pUnkOuter = pUnkOuter;
      else
            m_pUnkOuter = &m_innerUnknown;
}//end constructor
```

If a COM object wants to aggregate Person object it will call COM API function "CoCreateInstance" by passing a pointer to itself. The COM object creation mechanism will send this pointer to the Person object's constructor as the parameter. If aggregation is not in use the value of this pointer will be null. The "if" statement inside the constructor checks if the caller wants to aggregate the Person object or not. If it wants the aggregation then the controller IUnknown will be assigned to the parameter pointer. If it does not want to use aggregation then the controller pointer will be assigned to inner IUnknown. As a result when a call to any of IUnknown functions is made through controller IUnknown, the pointer will delegate this call to Person object if the aggregation is not in use or it will delegate this call to aggregating object (the outer object) if the aggregation is in use.

The implementation of IUnknown functions QueryInterface, AddRef and Release are straightforward and they simply call the controller IUnknown's corresponding function:

```
STDMETHODIMP_(ULONG) Person::AddRef(void) {
      return m_pUnkOuter->AddRef();
}
```

The corresponding function will be either one of InternalXXX functions of Person object or the same function in the aggregating object depending on the aggregation.

InternalQueryInterface first checks whether the requested interface is supported by Person object or not. The only two possible interfaces that can be requested from the Person object are IUnknown and IPerson. Any other request will result in an error message. As a rule [Ref. 2], to avoid an infinitive loop between inner and outer objects' QueryInterface functions, an aggregating object's QueryInterface must return a non-delegating IUnknown interface if it is asked for an IUnknown interface. In the Person object this rule is checked in InternalQueryInterface. If the requested interface is IUnknown then InternalQueryInterface returns a non-delegating IUnknown by casting its inner IUnknown instance to a IUnknown pointer with the following piece of code:

```
if(iid == IID_IUnknown)
      *ppv = static_cast<IUnknown*> (&m_innerUnknown);
```

If the requested interface is IPerson then the Person object is cast to an IPerson interface and sent back to the caller:

```
else if(iid == IID_IPerson)
      *ppv = static_cast<IPerson*> (this);
```

If the result of InternalQueryInterface function is a non-null value, this means that Person object has gained a new pointer, thus reference count must be incremented. As a rule [Ref. 4] AddRef function must be called through the pointer that is sent back to the caller. This is seen at the end of the function:

```
((IUnknown*)*ppv)->AddRef();
```

And finally if the result of InternalQueryInterface is successful then a success message is returned to the caller, indicating that he can safely use the Person object.

The InternalAddRef and InternalRelease functions simply increment / decrement the reference count. If the reference count reaches to zero, this means that there is no remaining pointers to the Person object, thus it can be destroyed, so InternalRelease function calls the Person object's destructor.

The last set of functions that must be implemented by Person class is the ones that are declared in IPerson interface. These functions are exactly the same with those of Person class discussed in the pervious section. Again "getAge" and "setBirthDate" functions have two versions and works as the same.

e. Declaration of PersonFactory:

A basic requirement for all [XP]COM objects is that they must have a class factory. A class factory is a special [XP]COM object whose main purpose is to implement IClassFactory (or nsIFactory in XPCOM) interface [Ref. 4]. Like IUnknown, IClassFactory is also another standard COM interface. It inherits from IUnknown and declares two new functions:

```
STDMETHODIMP CreateInstance(IUnknown* outerIUnkown,
                            REFIID riid, void** ppv);
STDMETHODIMP LockServer(BOOL lock);
```

CreateInstance function plays an important role in object creation mechanism in [XP]COM. When a client makes a call to [XP]COM API function CoCreateInstance, at some level [XP]COM creates the object by delegating this call to the requested object's class factory's CreateInstance function.

LockServer function is used by clients to keep a server in memory even when it is servicing no objects. Even though, a class factory implements reference counting

through IUnknown's AddRef and Release functions, some objects, such as local servers implemented in an executable file do not count references to the class factory object as sufficient to keep the code to the object in the memory; they exit when nothing but a class factory remains. Clients can keep these kinds of objects by calling LockServer on the class factories that they wanted to keep after creating them [Ref. 4]. There is no need to use this function for Bamboo developers. In fact, this function is used to improve performance of object instantiations. Most clients have no need to call this function. It is present primarily for the benefit of sophisticated clients with special performance needs from certain classes [Ref. 2].

As a convention class factory has the same name with the [XP]COM object with a "Factory" extension. So the name of the Person object's class factory will be "PersonFactory". It can be declared and implemented in the same header/source files with Person class or in new header/source files. For clarity, this study chooses the latter option for the COM object, declares it in PersonFactory.h and implements it in "PersonFactory.c" files. For XPCOM object it is declared in Person.h file and implemented in Person.c.

The implementation of CreateInstance first checks if the caller wants to aggregate the Person object. If it does, as a rule [Ref. 2] the outer object must explicitly ask for IUnknown when creating the Person object. The Person object will fail to be created if it is asked for an interface other than IUnknown. This rule is checked by the first "if" statement in the function:

```
if(outer != 0 && iid != IID_IUnknown) {
        *ppv = 0;
        return E_INVALIDARG;
```

```
}
```

If this rule is satisfied, the function creates an instance of Person class and delegates the request to InternalQueryInterface function and returns its result:

```
Person *p = new Person(outer);
HRESULT hr = p->InternalQueryInterface(iid, ppv);
return hr;
```

For Bamboo purposes, the implementation of LockServer function is straightforward and it simply returns "true".

```
STDMETHODIMP PersonFactory::LockServer(BOOL l)  {
      return S_OK;
}
```

f. Implementing the Global Functions:

Both COM and XPCOM require the dynamic link library that holds the implementation to export four global functions. For COM these functions are:

```
STDAPI DllGetClassObject(REFCLSID, REFIID, void**);
STDAPI DllRegisterServer();
STDAPI DllUnregisterServer();
STDAPI DllCanUnloadNow();
```

And for XPCOM these functions are:

```
nsresult NSGetFactory(nsISupports*, const nsCID&,
                      const char*, const char*,
                      nsIFactory**);
nsresult NSRegisterSelf(nsISupports*, const char*);
nsresult NSUnregisterSelf(nsISupports*, const char*);
PRBool NSCanUnload(nsISupports*);
```

COM and XPCOM use DllGetClassObject and NSGetFactory functions as an entry point to the dynamic link library. When a client wants [XP]COM to create an object for him,

[XP]COM finds the location of the dynamic link library from the registry and invokes these functions. Their implementation first checks if this dynamic link library implements the requested object. If it does not, functions return an error message with the following code:

```
if(clsid != CLSID_PersonObject)
        return CLASS_E_CLASSNOTAVAILABLE;
```

If the dynamic link library implements the object, the functions create an instance of the object's factory class, delegate the request to the QueryInterface of this instance and return the result of QueryInterface:

```
static PersonFactory personFactoryObject;
HRESULT hr = personFactoryObject.QueryInterface(iid, ppv);
return hr;
```

DllRegisterServer and NSRegisterSelf are used for self-registration. After implementation of Person object is finished, it must be registered with Windows or XPCOM registry. To achieve the registration [XP]COM invokes these functions. Their implementation creates a new key in Windows or XPCOM registry and puts all necessary information, such as absolute path to dynamic link library, for object creation.

COM and XPCOM use DllCanUnloadNow and NSCanUnload functions respectively to decide whether the dynamic link library is still in use or not. They invoke these functions periodically and if they get a "true" value from these functions, this means that the dynamic link library is not in use and can be unloaded for efficient memory usage. Since Person object is a simple example for COM implementation DllCanUnloadNow always returns a "false" value. But to show usage of some useful macros, NSCanUnload function is implemented differently in XPCOM. It checks if all

the pointers to Person and PersonFactory objects released these objects. If so, it return a true value:

```
PRBool NSCanUnload(nsISupports* serviceMgr) {
      return PRBool(gInstanceCnt == 0 && gLockCnt == 0);
}
```

g. Writing a Make File:

[XP]COMPersonClassModule's make file is a lot like any other make file in Bamboo. There are only two points that must be taken into account while writing it. First, COM and XPCOM libraries must be added to the link. COM uses "ole32.lib" and "advapi32.lib". Since these libraries come with Visual C++, there is no need to specify a path for them. XPCOM uses "nspr3.lib" and its path must also be provided. For a Windows NT platform, these libraries can be added to the link as follows:

```
# In COMPersonClassModule makefile
WINNT_LINK_LIBS     = ole32.lib \
                      advapi32.lib
# In XPCOMPersonClassModule makefile
WINNT_LINK_LIBS=
    ${BB_DIR}/roots/bbkernelModule/xpcomlibs/nspr3.lib
```

The other point that must be considered while writing the make file is related to the global functions that are used in COM Person object implementation. Since these function' are declared in COM header files, their signatures can not be changed. Thus, trying to export them with "BB_Proper_Export_Flag" macro will give a compile-time error. The only way to export them is writing a definition file and adding this file to the link. For Person object this definition file (Person.def) includes the following lines:

```
EXPORTS
      DllGetClassObject        PRIVATE
```

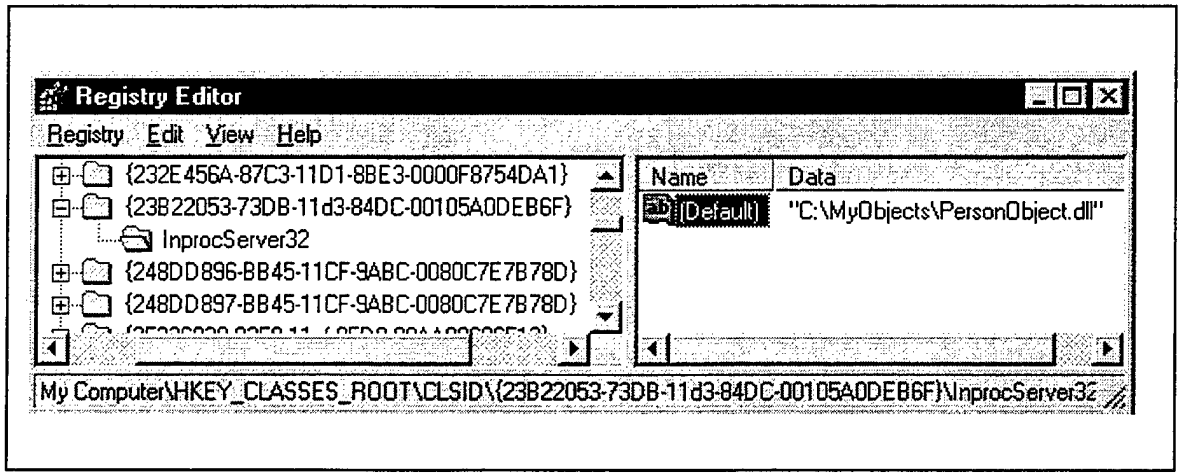

Figure 4.1. Person Object's Entry in the Windows Registry

```
DllRegisterServer       PRIVATE
DllUnregisterServer     PRIVATE
DllCanUnloadNow         PRIVATE
```

Person.def can be added to the link by using "DEFFILE" flag:

```
# In COMPersonClassModule only
DEFFILE = ../../src/Person.def
```

h. Registering the Person Object:

After the module is built, Person object must be registered with Windows or XPCOM registry so that the clients can use it. COM objects are registered / un-registered by using "regsvr32.exe":

```
regsvr32  COMPersonClassModule.dll  → For Registration
regsvr32  -u COMPersonClassModule.dll → For Un-registration
```

Windows registry can be viewed by using "regedit.exe". Person object's entry is shown in Figure 4.1.

XPCOM objects are registered by using "regxpcom.exe":

```
regxpcom  XPCOMPersonClassModule.dll → For Registration
regxpcom  XPCOMPersonClassModule.dll → For Un-registration
```

XPCOM repository can be viewed by using "regexport.exe". Person object's entry is shown in Figure 4.2.

```
### XPCOM_MEM_BLOAT_LOG defined -- logging bloat/leaks to xpcomlog.dat
Registry <default> opened OK.
nsRegistry::Common
nsRegistry::Users
nsRegistry::CurrentUser
Registry <Application Component Registry> opened OK.
nsRegistry::Common
nsRegistry::Common -  CLSID
nsRegistry::Common -  CLSID -  {710e2253-8f57-11d3-8896-f2a82a000000}
            ProgID       = component://bamboo/xpcom/Person
            ComponentType      = application/x-mozilla-native
            InprocServer                                              =
abs:E:\Thesis\BambooDir\Bamboo\roots\bbKernelModule\XPCOMDemo\XPCOMMod
ule\XPCOMPersonClassModule\lib\winnt\XPCOMPersonClassModule.dll
            ClassName    = Person Object
```

Figure 4.2. Person Object's Entry in XPCOM Registry

2. [XP]COMStudentClassModule:

This module implements an Student [XP]COM object. Student object has exactly the same functionality with Student class introduced in the previous section. It also has exactly the same data members declared as private but adds some new members that are necessary for implementation of a [XP]COM object. Two main differences between Student and Person objects are Student object is implemented as non-aggregatable and it reuses Person object by aggregation.

Since it is non-aggregatable, declaration and implementation of Student class is relatively easier than Person class. It does not need a controller and inner IUnknown. It has to implement only one set of IUnknown functions and the implementation of these

functions is pretty much the same with InternalXXX functions of Person class. The only different point that must be considered with the implementation of QueryInterface function is that it must also support IPerson requests. Because, Student object aggregates Person object and the clients of Student object see them as a single [XP]COM object. Student class supports this request by having a new data member "m_Person", which is a pointer to an instance of Person class, and delegating the request to the Person class's implementation by invoking QueryInterface function through this pointer:

```
if(iid == IID_IPerson)
      return m_Person->QueryInterface(iid, ppv);
```

Initialization of this Person object pointer is done during the Student object's creation in its constructor:

```
Student::Student() : refCount(0) {
      HRESULT hr = CoCreateInstance(CLSID_PersonObject,
                                    this,CLSCTX_ALL,
                                    IID_IUnknown,
                                    (void**)&m_Person);
}
```

As it can be seen from the implementation of the constructor, Student class reuses Person class as any client. It creates an instance of Person object by using COM API function CoCreateInstance. The first parameter (CLSID_PersonObject) of this function is specifies the requested object from the registry with an initial interface specified by the forth parameter (IID_IUnknown). The second parameter (this) is used for aggregation. Since Student object wants to aggregate Person object it sends a pointer to itself via this parameter. If it didn't want to use aggregation, this parameter would be a "null" value. The third parameter (CLSCTX_ALL) shows whether the object should run in process, out of process or on a different host machine. For Bamboo implementations there is no

need to deal with the details of this parameter and simply a value of "CLSTX_ALL" can be used. The last parameter of the function (m_Person) is the pointer of type Person class with an initial value of "null". If CoCreateInstance successes, it will hold the address of an actual Person class instance.

The last major difference between Student and Person objects' implementation is related to their class factories' CreateInstance function. Since Student object is implemented as a non-aggregatable object, the implementation of CreateInstance function must reject an aggregation request. The first "if" statement checks if the first parameter is a non-null value. If it is, this means that the caller wants to aggregate Student object, so it returns an error code indicating that Student object cannot be aggregated:

```
STDMETHODIMP
StudentFactory::CreateInstance(IUnknown* outer,
                                REFIID iid, void **ppv) {
      if(outer != 0) return CLASS_E_NOAGGREGATION;
      //Other Code
}
```

3. [XP]COMTestModule:

This module tests the [XP]COMStudentClassModule. It conducts exactly the same tests with NonCOMTestModule. The fact that Person and Student are two different [XP]COM objects is transparent to this module. It simply creates instances of Student object and uses both Student and Person objects through these instances as if they were a single [XP]COM object.

D. SUMMARY:

This chapter gave the implementation details of NONCOMModule and [XP]COMModule. NonCOMModule implemented an unsafe module that was subject to the versioning problem. Then COM and XPCOM rules were applied to this unsafe module to make it free from this problem.

THIS PAGE INTENTIONALLY LEFT BLANK

V. ANALYSIS

A. INTRODUCTION:

This chapter will discuss the results of ten different tests on NonCOMModule and show that it fails every time when it runs with some changes to Person class but without compiling Student class. Then the same tests will be performed on [XP]COMModule to show that these modules are not affected by the versioning problem.

B. NONCOMMODULE AND [XP]COMMODULE TESTING:

To prove that [XP]COM solves the versioning problem in Bamboo modules, ten different tests were conducted to each of NonCOMModule, COMModule and XPCOMModule. Each test changed the binary layout of the Person class in a different way. These changes covered all possible ways that the versioning problem could affect a C++ class and included:

1) Adding new data members to the class definition,
2) Changing the declaration order of the data members,
3) Changing the function implementations according to the new data members,
4) Removing the data members from the class definition,
5) Changing the type of the data members.

Each of these situations was tested with two different modifications. In each test first a logical expectation about the result of the test was written. Then by compiling *only* the

NonCOMPersonClassModule the test module was executed. Results were compared with the expectations. Finally exactly the same modifications applied to Person class in [XP]COMModule and then by compiling only [XP]COMPersonClassModule, the test module was executed.

Tests were conducted on a Windows NT 4.0 platform on which the data types were represented with the following number of bytes:

```
int      → 4 bytes
long     → 4 bytes
float    → 4 bytes
double   → 8 bytes
char     → 1 byte
pointer to any type → 4 bytes (size of integer)
_Date    → 2*long + char* = 12 bytes
```

For the first test the first version of Person class was used. As can be seen from Appendix B, Person class version 1 has the following data members:

```
Date birthDate;
char* name;
char* lastName;
char* address;
```

After each test all the modifications were deleted and Person class turned back to its original version.

TEST 0: This test made sure that all the test modules were implemented correctly in the first place. They were executed without making any modifications to any of the modules.

Expectation: Since all memory allocations are as they are supposed to be, all the three test modules must run without any problem.

Result of Non-COM Test Module: The module worked without any problem.

Results of COM and XPCOM Test Modules: Both of the modules worked without any problem.

TEST 1: In this test a new data member, named "temp" of type "double" was added to the start of the Person class. After this modification the new order of the data members in Person class was:

```
double temp;  //New data member
Date birthDate;
char* name;
char* lastName;
char* address;
```

Expectation: Since new data member is put in front of the existing data members, all of them will be shifted by 8 bytes in the memory. So an instance of Person class will be ended at memory location $m + 32$, where m is the starting location of the instance in the memory. Since the Student class is not recompiled, its constructor knows that Person class finishes at memory location $m + 24$, so it will allocate 8 bytes of memory for the school name and the curriculum of the student starting from the location $m + 24$. This will overwrite last name and address of the person. When getLastName function is called this must return school name of the student and when getAddress function is called, this must return the curriculum of the student.

Result of Non-COM Test Module: All the last names and addresses switched with the school names and curriculums respectively. The program crashed while processing the fifth person. The reason for the crash is that, as it can be seen from the people database, the school name for the fifth person is not specified. The implementation of setSchoolName function sets the school name pointer value to null, if the school name is

not specified. While test module tried to print out the school name of this person, it de-referenced the null pointer, thus the program crashed.

Results of COM and XPCOM Test Modules: Both of the modules worked without any problem.

TEST 2: This time an integer array that had 10 elements was added in front of the data members and then its elements were initialized in the constructor to have a 0 value. After this modification the order of the members was:

```
int temp[10];
Date birthDate;
char* name;
char* lastName;
char* address;
```

Expectation: For the same reasons with the first test, school name and curriculum will overwrite the name and last name of the person. When getName and getLast Name functions are called they must return the school name and the curriculum respectively.

Result of Non-COM Test Module: School name and curriculum of the first student appeared to be the name and the last name. Program crashed while printing out the address of the first person.

Results of COM and XPCOM Test Modules: Both of the modules worked without any problem.

TEST 3: In this test declaration order of the data members was changed. Location of member variable "address" was changed, so that it would be the first element in the declaration order. After this modification order of the data members was:

```
char* address;
Date birthDate;
char* name;
```

```
char* lastName;
```

Expectation: At the binary level, an instance of Person class will start from memory location m with the data member "address". Since the Student class is not compiled, its constructor knows that the member variable "address" is the last element in the declaration and has an address of $m + 20$. So the initialization of "address" in the Student class's constructor will overwrite the member variable "lastName". The value of "address" will always remain as a null. When the test program tries to print out the last name of the person, his address must be printed out and when it tries to print out the address of the person, the program must crash.

Result of Non-COM Test Module: Name of the first person was printed out correctly but his address appeared to be his last name. Before his address is printed out, the program crashed.

Results of COM and XPCOM Test Modules: Both of the modules worked without any problem.

TEST 4: This time the location of the last name was changed so that it would be the last data member in the class.

Expectation: For the same reasons with test 5, the Student class constructor will overwrite the last name of the person with his address. When getLastName function is called it must return address of the person.

Result of Non-COM Test Module: Name of the first person was printed out correctly but his address appeared to be his last name. Before his address is printed out, the program crashed.

Results of COM and XPCOM Test Modules: Both of the modules worked without any problem.

TEST 5: In this test a new data member, named "age" was added into the class declaration by un-commenting "//#define VERSION2" line in the beginning of the Person class. After this modification the order of the members was:

```
Date birthDate;
char* name;
char* lastName;
long age;
char* address;
```

Without making any signature changes, the implementation of setBirthDate and getAge function was modified so that they made use of this new data member. During construction phase of a person instance, setBirthDate function set the "age" by subtracting birth year of the person from the current year. Instead of recalculating the age of the person, getAge function simply returned this new data member.

Expectation: Since the school name will overwrite the address, getAddress function must return the school name.

Result of Non-COM Test Module: School names of the people appeared to be their addresses. Program crashed while it was processing the fifth person. Ages of the people were not correct.

Results of COM and XPCOM Test Modules: Both of the modules worked without any problem.

TEST 6: This time a new member variable, named "livingIn" was added to the class definition. This variable held the name of the country that the person was currently living in and was initialized to a default value of "USA" in the Person class's constructor.

"getAddress" function was modified so that it concatenated this new variable to the end of the address and returned the resultant string.

Expectation: The location of the data member "address" will be shifted by 4 bytes, since the new member is declared right before it. While the Student class constructor is initializing the address of the student, it will overwrite address of the person with the school name. When getAddress function is called, it must return a string by concatenating address of the person to the end of the school name.

Result of Non-COM Test Module: School name of the students appeared to be a part of their addresses. Program crashed while processing the fifth person.

Results of COM and XPCOM Test Modules: Both of the modules worked without any problem.

TEST 7: In this test one of the data members was removed from the class declaration. For this purpose, first, all the modules were recompiled so that they could see new data member "age" that was added into the class declaration in the previous test. Then "age" was remove from the declaration by commenting "#define VERSION2" line in the beginning of the Person class and then only this module was recompiled.

Expectation: Since the data member "age" is removed from the Person class, its constructor will allocate 4 bytes of memory for the data member "address" starting from the location $m + 20$ and ending at the location $m + 23$. But since the Student class is not recompiled, its constructor will think that memory address of the data member "address" is $m + 24$ and Person class ends at location $m + 27$. It allocates 8 bytes of memory for its data members starting from location $m + 28$. Meanwhile the memory location between $m + 24$ and $m + 27$ will not be used by any of the constructors. This means that a

memory leakage occurs. When the test program tries to print out the people addresses, the program must crash since they have null values.

Result of Non-COM Test Module: After printing out the name and last name of the first person correctly, program crashed when it tried to print out the address of the person.

Results of COM and XPCOM Test Modules: Both of the modules worked without any problem.

TEST 8: This time four new temporary data members of type "double" were added in front of the existing data members and both Person and Student classes were recompiled. Then these new members were removed and only Person class was recompiled.

Expectation: While initializing the address of the student, the Student class constructor will write this value to a memory location that is no more a part of the Person class and the actual address value will always remain as null. When getAddress function is called this must return a null value and crash the program.

Result of Non-COM Test Module: After printing out the name and the last name of the first person correctly, program crashed when it tried to print out the address of the person.

Results of COM and XPCOM Test Modules: Both of the modules worked without any problem.

TEST 9: In this test one of the data members' type was changed. For this purpose all the modules were recompiled so that they could see the new data member "age". Then this data member's type was changed from "long" to "double" and only Person class was recompiled.

Expectation: A "long" number is represented with 4 bytes on the testing platform, whereas a "double" number is represented with 8 bytes. So data member "age" will occupy 4 bytes of more space in the memory. This means that Student class's constructor will use last for bytes of "age" for address of the student and also overwrite the address of the person with a wrong value.

Result of Non-COM Test Module: The curriculums of the people appeared to be their addresses. All the ages were calculated to be 0.

Results of COM and XPCOM Test Modules: Both of the modules worked without any problem.

TEST 10: This time type of the data member "name" was changed from "char*" to "char[20]".

Expectation: Since a character is represented with 1 byte, the data members "lastName" and "address" will be shifted by 20 bytes in the memory. Student class constructor will allocate memory for its data members starting from location $m + 24$ ending at $m + 31$, which is now part of the Person class's memory space. This means that the school name and the curriculum will overwrite some part of person name. When getName function is called, this function should return wrong values for those names that have more then 12 characters. Meanwhile, the actual address value will always remain as a null. When getAddress function is called this must return a null value so crash the program.

Result of Non-COM Test Module: After printing out the name and last name of the first person correctly, program crashed when it tried to print out the address of the person.

Results of COM and XPCOM Test Modules: Both of the modules worked without any problem.

TEST NO:	TYPE OF MODIFICATION	NON-COM TEST MODULE	[XP]COM TEST MODULES
0	No change	OK	OK
1	A new data member is added	Crashed	OK
2	A new data member is added	Crashed	OK
3	The declaration order of data members is changed	Crashed	OK
4	The declaration order of data members is changed	Crashed	OK
5	A new data member is added and functions implementation is changed accordingly	Crashed	OK
6	A new data member is added and functions implementation is changed accordingly	Crashed	OK
7	A data member is removed from the class declaration	Crashed	OK
8	Four data members removed from the class declaration	Crashed	OK
9	Type of a data member is changed	Gave wrong results	OK
10	Type of a data member is changed	Crashed	OK

Table 5.1. Summary of the Tests

C. SUMMARY:

This chapter discussed the results of ten tests with different modifications on the Person class's binary layout in non-COM module, COM module and XPCOM module. Non-COM test module failed in each of these tests by either crashing the program or giving wrong results. On the other hand, COM and XPCOM test modules were not affected from any of these changes and worked without any problem.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. SUMMARY OF THE STUDY:

This study examined the usability of COM and XPCOM in Bamboo modules to overcome the versioning problem. For this purpose, three new modules were created under Bamboo. The first module was an unsafe module that had no countermeasures for the versioning problem. The second and third modules applied COM and XPCOM rules to the unsafe module respectively. Ten different tests, which covered all the possible ways that the versioning problem could affect a Bamboo module conducted on each of these modules. The results of these tests showed that an unsafe module always fails (by either crashing the program or giving wrong results), whereas COM and XPCOM modules are not affected by the versioning problem.

B. CONCLUSION:

The versioning problem is a serious obstacle for Bamboo's extensibility. The results of the tests that were conducted in this study proved that COM and XPCOM could be used in Bamboo to overcome this problem. However, since it is not portable, COM can only be used on Windows platforms. On the other hand, XPCOM has all the features of COM that Bamboo needs, plus portability. This makes XPCOM the most appropriate solution for Bamboo among the other approaches that are currently used for the versioning problem.

C. LESSONS LEARNED:

COM and XPCOM have relatively long learning curves. Understanding how they work requires some high level C++ programming knowledge, such as how virtual functions are treated in C++. Most of the implementation uses pointer operations that can easily confuse the programmers and lead them to error prone code.

There are many references for COM, including examples in the Microsoft Development Network (MSDN) Library. Unfortunately, almost all of these references assume that the reader is a very experienced C++ programmer familiar with the data types and functions specific to the Windows operating system. Most of the examples are far from easy to understand.

XPCOM has not even this much documentation. The only two information sources for XPCOM are the Mozilla Organization web site and relevant news groups. Some of the information on this site is out-dated and conflicts with the current XPCOM API. Available articles on XPCOM usually reference COM resources.

Another difficulty with XPCOM is that there is no (at least to the knowledge of the author) simple way to extract *only* the XPCOM module from the Mozilla source. The only apparent method is to build all of the Mozilla and then extract the necessary binaries for XPCOM. Of course, this is a very time consuming process.

A last point about XPCOM is that it is still under development. Every build comes with many changes. Some features in XPCOM were not operational until a new version was released during testing.

With both COM and XPCOM, once the programmer completes the learning curve implementation of [XP]COM objects becomes easier and easier. The most time consuming part of the code in Appendices (TBD) was implementing the COM Person object. After finishing it, implementing the Student was relatively easy.

D. RECOMMENDATIONS FOR FUTURE WORK:

XPCOM is not yet integrated into Bamboo. This study used a temporary solution to test its use; its details are provided in Appendix A. But this solution is not appropriate for Bamboo. Bamboo already incorporates NSPR, and XPCOM should be similarly included. Currently none of the modules in Bamboo (except for the kernel) uses any countermeasures for the versioning problem. After integrating XPCOM, all C++ classes used in these modules must be re-implemented as XPCOM objects.

The importance of using AddRef and Release functions appropriately is addressed in Chapter III. Even though the references list all situations that require a call to these functions, this is not always easy to follow especially for the large classes. XPCOM provides a very useful tool, named "nsCOMPtr" for this matter and takes the burden from the programmer's shoulder. "nsCOMPtr" is a template class that acts, syntactically, just like an ordinary pointer in C or C++. It is also called as "smart pointer" because, unlike a raw XPCOM interface pointer, nsCOMPtr manages AddRef, Release, and QueryInterface functions for the programmer. While implementing the C++ classes as XPCOM objects in Bamboo modules, it will be beneficial to use this tool.

As was mentioned before, COM or XPCOM objects can not use classic inheritance to reuse other COM or XPCOM objects. The only alternatives are containment and aggregation. Containment is easier to implement than aggregation, but it gives only "has-a" relationship between using and reused objects. Aggregation, on the other hand, gives "is-a" relationship between two objects and is more powerful. While implementing the C++ classes as XPCOM objects in Bamboo modules, it is advisable to implement these classes as aggregatable.

APPENDIX A. XPCOM MODULE USER GUIDE

As was mentioned in Chapter III, this study used a temporary solution to integrate XPCOM with Bamboo models. For this purpose a new module, named as XPCOMDemo was created. This module served as a base module for all the modules that used XPCOM. By depending on XPCOMDemo module, each module was able to use necessary XPCOM header files without specifying their absolute path. This Appendix will explain step by step how to create XPCOMDemo module and use it.

A. CREATING XPCOM MODULE:

1) *Create a new module, named as XPCOMDemo.* This module will serve as a base module for all XPCOM modules. It does not have an implementation and is composed of the following sub-directories: include, src, xpcomdlls, xpcomidls, and xpcomlibs.

2) *Extract the necessary header files for XPCOM.* The following header files are necessary for XPCOM modules and must be copied into the "include" directory of XPCOMDemo:

nsCom.h	nsComponentManager.h	nsComponentManagerUtils.h
nscore.h	nsCppSharedAllocator.h	nsIComponentManager.h
nsDebug.h	nsError.h	nsIAllocator.h
nsCRT.h	nsFileSpec.h	nsStringUtil.h
nsID.h	nsIEnumerator.h	nsIFactory.h
nsIID.h	nsIFileSpec.h	nsIServiceManager.h
nsStr.h	nsISupports.h	nsISupportsUtils.h
nsrootidl.h	nsString.h	nsString2.h
nsIAtom.h	nsTraceRefcnt.h	plstr.h

```
pratom.h      prcmon.h      prcpucfg.h
prinet.h      prinrval.h    prio.h
prlock.h      prlong.h      prmon.h
prtime.h      prtypes.h
```

3) *Extract the necessary dynamic link library and executable files for XPCOM.* The following DLL and executable files are necessary for XPCOM modules and must be copied into the "xpcomdlls" directory of XPCOMDemo module: xpcom.dll, nspr3.dll, plc3.dll, plds3.dll and xpidl.exe. The following DLL and executable files must be copied into the directory that hosts bamboo.exe (For Windows NT BB_DIR\main\bin\winnt): mozreg.dll, regExport.exe and regxpcom.exe.

4) *Extract the necessary IDL files for XPCOM.* The following IDL files are required for XPCOM modules and must be copied into "xpcomidls" directory of XPCOMDemo module: nsISupports.idl, nsrootidl.idl

5) *Extract the necessary library files for XPCOM.* The following library files are necessary for XPCOM modules and must be copied into "xpcomlibs" directory of XPCOMDemo: nspr3.lib, xpcom.lib.

6) *Adjust "PATH" environment variable.* Change "PATH" environment variable so that it includes the directory that hosts "bamboo.exe" and "xpcomdlls" directory.

B. USING XPCOMDEMO MODULE:

1) *Create and implement your module as described in Chapter IV by specifying a dependency on XPCOMDemo module.* Compile your IDL file by using xpidl.exe with "–m" and "–I" switches. "-m" switch specifies the compilation mode and

"header" option must be used. "-I" switch specifies the absolute path to the included IDL files. Since every IDL file will include "nsISupports.idl" this switch must be used at least with the path to the "xpcomidls" directory of XPCOMDemo. Example:

```
C:\Bamboo\bbKernelModule\MyXPCOMModule\include> xpidl -m
header -I C:\Bamboo\bbKernelModule\XPCOMDemo\xpcomidls
IPerson.idl
```

2) *Compile your module.*

3) *Register your module.* Use "regxpcom.exe" with the path to the module's DLL and the DLL name to register your module. Example:

```
C:\Bamboo\bbKernelModule\MyXPCOMModule> regxpcom lib\winnt\
MyXPCOMModule.dll
```

"regxpcom.exe" creates a new file, named as "component.reg" under the directory that hosts "bamboo.exe" and stores the necessary information to create instances of XPCOM object. Contents of the registry can be seen by using "regexport.exe" within this directory.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. NONCOM MODULE SOURCE CODE

```
// ************************************************************************
// EXECUTIVE SUMMARY
// File Name: Person.h
// Description: Decleration of Person class
// Author(s): Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file defines Person class.
// ************************************************************************

#ifndef _PERSON_H_
#define _PERSON_H_

#include "bbGlobals.h"
#include <string.h>
#include <iostream.h>
#include <string.h>

typedef struct _Date {
      long day;
      char* month;
      long year;
      _Date() {
            day = 0;
            year = 0;
            month = 0;
      }
      Date(char* m, long d, long y) {
            day = d;
            year = y;
            month = new char[strlen(m) + 1];
            strcpy(month, m);
      }
}Date;

//#define VERSION2
class BB_Proper_Export_Flag Person {
private :
      Date birthDate;
      char* name;
      char* lastName;
#ifdef VERSION2
      long age;
#endif
protected :
      char* address;
public:
      Person();
      Person(Date* p_birthDate, char* p_name,
            char* p_lastName, char* p_address = 0);
      ~Person();

      char* getName();
      char* getLastName();
      char* getAddress();
      Date* getBirthDate();
      long getAge();
```

```
      void setName(char* p_name);
      void setLastName(char* p_lastName);
      void setAddress(char* p_address);
      void setBirthDate(Date* p_birthDate);
      void setPersonInfo(char* p_name, char* p_lastName,
                         char* p_address, char* monthOfBirth,
                         long dayOfBirth, long yearOfBirth);
};
#endif


// ************************************************************************
//   EXECUTIVE SUMMARY
// File Name: Person.c
// Description: Implementation of Person class
// Author(s): Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file implements Person class.
// ************************************************************************


#include "Person.h"
#include <iostream.h>
#include <string.h>
#include <time.h>

long getThisYear();

Person::Person(){
}

Person::Person(Date* p_birthDate, char* p_name,
               char* p_lastName, char* p_address) {
      setBirthDate(p_birthDate);
      setName(p_name);
      setLastName(p_lastName);
      setAddress(p_address);
}

Person::~Person() {
      if(name) delete [] name;
      if(lastName) delete [] lastName;
      if(address) delete [] address;
}

char* Person::getName() {
      char* n = new char[strlen(name) + 1];
      strcpy(n, name);
      return n;
}

char* Person::getLastName() {
      char* ln = new char[strlen(lastName) + 1];
      strcpy(ln, lastName);
      return ln;
}

char* Person::getAddress() {
```

```
        char* addr = new char[strlen(address) + 1];
        strcpy(addr, address);
        return addr;
}

Date* Person::getBirthDate() {
        Date* birth = new Date;
        birth->day = birthDate.day;
        birth->year = birthDate.year;
        birth->month = new char[strlen(birthDate.month) + 1];
        strcpy(birth->month, birthDate.month);
        return birth;
}

void Person::setName(char* p_name) {
        name = new char[strlen(p_name) + 1];
        strcpy(name, p_name);
}

void Person::setLastName(char* lName) {
        lastName = new char[strlen(lName) + 1];
        strcpy(lastName, lName);
}

void Person::setAddress(char* addr) {
        if(addr) {
                address = new char[strlen((char*)addr) + 1];
                strcpy(address, addr);
        }
        else
                address = 0;
}

void Person::setBirthDate(Date* birth) {
        birthDate.day = birth->day;
        birthDate.year = birth->year;
        birthDate.month = new char[strlen(birth->month) + 1];
        strcpy(birthDate.month, birth->month);
#ifdef VERSION2
        age = getThisYear() - birth->year;
#endif
}

void Person::setPersonInfo(char* p_name, char* lName,
                           char* addr, char* monthOfBirth,
                           long dayOfBirth, long yearOfBirth) {
        setName(p_name);
        setLastName(lName);
        setAddress(addr);

        Date* d = new Date;
        d->month = new char[strlen(monthOfBirth) + 1];
        strcpy(d->month, monthOfBirth);
        d->day = dayOfBirth;
        d->year = yearOfBirth;
        setBirthDate(d);
}
```

```
long Person::getAge() {
#ifdef VERSION2
      return age;
#else
      return 2000 - birthDate.year;
#endif;
}

long getThisYear() {
      time_t t;
      time(&t);
      struct tm* t2 = localtime(&t);
      return (long)(t2->tm_year + 1900);
}
//End File Person.c


// ************************************************************************
// EXECUTIVE SUMMARY
// File Name: personModule.h
// Description: Declaration for the module's global functions
// Author: Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file declares 2 global C functions ; the
//         init and exit module funcs.
// ************************************************************************


#ifndef _PERSONMODULE_H
#define _PERSONMODULE_H

#include "bbGlobals.h"


extern "C"
  {
  BB_Proper_Export_Flag bool initModule();
  BB_Proper_Export_Flag bool exitModule();
  };
#endif
//End File personModule.h


// ************************************************************************
//   EXECUTIVE SUMMARY
// Module Name: personModule.c
// Description: Definition of personModule.c
// Author(s): Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file defines the global functions
// ************************************************************************

#include "personModule.h"
#include "person.h"
#include "bbSystem.h"
#include <iostream.h>

bool initModule()  {
      cout << "Person Module Started.  Using : ";
```

```
#ifdef VERSION2
      cout << "VERSION 2"  << endl;
#else
      cout << "VERSION 1"  << endl;
#endif
  return 1;
}// End File personModule.c

// ************************************************************************
// EXECUTIVE SUMMARY
// File Name: Student.h
// Description: Declaration of Student class
// Author: Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file declares Student class
// ************************************************************************

#ifndef _STUDENT_H_
#define _STUDENT_H_

#include "Person.h"

class BB_Proper_Export_Flag Student : public Person {
      char* schoolName;
      char* curric;
public:
      Student(char* name, char* lName, char* addr,
              Date* birth, char* school, char* curric);
      ~Student();
      char* getSchoolName();
      char* getCurric();
      void setSchoolName(char* school);
      void setCurric(char* curric);
      void setStudentInfo(char* school, char* curric);
};
#endif  //End File Student.h

// ************************************************************************
//   EXECUTIVE SUMMARY
// File Name: Student.c
// Description: Implementation of Student class
// Author: Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file implements Student class
// ************************************************************************

#include "Student.h"
#include <iostream.h>
#include <string.h>

Student::Student(char* p_name, char* lName, char* addr, Date* birth,
                 char* sch, char* c) : Person(birth, p_name, lName)  {
      address = new char[strlen((char*)addr) + 1];
      strcpy(address, addr);
      setSchoolName(sch);
      setCurric(c);
}

Student::~Student() {
```

```
        if(schoolName) delete [] schoolName;
        if(curric) delete [] curric;
}

char* Student::getSchoolName() {
        if(!schoolName) {
                return "NOT SPECIFIED";
        }
        char* s = new char[strlen(schoolName) + 1];
        strcpy(s, schoolName);
        return s;
}

char* Student::getCurric() {
        char* c = new char[strlen(curric) + 1];
        strcpy(c, curric);
        return c;
}

void Student::setSchoolName(char* p_schoolName) {
        if(!strcmp(p_schoolName, " ")) {
                schoolName = NULL;
        }
        else {
                schoolName = new char[strlen(p_schoolName) + 1];
                strcpy(schoolName, p_schoolName);
        }
}

void Student::setCurric(char* p_curric) {
        curric = new char[strlen(p_curric) + 1];
        strcpy(curric, p_curric);
}

void Student::setStudentInfo(char* p_schoolName, char* p_curric) {
        setSchoolName(p_schoolName);
        setCurric(p_curric);
}

//End File Student.c

// ************************************************************************
//   EXECUTIVE SUMMARY
// File Name: studentModule.h
// Description: Declaration for the module's global functions
// Author: Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file declares 2 global C functions ; the
//         init and exit module functions.
// ************************************************************************

#ifndef _STUDENTMODULE_H_
#define _STUDENTMODULE_H_

#include "bbGlobals.h"


extern "C"    {
```

```
  BB_Proper_Export_Flag bool initModule();
  BB_Proper_Export_Flag bool exitModule();
};
#endif


// ************************************************************************
//   EXECUTIVE SUMMARY
// File Name: studentModule.c
// Description: Definition of StudentModule
// Author(s): Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file defines the global functions
// ************************************************************************

#include "studentModule.h"
#include "bbSystem.h"


bool initModule()  {
      cout << "Student Module Started. Using : ";
#ifdef VERSION2
      cout << "VERSION 2\n\n"  << endl;
#else
      cout << "VERSION 1\n\n"  << endl;
#endif
  return 1;
}

bool exitModule()  {
  return 1;
}
//End File StudentModule.c

// ************************************************************************
//   EXECUTIVE SUMMARY
// Module Name: module.c
// Description: Definition of _module.c
// Author(s): Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file tests the Person and Student classes
// ************************************************************************

#include <iostream.h>
#include "bbSystem.h"
#include "module.h"
#include <string.h>
#include "PeopleInfo.h"
#include "Student.h"

long convert(char* str);

BB_Proper_Export_Flag bool initModule()
  {
      cout << "        ---------- Welcome to Student Test ----------"
           << endl;
      Date *d = new Date;
      unsigned char temp[50];
      const int entries = sizeof(peopleTable) / sizeof(*peopleTable);
```

```
        Student *students[entries];
        for(int ix = 0; ix < entries; ix++) {
              d = new Date(peopleTable[ix][3], convert(peopleTable[ix][4]),
                           convert(peopleTable[ix][5]));

              students[ix] = new Student(peopleTable[ix][0],
                                         peopleTable[ix][1],
                                         peopleTable[ix][2], d,
                                         studentTable[ix][0],
                                         studentTable[ix][1]);
        }
        cout << "Studen Info    : " << endl;
        cout << "---------------\n" << endl;
        for(ix = 0; ix < entries; ix++) {
              cout << "Name         : " << students[ix]->getName() << " ";
              cout << students[ix]->getLastName() << endl;
              cout << "Address      : " << students[ix]->getAddress() << endl;
              d = students[ix]->getBirthDate();
              cout << "Birth Date   : " << d->month << " "
                   << d->day << ", " << d->year << endl;
              cout << "School       : " << students[ix]->getSchoolName() << endl;
              cout << "Curric       : " << students[ix]->getCurric() <<  endl;
              cout << "Age          : " << students[ix]->getAge() << "\n" << endl;
        }
        cout <<  endl;
        bbSystem::shutdown();
        return 1;
    }

BB_Proper_Export_Flag bool exitModule() {
    return 1;
}

long convert(char* str) {
        long value = 0;
        long power = 1;
        int digits = strlen(str);
        for(int ix = 0; ix < digits; ix++) {
              long m = digits - (ix + 1);
              for(int iy = 0; iy < m; iy++) {
                    power *= 10;
              }
              value += (str[ix]-48)*power;
              power = 1;
        }
        return value;
}
//End File Module.c
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. COM MODULE SOURCE CODE

```
// ************************************************************************
// EXECUTIVE SUMMARY
// Module Name: IPerson.idl
// Description: Declaration for IPerson Interface
// Author: Mithat Daglar daglar@cs.nps.navy.mil
// Informal: This file defines IPerson interface in Microsoft
//           Interface Definition Language (IDL).  This file must
//           be compiled by using midl.exe compiler to get
//           IPerson.h and IPerson_i.c
// ************************************************************************

#ifndef _IPERSON_IDL_
#define _IPERSON_IDL_

import "oaidl.idl";
import "ocidl.idl";

typedef struct _Date {
      long day;
      char* month;
      long year;
}Date;

[
      local, object, uuid(23B22051-73DB-11d3-84DC-00105A0DEB6F),
]
interface IPerson : IUnknown {
      HRESULT getName([out, retval] char* name);
      HRESULT getLastName([out, retval] char* lName);
      HRESULT getAddress([out, retval] char* addr);
      HRESULT getBirthDate([out, retval] Date* birth);
      HRESULT getAge([out, retval] long* age);
      HRESULT setName([in] char* name);
      HRESULT setLastName([in] char* lName);
      HRESULT setAddress([in] char* addr);
      HRESULT setBirthDate([in] Date* birth);
      HRESULT setPersonInfo([in] char* name, [in] char* lName,
                            [in] char* addr, [in] char* monthOfBirth,
                            [in] long dayOfBirth, [in] long yearOfBirth);
};
[
      uuid(23B22052-73DB-11d3-84DC-00105A0DEB6F)
]
library PersonLib
{
      importlib("stdole32.tlb");
      importlib("stdole2.tlb");
      [
            uuid(23B22053-73DB-11d3-84DC-00105A0DEB6F)
      ]
      coclass PersonObject {
            [default] interface IPerson;
      };
};
#endif
//End File IPerson.idl
```

```
// ************************************************************************
//   EXECUTIVE SUMMARY
// File Name: Person.h
// Description: Definition of Person class
// Author(s): Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file defines Person as a "Aggregatable" COM object
// ************************************************************************

#ifndef _PERSON_H_
#define _PERSON_H_

#include "IPerson.h"
#include <stddef.h>

//#define VERSION2
class Person : public IPerson {
private :
      Date birthDate;
      unsigned char* name;
      unsigned char* lastName;
#ifdef VERSION2
      long age;
#endif
      long refCount;
      IUnknown *m_pUnkOuter;
protected :
      unsigned char* address;
public:
      Person(IUnknown *pUnkOuter);
      ~Person();

      //NON-DELEGATING IUNKNOWN METHODS
      STDMETHODIMP InternalQueryInterface(REFIID, void**);
      STDMETHODIMP_(ULONG) InternalAddRef(void);
      STDMETHODIMP_(ULONG) InternalRelease(void);

      //DELEGATING IUNKNOWN METHODS
      STDMETHODIMP QueryInterface(REFIID, void**);
      STDMETHODIMP_(ULONG) AddRef(void);
      STDMETHODIMP_(ULONG) Release(void);

      //IPERSON METHODS
      STDMETHODIMP getName(unsigned char* name);
      STDMETHODIMP getLastName(unsigned char* lName);
      STDMETHODIMP getAddress(unsigned char* addr);
      STDMETHODIMP getBirthDate(Date* birth);
      STDMETHODIMP getAge(long*);
      STDMETHODIMP setName(unsigned char* name);
      STDMETHODIMP setLastName(unsigned char* lName);
      STDMETHODIMP setAddress(unsigned char* addr);
      STDMETHODIMP setBirthDate(Date* birth);
      STDMETHODIMP setPersonInfo(unsigned char* name,
                                 unsigned char* lName, unsigned char* addr,
                                 unsigned char* monthOfBirth,
                                 long dayOfBirth, long yearOfBirth);
```

```
        class XNDUnknown : public IUnknown {
                Person* This() {
                        return (Person*)((BYTE*)this -
                                        offsetof(Person, m_innerUnknown));
                }
                STDMETHODIMP QueryInterface(REFIID r, void** p) {
                        return This()->InternalQueryInterface(r, p);
                }
                STDMETHODIMP_(ULONG) AddRef(void) {
                        return This()->InternalAddRef();
                }
                STDMETHODIMP_(ULONG) Release(void) {
                        return This()->InternalRelease();
                }
        };
        XNDUnknown m_innerUnknown;
};
#endif
//End File Person.h

// ********************************************************************
//   EXECUTIVE SUMMARY
// File Name: Person.c
// Description: Implements Person class
// Author(s): Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file implements Person class as a COM object
// ********************************************************************

#include "Person.h"
#include "IPerson_i.c"
#include <iostream.h>
#include <string.h>
#include <time.h>

long getThisYear();

Person::Person(IUnknown *pUnkOuter) : refCount(0) {
        if(pUnkOuter)
                m_pUnkOuter = pUnkOuter;
        else
                m_pUnkOuter = &m_innerUnknown;
}

Person::~Person() {
        if(name) delete [] name;
        if(lastName) delete [] lastName;
        if(address) delete [] address;
        if(birthDate.month) delete [] birthDate.month;
}

//IUNKNOWN Functions
STDMETHODIMP Person::InternalQueryInterface(REFIID iid, void** ppv) {
        if(iid == IID_IUnknown) {
                *ppv = static_cast<IUnknown*> (&m_innerUnknown);
        }
        else if(iid == IID_IPerson) {
```

```
            *ppv = static_cast<IPerson*> (this);
      }
      else {
            *ppv = NULL;
            return E_NOINTERFACE;

      }

      ((IUnknown*)*ppv)->AddRef();
      return S_OK;
}
STDMETHODIMP_(ULONG) Person::InternalAddRef(void) {
      return ++refCount;
}

STDMETHODIMP_(ULONG) Person::InternalRelease(void) {
      long rem = --refCount;
      if(rem == 0) {
            delete this;
      }

      return rem;
}

STDMETHODIMP Person::QueryInterface(REFIID iid, void** ppv) {
      return m_pUnkOuter->QueryInterface(iid, ppv);
}

STDMETHODIMP_(ULONG) Person::AddRef(void) {
      return m_pUnkOuter->AddRef();
}

STDMETHODIMP_(ULONG) Person::Release(void) {
      return m_pUnkOuter->Release();
}

//IPERSON Functions
STDMETHODIMP Person::getName(unsigned char* n) {
      strcpy((char*)n, (char*)name);
      return S_OK;
}

STDMETHODIMP Person::getLastName(unsigned char* ln) {
      strcpy((char*)ln, (char*)lastName);
      return S_OK;
}

STDMETHODIMP Person::getAddress(unsigned char* addr) {
      strcpy((char*)addr, (char*)address);
      return S_OK;
}

STDMETHODIMP Person::getBirthDate(Date* birth) {
      birth->day = birthDate.day;
      birth->year = birthDate.year;
      birth->month = new unsigned char[strlen((char*)birthDate.month) + 1];
      strcpy((char*)(birth->month), (char*)birthDate.month);
```

```
	return S_OK;
}

STDMETHODIMP Person::setName(unsigned char* n) {
	if(name) delete [] name;
	name = new unsigned char[strlen((char*)n) + 1];
	strcpy((char*)name, (char*)n);
	return S_OK;
}

STDMETHODIMP Person::setLastName(unsigned char* lName) {
	if(lastName) delete [] lastName;
	lastName = new unsigned char[strlen((char*)lName) + 1];
	strcpy((char*)lastName, (char*)lName);
	return S_OK;
}

STDMETHODIMP Person::setAddress(unsigned char* addr) {
	if(address) delete [] address;
	address = new unsigned char[strlen((char*)addr) + 1];
	strcpy((char*)address, (char*)addr);
	return S_OK;
}

STDMETHODIMP Person::setBirthDate(Date* birth) {
	birthDate.day = birth->day;
	birthDate.year = birth->year;
	birthDate.month = new unsigned char[strlen((char*)(birth->month)) + 1];
	strcpy((char*)birthDate.month, (char*)birth->month);
#ifdef VERSION2
	age = getThisYear() - birth->year;
#endif

	return S_OK;
}

STDMETHODIMP Person::setPersonInfo(unsigned char* n, unsigned char* lName,
                                   unsigned char* addr,
                                   unsigned char* monthOfBirth,
                                   long dayOfBirth, long yearOfBirth)
{
	setName(n);
	setLastName(lName);
	setAddress(addr);
	Date* d = new Date;
	d->month = new unsigned char[strlen((char*)monthOfBirth) + 1];
	strcpy((char*)(d->month), (char*) monthOfBirth);
	d->day = dayOfBirth;
	d->year = yearOfBirth;
	setBirthDate(d);
	return S_OK;
}

STDMETHODIMP Person::getAge(long* a) {
#ifdef VERSION2
	*a = age;
#else
```

```
	*a = 2000 - birthDate.year;
#endif;
	return S_OK;
}

long getThisYear() {
	time_t t;
	time(&t);
	struct tm* t2 = localtime(&t);
	return (long)(t2->tm_year + 1900);
}//End File Person.c

// ************************************************************************
//   EXECUTIVE SUMMARY
// File Name: PersonFactory.h
// Description: Defines PersonFactory class
// Author(s): Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file defines PersonFactory class which is used to
//                create Person COM objects
// ************************************************************************

#ifndef _PERSONFACTORY_H_
#define _PERSONFACTORY_H_

#include "IPerson.h"
#include "Person.h"

class PersonFactory : public IClassFactory {
	ULONG refCount;
public:
	PersonFactory();
	~PersonFactory();
	//IUNKNOWN Functions
	STDMETHODIMP QueryInterface(REFIID, void**);
	STDMETHODIMP_(ULONG) AddRef(void);
	STDMETHODIMP_(ULONG) Release(void);

	//ICLASSFACTORY Functions
	STDMETHODIMP CreateInstance(IUnknown*, REFIID, void**);
	STDMETHODIMP LockServer(BOOL);
};
STDAPI DllGetClassObject(REFCLSID, REFIID, void**);
STDAPI DllRegisterServer();
STDAPI DllUnregisterServer();
STDAPI DllCanUnloadNow();
#endif
//End File PersonFactory.h

// ************************************************************************
//   EXECUTIVE SUMMARY
// File Name: PersonFactory.c
// Description: Implements PersonFactory class
// Author(s): Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file implements PersonFactory class which is used to
//                create Person COM objects
// ************************************************************************
```

```
#include "PersonFactory.h"
#include <iostream.h>

PersonFactory::PersonFactory() : refCount(0) {
}

PersonFactory::~PersonFactory() {}

STDMETHODIMP PersonFactory::QueryInterface(REFIID iid, void **ppv ) {
      *ppv = NULL;
      if(iid == IID_IUnknown || iid == IID_IClassFactory) {
            *ppv = (IPerson*) this;
            AddRef();
            return S_OK;
      }
      else {
            *ppv = NULL;
            return E_NOINTERFACE;
      }
}

STDMETHODIMP_(ULONG) PersonFactory::AddRef(void) {
      return ++refCount;
}

STDMETHODIMP_(ULONG) PersonFactory::Release(void) {
      long rem = --refCount;
      if(rem == 0) {
            delete this;
      }
      return rem;
}

//ICLASSFACTORY Functions
STDMETHODIMP PersonFactory::CreateInstance(IUnknown* outer,
                                           REFIID iid, void **ppv) {
      if(outer != 0 && iid != IID_IUnknown) {
            *ppv = 0;
            return E_INVALIDARG;
      }
      Person *p = new Person(outer);
      if(p == 0) {
            *ppv = 0;
            return E_OUTOFMEMORY;
      }
      p->InternalAddRef();
      HRESULT hr = p->InternalQueryInterface(iid, ppv);
      p->InternalRelease();
      return hr;
}

STDMETHODIMP PersonFactory::LockServer(BOOL l)  {
      return S_OK;

}

STDAPI DllGetClassObject(REFCLSID clsid, REFIID iid, void **ppv) {
```

```
        if(clsid != CLSID_PersonObject) {
                return CLASS_E_CLASSNOTAVAILABLE;
        }

        static PersonFactory personFactoryObject;
        HRESULT hr = personFactoryObject.QueryInterface(iid, ppv);

        if(FAILED(hr)) {
                *ppv = NULL;
        }
        return hr;
}


const char *regTable[][3] = {
        {"CLSID\\{23B22053-73DB-11d3-84DC-00105A0DEB6F}", 0, "PersonObject" },
        {"CLSID\\{23B22053-73DB-11d3-84DC-00105A0DEB6F}\\InprocServer32", 0,
         (const char*)-1}
};

STDAPI DllRegisterServer() {
        HRESULT hr = S_OK;
        int entries = sizeof(regTable) / sizeof(*regTable);
        char* file= new char[255];
        HMODULE module = GetModuleHandle("COMPersonClassModule.dll");

        GetModuleFileName(module, file, 255);

        for(int ix = 0; SUCCEEDED(hr) && ix < entries; ix++) {
                const char *keyName = regTable[ix][0];
                const char *valueName = regTable[ix][1];
                const char *value = regTable[ix][2];
                if(value == (const char*) - 1) value = file;


                HKEY hkey;
                long err = RegCreateKeyA(HKEY_CLASSES_ROOT, keyName, &hkey);

                if(err != 0) {
                        err = RegSetValueExA(hkey, valueName, 0, REG_SZ,
                                        (const BYTE*) value, (strlen(value) +1));

                        RegCloseKey(hkey);
                }
                if( err == 0 ) {
                        DllUnregisterServer();
                        hr = S_FALSE;
                }
        }
        return hr;
}

STDAPI DllUnregisterServer() {
        HRESULT hr = S_OK;
```

```
	int entries = sizeof(regTable) / sizeof(*regTable);

	for(int ix = entries - 1; ix >= 0; ix--) {
		const char* keyName = regTable[ix][0];

		long err = RegDeleteKeyA(HKEY_CLASSES_ROOT, keyName);

		if(err == 0) {
			hr = S_FALSE;
		}
	}
	return hr;
}
STDAPI DllCanUnloadNow() {
	return S_FALSE;
}
//End File PersonFactory.c


// ********************************************************************
// EXECUTIVE SUMMARY
// File Name: IStudent.idl
// Description: Declaration for IStudent Interface
// Author: Mithat Daglar daglar@cs.nps.navy.mil
// Informal: This file defines IStudent interface in Microsoft
//           Interface Definition Language (IDL).  This file must
//           be compiled by using midl compiler to get IStudent.h and
//           IStudent_i.c
// ********************************************************************

#ifndef _ISTUDENT_IDL_
#define _ISTUDENT_IDL_

import "oaidl.idl";
import "ocidl.idl";

[
	local, object, uuid(A4882DC1-6B9A-11d3-9B26-0800690F2271)
]
interface IStudent : IUnknown {
	HRESULT getSchoolName([out, retval] char* school);
	HRESULT getCurric([out, retval] char* curric);
	HRESULT setSchoolName([in] char* school);
	HRESULT setCurric([in] char* curric);
	HRESULT setStudentInfo([in] char* school, [in] char* curric);
};

[
	uuid(A4882DC2-6B9A-11d3-9B26-0800690F2271)
]
library StudentLib
{
	importlib("stdole32.tlb");
	importlib("stdole2.tlb");

	[
		uuid(A4882DC3-6B9A-11d3-9B26-0800690F2271)
```

```
        ]
        coclass StudentObject {
                [default] interface IStudent;
        };
};
#endif
//End File Student.idl


// ************************************************************************
//   EXECUTIVE SUMMARY
// File Name: Student.h
// Description: Declaration of Student class
// Author: Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file defines Student class as a COM object that aggragates
//           Person class but not aggragatable.
// ************************************************************************


#ifndef _STUDENT_H_
#define _STUDENT_H_

#include "IStudent.h"
#include "IPerson.h"

class Student : public IStudent {
        long refCount;
        unsigned char* schoolName;
        unsigned char* curric;

        IPerson *m_Person;
public:
        Student();
        ~Student();
        //IUNKNOWN Functions
        STDMETHODIMP QueryInterface(REFIID, void**);
        STDMETHODIMP_(ULONG) AddRef(void);
        STDMETHODIMP_(ULONG) Release(void);

        //ISTUDENT Functions
        STDMETHODIMP getSchoolName(unsigned char* school);
        STDMETHODIMP getCurric(unsigned char* curric);
        STDMETHODIMP setSchoolName(unsigned char* school);
        STDMETHODIMP setCurric(unsigned char* curric);
        STDMETHODIMP setStudentInfo(unsigned char* school, unsigned char* curric);
};
#endif
//End Student.h

// ************************************************************************
//   EXECUTIVE SUMMARY
// File Name: Student.c
// Description: Implementation of Student class
// Author: Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file implements Student class as a COM object that
//                 aggragates Person class but not aggragatable.
// ************************************************************************
```

```
#include "Student.h"
#include "IStudent.h"
#include "IStudent_i.c"
#include "IPerson_i.c"
#include <iostream.h>
#include <string.h>
#include <assert.h>

Student::Student() : refCount(0) {
      ++refCount;
      HRESULT hr = CoCreateInstance(CLSID_PersonObject, this, CLSCTX_ALL,
                                    IID_IUnknown, (void**)&m_Person);
      assert(SUCCEEDED(hr));
      --refCount;
}

Student::~Student() {
      if(schoolName) delete [] schoolName;
      if(curric) delete [] curric;
      delete this;
}

//IUNKNOWN Functions
STDMETHODIMP Student::QueryInterface(REFIID iid, void** ppv) {
      if(iid == IID_IUnknown || iid == IID_IStudent) {
            *ppv = (IStudent*) this;
      }
      else if(iid == IID_IPerson) {
            return m_Person->QueryInterface(iid, ppv);
      }
      else {
            *ppv = NULL;
            return E_NOINTERFACE;
      }
      if(*ppv) {
            ((IUnknown*)*ppv)->AddRef();
            return NO_ERROR;
      }
      return E_NOINTERFACE;
}

STDMETHODIMP_(ULONG) Student::AddRef(void) {
      return ++refCount;
}

STDMETHODIMP_(ULONG) Student::Release(void) {
      long rem = --refCount;
      if(rem == 0) {
            delete this;
      }
      return rem;
}

//IStudent Fanctions
STDMETHODIMP Student::getSchoolName(unsigned char* s) {
      if(!schoolName) {
```

```
            strcpy((char*)s, (char*)"NOT-SPECIFIED");
      }
      else {
            strcpy((char*)s, (char*)schoolName);
      }
      return S_OK;
}
STDMETHODIMP Student::getCurric(unsigned char* c) {
      strcpy((char*)c, (char*)curric);
      return S_OK;
}


STDMETHODIMP Student::setSchoolName(unsigned char* s) {
      if(!strcmp((char*)s, " ")) {
            schoolName = NULL;
      }
      else {
            schoolName = new unsigned char[strlen((char*)s) + 1];
            strcpy((char*)schoolName, (char*)s);
      }
      return S_OK;
}

STDMETHODIMP Student::setCurric(unsigned char* c) {
      curric = new unsigned char[strlen((char*)c) + 1];
      strcpy((char*)curric, (char*)c);
      return S_OK;
}
STDMETHODIMP Student::setStudentInfo(unsigned char* sch, unsigned char* c) {
      setSchoolName(sch);
      setCurric(c);
      return S_OK;
}
//End File Student.c

// ************************************************************************
//   EXECUTIVE SUMMARY
// File Name: StudentFactory.h
// Description: Declaration of StudentFactory class
// Author: Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file defines StudentFactory class which is used
//                to create instances of Student COM object
// ************************************************************************

#ifndef _STUDENTFACTORY_H_
#define _STUDENTFACTORY_H_

#include "IStudent.h"
#include "Student.h"

class StudentFactory : public IClassFactory {
      ULONG refCount;

public:
      StudentFactory();
      ~StudentFactory();
```

```
        //IUNKNOWN Functions
        STDMETHODIMP QueryInterface(REFIID, void**);
        STDMETHODIMP_(ULONG) AddRef(void);
        STDMETHODIMP_(ULONG) Release(void);

        //ICLASSFACTORY Functions
        STDMETHODIMP CreateInstance(IUnknown*, REFIID, void**);
        STDMETHODIMP LockServer(BOOL);
};
STDAPI DllGetClassObject(REFCLSID, REFIID, void**);
STDAPI DllRegisterServer();
STDAPI DllUnregisterServer();
STDAPI DllCanUnloadNow();
#endif
//End File StudentFactory.h



// ************************************************************************
//   EXECUTIVE SUMMARY
// File Name: StudentFactory.c
// Description: Implementation of StudentFactory class
// Author: Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file implements StudentFactory class which is used
//                  to create instances of Student COM object
// ************************************************************************

#include "StudentFactory.h"
#include <iostream.h>

StudentFactory::StudentFactory() : refCount(0) {
}

StudentFactory::~StudentFactory() {}

STDMETHODIMP StudentFactory::QueryInterface(REFIID iid, void **ppv ) {
        *ppv = NULL;
        if(iid == IID_IUnknown || iid == IID_IClassFactory) {
                *ppv = (IStudent*) this;
                AddRef();
                return S_OK;
        }
        else {
                *ppv = NULL;
                return E_NOINTERFACE;
        }
}

STDMETHODIMP_(ULONG) StudentFactory::AddRef(void) {
        return ++refCount;
}

STDMETHODIMP_(ULONG) StudentFactory::Release(void) {
        long rem = --refCount;
        if(rem == 0) {
                delete this;
        }
```

```
        return rem;
}

//ICLASSFACTORY Functions
STDMETHODIMP StudentFactory::CreateInstance(IUnknown* outer,
                                            REFIID iid, void **ppv) {
        if(outer != 0) return CLASS_E_NOAGGREGATION;
        Student *p = new Student();
        if(p == 0) {
                return E_OUTOFMEMORY;
        }
        p->AddRef();
        HRESULT hr = p->QueryInterface(iid, ppv);
        if(FAILED(hr)) {
                delete p;
                *ppv = NULL;
                return E_NOINTERFACE;
        }
        p->Release();
        return NO_ERROR;
}

STDMETHODIMP StudentFactory::LockServer(BOOL l)  {
        return S_OK;
}


//GLOBAL FUNCTIONS

STDAPI DllGetClassObject(REFCLSID clsid, REFIID iid, void **ppv) {
        static StudentFactory studentFactoryObject;
        if(clsid != CLSID_StudentObject) {
                return CLASS_E_CLASSNOTAVAILABLE;
        }
        HRESULT hr = studentFactoryObject.QueryInterface(iid, ppv);
        if(FAILED(hr)) {
                *ppv = NULL;
        }
        return S_OK;
}

const char *regTable[][3] = {
        {"CLSID\\{A4882DC3-6B9A-11d3-9B26-0800690F2271}", 0, "StudentObject" },
        {"CLSID\\{A4882DC3-6B9A-11d3-9B26-0800690F2271}\\InprocServer32", 0,
       (const char*)-1}
};

STDAPI DllRegisterServer() {
        HRESULT hr = S_OK;
        int entries = sizeof(regTable) / sizeof(*regTable);
        char* file= new char[255];
        HMODULE module = GetModuleHandle("COMStudentClassModule.dll");
        GetModuleFileName(module, file, 255);
        for(int ix = 0; SUCCEEDED(hr) && ix < entries; ix++) {
                const char *keyName = regTable[ix][0];
                const char *valueName = regTable[ix][1];
                const char *value = regTable[ix][2];
```

```
            if(value == (const char*) - 1) value = file;
            HKEY hkey;
            long err = RegCreateKeyA(HKEY_CLASSES_ROOT, keyName, &hkey);
            if(err != 0) {
                  err = RegSetValueExA(hkey, valueName, 0, REG_SZ,
                                       (const BYTE*) value, (strlen(value) +1));

                  RegCloseKey(hkey);
            }
            if( err == 0 ) {
                  DllUnregisterServer();
                  hr = S_FALSE;
            }
      }
      return hr;
}

STDAPI DllUnregisterServer() {
      HRESULT hr = S_OK;
      int entries = sizeof(regTable) / sizeof(*regTable);
      for(int ix = entries - 1; ix >= 0; ix--) {
            const char* keyName = regTable[ix][0];
            long err = RegDeleteKeyA(HKEY_CLASSES_ROOT, keyName);
            if(err == 0) {
                  hr = S_FALSE;
            }
      }
      return hr;
}
STDAPI DllCanUnloadNow() {
      return S_FALSE;
}
//End File StudentFactory.c


// ********************************************************************
//   EXECUTIVE SUMMARY
// Module Name: module.c
// Description: Definition of _module.c
// Author(s): Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file tests the Person and Student COM objects
// ********************************************************************

#include "IStudent.h"
#include "IStudent_i.c"
#include "IPerson.h"
#include "IPerson_i.c"
#include <iostream.h>
#include "bbSystem.h"
#include "module.h"
#include <string.h>
#include "PeopleInfo.h"

long convert( unsigned char* str);

BB_Proper_Export_Flag bool initModule()  {
      cout << " --- Welcome to Student Test ---\n" << endl;
```

```
        HRESULT hr = CoInitialize(NULL);
        assert(SUCCEEDED(hr));
        unsigned char temp[50];
        const int entries = sizeof(peopleTable) / sizeof(*peopleTable);
        IPerson *ipp[entries];
        IStudent *isp[entries];
        for(int ix = 0; ix < entries; ix++) {
              hr = CoCreateInstance(CLSID_StudentObject, NULL, CLSCTX_ALL,
                                    IID_IStudent, (void**)(&isp[ix]));
              assert(SUCCEEDED(hr));
              hr = isp[ix]->QueryInterface(IID_IPerson, (void**)&ipp[ix]);
              assert(SUCCEEDED(hr));
        }
        for(ix = 0; ix < entries; ix++) {
              ipp[ix]->setPersonInfo(peopleTable[ix][0], peopleTable[ix][1],
                                     peopleTable[ix][2], peopleTable[ix][3],
                                     convert(peopleTable[ix][4]),
                                     convert(peopleTable[ix][5]));
              isp[ix]->setStudentInfo(studentTable[ix][0], studentTable[ix][1]);
        }
        cout << "Studen Info    : " << endl;
        cout << "---------------\n" << endl;
        for(ix = 0; ix < entries; ix++) {
              Date* d = new Date;
              ipp[ix]->getName(temp);
              cout << "Name         : " << temp << " ";
              ipp[ix]->getLastName(temp);
              cout << temp << endl;
              ipp[ix]->getAddress(temp);
              cout << "Address      : " << temp << endl;
              ipp[ix]->getBirthDate(d);
              cout << "Birth Date   : " << d->month << " "
                   << d->day << ", " << d->year << endl;
              isp[ix]->getSchoolName(temp);
              cout << "School       : " << temp << endl;
              isp[ix]->getCurric(temp);
              cout << "Curric       : " << temp << endl;
              long* age = new long;
              ipp[ix]->getAge(age);
              cout << "Age          : " << *age << "\n" << endl;
        }
        bbSystem::shutdown();
        return 1;
}

BB_Proper_Export_Flag bool exitModule() {
  return 1;
}

long convert( unsigned char* str) {
        long value = 0;
        long power = 1;
        int digits = strlen((char*) str);
        for(int ix = 0; ix < digits; ix++) {
              long m = digits - (ix + 1);
              for(int iy = 0; iy < m; iy++) {
                    power *= 10;
```

```
            }
            value += (str[ix]-48)*power;
            power = 1;
        }
        return value;
}
//End File Module.c
```

APPENDIX D. XPCOM MODULE SOURCE CODE

```
// ***************************************************************************
//   EXECUTIVE SUMMARY
// File Name: IPerson.idl
// Description: Decleration of IPerson interface
// Author: Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file defines IPerson interface in ORB IDL
// ***************************************************************************

#include "nsISupports.idl"

%{C++
typedef struct Date
{
      long day;
      char* month;
      long year;
}Date;
%}
[ptr] native Date(Date);

[scriptable, uuid(162641E1-8F17-11d3-84EA-00105A0DEB6F)]
interface IPerson : nsISupports
{
      attribute string name;
      attribute string lastName;
      attribute string address;
      attribute Date birthDate;
      attribute long age;
      nsresult setPersonInfo(in string name, in string lName,
                             in string addr, in string monthOfBirth,
                             in long dayOfBirth, in long yearOfBirth);
};

%{ C++
// {710E2253-8F57-11d3-8896-F2A82A000000}
#define PERSONOBJECT_CID \
{ 0x710e2253, 0x8f57, 0x11d3, { 0x88, 0x96, 0xf2, 0xa8, 0x2a, 0x0, 0x0, 0x0 } };
%}
//End File IPerson.idl

// ***************************************************************************
//   EXECUTIVE SUMMARY
// File Name: Person.h
// Description: Decleration of Person class
// Author(s): Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file defines Person class as a XPCOM object
// ***************************************************************************

#ifndef _PERSON_H_
#define _PERSON_H_

#include "nsIServiceManager.h"
#include <iostream.h>
#include "IPerson.h"

static PRInt32 gLockCnt = 0;
static PRInt32 gInstanceCnt = 0;
```

```
//#define VERSION2
class Person : public IPerson {
private :
      Date birthDate;
      char* name;
      char* lastName;
#ifdef VERSION2
      long age;
#endif
      nsISupports *m_pnsISupsOuter;
protected :
      char* address;
public:
      Person(nsISupports *pnsISupsOuter);
      virtual ~Person();
      NS_DECL_IPERSON
      NS_DECL_ISUPPORTS
      NS_IMETHOD InternalQueryInterface(const nsIID & uuid, void * *result);
      NS_IMETHOD_(nsrefcnt) InternalAddRef(void);
      NS_IMETHOD_(nsrefcnt) InternalRelease(void);

      class XNDnsISupports : public nsISupports {
            Person* This() {
                  return (Person*)((uint8*)this -
                                    offsetof(Person, m_innernsISupports));
            }
            NS_IMETHODIMP QueryInterface(const nsIID & uuid, void * *result) {
                  return This()->InternalQueryInterface(uuid, result);
            }
            NS_IMETHODIMP_(nsrefcnt) AddRef(void)
                  { return This()->InternalAddRef(); }
            NS_IMETHODIMP_(nsrefcnt) Release(void)
                  { return This()->InternalRelease(); }
      };
            XNDnsISupports m_innernsISupports;
};

class PersonFactory : public nsIFactory {
public:
      PersonFactory();
    virtual ~PersonFactory();
    NS_DECL_ISUPPORTS
    NS_IMETHOD CreateInstance(nsISupports *aOuter,
                              const nsIID &aIID,
                              void **aResult);
    NS_IMETHOD LockFactory(PRBool aLock);
};
#endif
//End File Person.h

// ********************************************************************
//   EXECUTIVE SUMMARY
// File Name: Person.c
// Description: Implementation of Person class
// Author: Mithat Daglar, daglar@cs.nps.navy.mil
```

```
// Informal: This file implements Person class as XPCOM object
// ***********************************************************************

#include "Person.h"
#include <time.h>

Person::Person(nsISupports *pnsISupsOuter) {
    NS_INIT_ISUPPORTS();
      gInstanceCnt++;
      if(pnsISupsOuter)
            m_pnsISupsOuter = pnsISupsOuter;
      else
            m_pnsISupsOuter = &m_innernsISupports;
}

Person::~Person() {
    NS_ASSERTION(mRefCnt == 0,"Wrong ref count");
    //PR_AtomicDecrement(&gInstanceCnt);
      gInstanceCnt--;
}

NS_IMETHODIMP Person::InternalQueryInterface(const nsIID &aIID, void **aResult)
{
      if (!aResult)
            return NS_ERROR_NULL_POINTER;
      if (aIID.Equals(NS_GET_IID(nsISupports))) {
        *aResult = NS_STATIC_CAST(nsISupports*,(&m_innernsISupports));
      }
      else if (aIID.Equals(NS_GET_IID(IPerson))) {
            *aResult = NS_STATIC_CAST(nsISupports*,
                                      NS_STATIC_CAST(IPerson*,this));
      }
      else {
        *aResult = nsnull;
        return NS_ERROR_NO_INTERFACE;
      }
    NS_ADDREF(NS_STATIC_CAST(nsISupports*,*aResult));
    return NS_OK;
}

NS_IMETHODIMP Person::QueryInterface(const nsIID &aIID, void **aResult) {
      return m_pnsISupsOuter->QueryInterface(aIID, aResult);
}

NS_IMETHODIMP_(nsrefcnt) Person::InternalAddRef(void) {
      NS_PRECONDITION(PRInt32(mRefCnt) >= 0, "illegal refcnt");
      ++mRefCnt;
      NS_LOG_ADDREF(this, mRefCnt, #_class, sizeof(*this));
      return mRefCnt;
}

NS_IMETHODIMP_(nsrefcnt) Person::AddRef(void) {
      return m_pnsISupsOuter->AddRef();
}

NS_IMETHODIMP_(nsrefcnt) Person::InternalRelease(void) {
      NS_PRECONDITION(0 != mRefCnt, "dup release");
```

```
        --mRefCnt;
        NS_LOG_RELEASE(this, mRefCnt, #_class);
        if (mRefCnt == 0) {
              mRefCnt = 1; /* stabilize */
              NS_DELETEXPCOM(this);
              return 0;
        }
        return mRefCnt;
}

NS_IMETHODIMP_(nsrefcnt) Person::Release(void) {
        return m_pnsISupsOuter->Release();
}

NS_IMETHODIMP Person::GetName(char * *aName) {
        strcpy(*aName, name);
        return NS_OK;
}

NS_IMETHODIMP Person::SetName(const char * aName) {
        if(name) delete [] name;
        name = new char[strlen(aName) + 1];
        strcpy(name, aName);
        return NS_OK;
}

NS_IMETHODIMP Person::GetLastName(char * *aLastName) {
        strcpy(*aLastName, lastName);
        return NS_OK;
}

NS_IMETHODIMP Person::SetLastName(const char * aLastName) {
        if(lastName) delete [] lastName;
        lastName = new char[strlen(aLastName) + 1];
        strcpy(lastName, aLastName);
        return NS_OK;
}

NS_IMETHODIMP Person::GetAddress(char * *aAddress) {
        strcpy(*aAddress, address);
        return NS_OK;
}

NS_IMETHODIMP Person::SetAddress(const char * aAddress) {
        if(address) delete [] address;
        address = new char[strlen(aAddress) + 1];
        strcpy(address, aAddress);
        return NS_OK;
}

NS_IMETHODIMP Person::GetBirthDate(Date * *aBirthDate) {
        (*aBirthDate)->day = birthDate.day;
        (*aBirthDate)->year = birthDate.year;
        (*aBirthDate)->month = new char[strlen(birthDate.month) + 1];
        strcpy(((*aBirthDate)->month),birthDate.month);
        return NS_OK;
}
```

```
long getThisYear(void);  //Global Non-Member Function

NS_IMETHODIMP Person::SetBirthDate(Date * aBirthDate) {
      birthDate.day = aBirthDate->day;
      birthDate.year = aBirthDate->year;
      birthDate.month = new char[strlen(aBirthDate->month) + 1];
      strcpy(birthDate.month, aBirthDate->month);
#ifdef VERSION2
      age = getThisYear() - aBirthDate->year;
#endif
      return NS_OK;
}

NS_IMETHODIMP Person::GetAge(PRInt32 *aAge) {
#ifdef VERSION2
      *aAge = age;
#else
      *aAge = 2000 - birthDate.year;
#endif;
      return NS_OK;
}

long getThisYear() {
      time_t t;
      time(&t);
      struct tm* t2 = localtime(&t);
      return (long)(t2->tm_year + 1900);
}

NS_IMETHODIMP Person::SetAge(PRInt32 aAge) {
#ifdef VERSION2
      age = aAge;
#endif
      return NS_OK;
}

NS_IMETHODIMP Person::SetPersonInfo(const char *aName, const char *lName,
                                    const char *addr, const char *monthOfBirth,
                                    PRInt32 dayOfBirth, PRInt32 yearOfBirth,
                                    nsresult *_retval) {
      SetName(aName);
      SetLastName(lName);
      SetAddress(addr);
      Date* d = new Date;
      d->month = new char[strlen(monthOfBirth) + 1];
      strcpy(d->month,  monthOfBirth);
      d->day = dayOfBirth;
      d->year = yearOfBirth;
      SetBirthDate(d);
      return NS_OK;
}

PersonFactory::PersonFactory () {
      NS_INIT_ISUPPORTS();
    //PR_AtomicIncrement(&gInstanceCnt);
      gInstanceCnt++;
```

```
}

PersonFactory::~PersonFactory () {
      NS_ASSERTION(mRefCnt == 0,"Wrong ref count");
    //PR_AtomicDecrement(&gInstanceCnt);
      gInstanceCnt--;
}

NS_IMETHODIMP PersonFactory::QueryInterface(const nsIID &aIID, void **aResult) {
      if (!aResult)
            return NS_ERROR_NULL_POINTER;
    if (aIID.Equals(NS_GET_IID(nsISupports))) {
        *aResult = NS_STATIC_CAST(nsISupports*,this);
    }
      else if (aIID.Equals(NS_GET_IID(nsIFactory))) {
        *aResult =
NS_STATIC_CAST(nsISupports*,NS_STATIC_CAST(nsIFactory*,this));
    }
      else {
        *aResult = nsnull;
        return NS_ERROR_NO_INTERFACE;
    }
    NS_ADDREF(NS_STATIC_CAST(nsISupports*,*aResult));
    return NS_OK;
}

NS_IMPL_ADDREF(PersonFactory)
NS_IMPL_RELEASE(PersonFactory)

NS_IMETHODIMP
PersonFactory::CreateInstance(nsISupports *aOuter,
                              const nsIID &aIID,
                                 void **aResult)    {
      if (!aResult)
            return NS_ERROR_NULL_POINTER;

      *aResult = nsnull;

      if(aOuter != 0 && !aIID.Equals(NS_GET_IID(nsISupports))) {
            *aResult = 0;
            return NS_ERROR_INVALID_ARG;
      }
    Person *inst = new Person(aOuter);
    if (!inst) {
          return NS_ERROR_OUT_OF_MEMORY;
    }
    NS_ADDREF(inst);
    nsresult rv = inst->InternalQueryInterface(aIID, aResult);
      NS_RELEASE(inst);
   return rv;
}

NS_IMETHODIMP PersonFactory::LockFactory(PRBool aLock) {
      if (aLock) {
            gLockCnt++;
      }
      else {
```

```
            gLockCnt--;
        }
    return NS_OK;
}

static NS_DEFINE_CID(kPersonObjectCID, PERSONOBJECT_CID);
static NS_DEFINE_CID(kComponentManagerCID, NS_COMPONENTMANAGER_CID);

static const char* GetDesc(void) {
      static char desc[] = "Person Object";
    return desc;
}

extern "C" NS_EXPORT nsresult NSGetFactory(nsISupports *serviceMgr,
                                           const nsCID &aCID,
                                           const char *aClassName,
                                           const char *aProgID,
                                           nsIFactory **aResult) {

      if (!aResult)
            return NS_ERROR_NULL_POINTER;
      *aResult = nsnull;
      PersonFactory *inst;
      if (aCID.Equals(kPersonObjectCID)) {
        inst = new PersonFactory();
      }
      else {
            return NS_ERROR_NO_INTERFACE;
      }
      if (!inst)
            return NS_ERROR_OUT_OF_MEMORY;
      NS_ADDREF(inst);
      nsresult rv = inst->QueryInterface(NS_GET_IID(nsIFactory),
                                         (void **) aResult);
      NS_RELEASE(inst);
      return rv;
}

extern "C" NS_EXPORT PRBool NSCanUnload(nsISupports* serviceMgr) {
      return PRBool(gInstanceCnt == 0 && gLockCnt == 0);
}

extern "C" NS_EXPORT nsresult NSRegisterSelf(nsISupports *aServMgr,
                                             const char *path) {
      nsresult rv = NS_OK;
      nsIServiceManager *sm;
      rv = aServMgr->QueryInterface(NS_GET_IID(nsIServiceManager),
                                    (void **)&sm);
      if (NS_FAILED(rv))
            return rv;
      nsIComponentManager *cm;
      rv = sm->GetService(kComponentManagerCID, NS_GET_IID(nsIComponentManager),
                          (nsISupports **)&cm);
      if (NS_FAILED(rv)) {
            NS_RELEASE(sm);
            return rv;
      }
```

```
        rv = cm->RegisterComponent(kPersonObjectCID, GetDesc(),
                                   "component://bamboo/xpcom/Person",
                                   path, PR_TRUE, PR_TRUE);
        sm->ReleaseService(kComponentManagerCID, cm);
        NS_RELEASE(sm);

#ifdef NS_DEBUG
        printf("*** %s registered\n",GetDesc());
#endif
        return rv;
}

extern "C" NS_EXPORT nsresult NSUnregisterSelf(nsISupports* aServMgr,
                                                const char *path) {
        nsresult rv = NS_OK;
        nsIServiceManager *sm;
        rv = aServMgr->QueryInterface(NS_GET_IID(nsIServiceManager),
                                      (void **)&sm);
        if (NS_FAILED(rv))
                return rv;
        nsIComponentManager *cm;
        rv = sm->GetService(kComponentManagerCID, NS_GET_IID(nsIComponentManager),
                            (nsISupports **)&cm);
        if (NS_FAILED(rv)) {
                NS_RELEASE(sm);
                return rv;
        }
        rv = cm->UnregisterComponent(kPersonObjectCID, path);
        sm->ReleaseService(kComponentManagerCID, cm);
        NS_RELEASE(sm);
        return rv;
}
//End Person.c

// ***********************************************************************
//   EXECUTIVE SUMMARY
// File Name: Student.idl
// Description: Decleration of IStudent Interface
// Author(s): Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file declares IStudent Interface
// ***********************************************************************

#include "nsISupports.idl"

[scriptable, uuid(710E2251-8F57-11d3-8896-F2A82A000000)]
interface IStudent : nsISupports
{
        attribute string schoolName;
        attribute string curric;
        void setStudentInfo(in string school, in string curric);
};


%{ C++
// {710E2254-8F57-11d3-8896-F2A82A000000}
#define STUDENTOBJECT_CID \
```

```
{ 0x710e2254, 0x8f57, 0x11d3, { 0x88, 0x96, 0xf2, 0xa8, 0x2a, 0x0, 0x0, 0x0 } };
%}
//End File Student.idl

// ********************************************************************
//   EXECUTIVE SUMMARY
// File Name: Student.h
// Description: Decleration of Student class
// Author: Mithat Daglar, daglar@cs.nps.navy.mil
// Informal: This file defines Student class as XPCOM object that
//                 aggragates Person XPCOM object
// ********************************************************************

#ifndef _STUDENT_H_
#define _STUDENT_H_

#include "nsIServiceManager.h"
#include "IStudent.h"
#include "IPerson.h"
#include <iostream.h>

static PRInt32 gLockCnt = 0;
static PRInt32 gInstanceCnt = 0;

class Student : public IStudent {
      char* schoolName;
      char* curric;
      IPerson* m_Person;
public:
      Student();
      virtual ~Student();

      //nsISupports Methods
      NS_DECL_ISUPPORTS

      //IStudent methods
      NS_DECL_ISTUDENT
};

class StudentFactory : public nsIFactory {
public:
      StudentFactory();
      virtual ~StudentFactory();

      //nsISupports Methods
      NS_DECL_ISUPPORTS

      //nsIFactory Methods
      NS_IMETHOD CreateInstance(nsISupports *aOuter,
                                const nsIID &aIID,
                                void **aResult);
    NS_IMETHOD LockFactory(PRBool aLock);
};
#endif
//End File Student.h
```

```
// ************************************************************************
//   EXECUTIVE SUMMARY
// File Name: Student.c
// Description: Implementation of Student class
// Author(s): Mithat Daglar, dgalar@cs.nps.navy.mil
// Informal: This file implements Student class as a XPCOM object that
//                 aggragates Person XPCOM object
// ************************************************************************

#include "Student.h"

Student::Student() {
      NS_INIT_ISUPPORTS();
      gInstanceCnt++;
      mRefCnt++;
      nsresult rv;
      rv = nsComponentManager::CreateInstance("component://bamboo/xpcom/Person",
                                              this, NS_GET_IID(nsISupports),
                                              (void**)&m_Person);
      if (NS_FAILED(rv)) {
            printf("In Student::Student Failed to create instance IPerson\n");
            exit(1);
    }
      mRefCnt--;
}

Student::~Student() {
      NS_ASSERTION(mRefCnt == 0,"Wrong ref count");
      gInstanceCnt--;
}

NS_IMETHODIMP Student::QueryInterface(const nsIID &aIID, void **aResult) {
      if (!aResult)
            return NS_ERROR_NULL_POINTER;
      if (aIID.Equals(NS_GET_IID(nsISupports))) {
            *aResult = NS_STATIC_CAST(nsISupports*,this);
      }
      else if (aIID.Equals(NS_GET_IID(IStudent))) {
            *aResult = NS_STATIC_CAST(nsISupports*,
                                      NS_STATIC_CAST(IStudent*,this));
      }
      else if(aIID.Equals(NS_GET_IID(IPerson))) {
            nsresult rv = m_Person->QueryInterface(aIID, aResult);
            return rv;
      }
      else {
            *aResult = nsnull;
            return NS_ERROR_NO_INTERFACE;
      }
      NS_ADDREF(NS_STATIC_CAST(nsISupports*,*aResult));
      return NS_OK;
}

NS_IMPL_ADDREF(Student)
NS_IMPL_RELEASE(Student)

NS_IMETHODIMP Student::GetSchoolName(char * *aSchoolName) {
```

```
        if(!schoolName) {
                strcpy(*aSchoolName, "NOT-SPECIFIED");
        }
        else {
                strcpy(*aSchoolName, schoolName);
        }
        return NS_OK;
}

NS_IMETHODIMP Student::SetSchoolName(const char * aSchoolName) {
        if(!strcmp(aSchoolName, " ")) {
                schoolName = NULL;
        }
        else {
                schoolName = new char[strlen(aSchoolName) + 1];
                strcpy(schoolName, aSchoolName);
        }
        return NS_OK;
}
NS_IMETHODIMP Student::GetCurric(char * *aCurric) {
        strcpy(*aCurric, curric);
        return NS_OK;
}
NS_IMETHODIMP Student::SetCurric(const char * aCurric) {
        curric = new char[strlen(aCurric) + 1];
        strcpy(curric, aCurric);
        return NS_OK;
}

NS_IMETHODIMP Student::SetStudentInfo(const char *school, const char *curric) {
        SetSchoolName(school);
        SetCurric(curric);
        return NS_OK;
}

// Class StudentFactory
StudentFactory::StudentFactory () {
        NS_INIT_ISUPPORTS();
        gInstanceCnt++;
}

StudentFactory::~StudentFactory ()    {
        NS_ASSERTION(mRefCnt == 0,"Wrong ref count");
        gInstanceCnt--;
}

NS_IMETHODIMP StudentFactory::QueryInterface(const nsIID &aIID, void **aResult)
{
        if (!aResult)
                return NS_ERROR_NULL_POINTER;
        if (aIID.Equals(NS_GET_IID(nsISupports))) {
                *aResult = NS_STATIC_CAST(nsISupports*,this);
        }
        else if (aIID.Equals(NS_GET_IID(nsIFactory))) {
                *aResult =
NS_STATIC_CAST(nsISupports*,NS_STATIC_CAST(nsIFactory*,this));
        }
```

```
        else {
                *aResult = nsnull;
                return NS_ERROR_NO_INTERFACE;
        }
        NS_ADDREF(NS_STATIC_CAST(nsISupports*,*aResult));
        return NS_OK;
}

NS_IMPL_ADDREF(StudentFactory)
NS_IMPL_RELEASE(StudentFactory)

NS_IMETHODIMP StudentFactory::CreateInstance(nsISupports *aOuter,
                                             const nsIID &aIID, void **aResult)
{
        if (!aResult)
                return NS_ERROR_NULL_POINTER;
        *aResult = nsnull;
        if (aOuter)
                return NS_ERROR_NO_AGGREGATION;
        Student *inst = new Student();
        if (!inst) {
                return NS_ERROR_OUT_OF_MEMORY;
        }
        NS_ADDREF(inst);
        nsresult rv = inst->QueryInterface(aIID, aResult);
        NS_RELEASE(inst);
        return rv;
}

NS_IMETHODIMP StudentFactory::LockFactory(PRBool aLock)  {
        if (aLock) {
                gLockCnt++;
        }
        else {
                gLockCnt--;
        }
        return NS_OK;
}

static NS_DEFINE_CID(kStudentObjectCID, STUDENTOBJECT_CID);
static NS_DEFINE_CID(kComponentManagerCID, NS_COMPONENTMANAGER_CID);

static const char* GetDesc(void)  {
        static char desc[] = "Student Object";
        return desc;
}

extern "C" NS_EXPORT nsresult NSGetFactory(nsISupports *serviceMgr,
                                           const nsCID &aCID,
                                           const char *aClassName,
                                           const char *aProgID,
                                           nsIFactory **aResult)   {
        if (!aResult)
                return NS_ERROR_NULL_POINTER;
        *aResult = nsnull;
        StudentFactory *inst;
        if (aCID.Equals(kStudentObjectCID)) {
```

```
            inst = new StudentFactory();
        }
        else {
            return NS_ERROR_NO_INTERFACE;
        }
        if (!inst)
            return NS_ERROR_OUT_OF_MEMORY;
        NS_ADDREF(inst);
        nsresult rv = inst->QueryInterface(NS_GET_IID(nsIFactory),
                                           (void **) aResult);
        NS_RELEASE(inst);
        return rv;
}

extern "C" NS_EXPORT PRBool NSCanUnload(nsISupports* serviceMgr)  {
        return PRBool(gInstanceCnt == 0 && gLockCnt == 0);
}

extern "C" NS_EXPORT nsresult NSRegisterSelf(nsISupports *aServMgr,
                                             const char *path)  {
        nsresult rv = NS_OK;
        nsIServiceManager *sm;
        rv = aServMgr->QueryInterface(NS_GET_IID(nsIServiceManager),
                                      (void **)&sm);
        if (NS_FAILED(rv))
            return rv;
        nsIComponentManager *cm;
        rv = sm->GetService(kComponentManagerCID, NS_GET_IID(nsIComponentManager),
                            (nsISupports **)&cm);
        if (NS_FAILED(rv)) {
            NS_RELEASE(sm);
            return rv;
        }
        rv = cm->RegisterComponent(kStudentObjectCID, GetDesc(),
                                   "component://bamboo/xpcom/Student",
                                   path, PR_TRUE, PR_TRUE);
        sm->ReleaseService(kComponentManagerCID, cm);
        NS_RELEASE(sm);
#ifdef NS_DEBUG
        printf("*** %s registered\n",GetDesc());
#endif
        return rv;
}

extern "C" NS_EXPORT nsresult NSUnregisterSelf(nsISupports* aServMgr,
                                               const char *path)  {
        nsresult rv = NS_OK;
        nsIServiceManager *sm;
        rv = aServMgr->QueryInterface(NS_GET_IID(nsIServiceManager),
                                      (void **)&sm);

        if (NS_FAILED(rv))
            return rv;
        nsIComponentManager *cm;
        rv = sm->GetService(kComponentManagerCID, NS_GET_IID(nsIComponentManager),
                            (nsISupports **)&cm);
```

```
        if (NS_FAILED(rv)) {
                NS_RELEASE(sm);
                return rv;
        }
        rv = cm->UnregisterComponent(kStudentObjectCID, path);
        sm->ReleaseService(kComponentManagerCID, cm);
        NS_RELEASE(sm);
        return rv;
}
//End File Student.c

// ***********************************************************************
//   EXECUTIVE SUMMARY
// Module Name: module.c
// Description: Definition of module.c
// Author(s): Mithat Daglar, dgalar@cs.nps.navy.mil
// Informal: This file tests Student and Person
//           XPCOM objects
// ***********************************************************************

#include "bbSystem.h"
#include "module.h"
#include "nsIServiceManager.h"
#include <iostream.h>
#include "IPerson.h"
#include "IStudent.h"
#include <assert.h>
#include "PeopleInfo.h"

long convert(char* str);

bool initModule()  {
        cout << "Welcome to XPCOM test program" << endl;
        char temp[50];
        char **temp2;
        *temp2 = new char[50];
        const int entries = sizeof(peopleTable) / sizeof(*peopleTable);
        IPerson *ipp[entries];
        IStudent *isp[entries];

        for(int ix = 0; ix < entries; ix++) {

                nsresult rv = nsComponentManager::CreateInstance(
                                "component://bamboo/xpcom/Student",
                                 nsnull, NS_GET_IID(IStudent),
                                 (void**)(&isp[ix]));
                assert(NS_SUCCEEDED(rv));
                rv = isp[ix]->QueryInterface(NS_GET_IID(IPerson), (void**)&ipp[ix]);
                assert(NS_SUCCEEDED(rv));
        }
        for(ix = 0; ix < entries; ix++) {
                ipp[ix]->SetPersonInfo(peopleTable[ix][0], peopleTable[ix][1],
                                       peopleTable[ix][2], peopleTable[ix][3],
                                       convert(peopleTable[ix][4]),
                                       convert(peopleTable[ix][5]), 0);
                isp[ix]->SetStudentInfo(studentTable[ix][0], studentTable[ix][1]);
        }
```

```
        cout << "Studen Info    : " << endl;
        cout << "----------------\n" << endl;
        for(ix = 0; ix < entries; ix++) {
              Date* d = new Date;
              ipp[ix]->GetName(temp2);
              cout << "Name          : " << *temp2 << " ";
              ipp[ix]->GetLastName(temp2);
              cout << *temp2 << endl;
              ipp[ix]->GetAddress(temp2);
              cout << "Address       : " << *temp2 << endl;
              ipp[ix]->GetBirthDate(&d);
              cout << "Birth Date    : " << d->month << " "
                   << d->day << ", " << d->year << endl;
              isp[ix]->GetSchoolName(temp2);
              cout << "School        : " << *temp2 << endl;
              isp[ix]->GetCurric(temp2);
              cout << "Curric        : " << *temp2 << endl;
              int* age = new int;
              ipp[ix]->GetAge(age);
              cout << "Age           : " << *age << "\n" << endl;
        }
        bbSystem::shutdown();
        return 1;
}

bool exitModule()  {
  return 1;
}

long convert(char* str) {
        long value = 0;
        long power = 1;
        int digits = strlen((char*) str);
        for(int ix = 0; ix < digits; ix++) {
              long m = digits - (ix + 1);
              for(int iy = 0; iy < m; iy++) {
                    power *= 10;
              }
              value += (str[ix]-48)*power;
              power = 1;
        }
        return value;
}
//End File Module.c
```

LIST OF REFERENCES

[1] Kent Watsen and Mike Zyda, *Bamboo – A Portable System for Dynamically Extensible, Real-time, Networked, Virtual Environments*, March 1998.

[2] Microsoft Development Network (MSDN) Library.

[3] Bamboo Web Site, [http://www.watsen.net/Bamboo], October 1999.

[4] Don Box, *Essential COM,* 1998

[5] Michi Henning and Steve Vinoski, *Advenced CORBA Programming with C++,* 1999.

[6] Sara Williams and Charlie Kindel, *The Component Object Model: A Technical Overview*, 1994.

[7] Roger Sessions, *COM and DCOM*, 1998.

[8] Kraig Brockschmidt, *What OLE Is Really About,* 1996.

[9] Steve Robinson and Alex Krassel, *COMponents,* August 1997.

[10] Mozilla.org at a Glance, [http://www.mozilla.org/mozorg.html], January 2000.

[11] Johnny Stenback and Heikki Toivonen, *Extending Mozilla or How to Do the Impossible*, October 1999.

[12] Module Owners, [http://www.mozilla.org/owners.html], October 1999.

[13] Owen Tallman and J. Bradford Kain, *COM versus CORBA: A Decision Framework*, September 1998.

[14] Roger Session, Andreas Vogel and Eugene Kim, *COM versus CORBA.*

[15] XPCOM Web Site, [http://www.mozilla.org/projects/xpcom/], November 1999.

[16] Will Scullin, "Modularization Techniques", [http://www.mozilla.org/docs/modunote.htm]. February 1998.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Deniz Kuvvetleri Komutanligi 1
   Personel Daire Baskanligi
   Bakanliklar
   Ankara, TURKEY

4. Deniz Kuvvetleri Komutanligi 1
   Kutuphanesi
   Bakanliklar
   Ankara, TURKEY

5. Deniz Harp Okulu 2
   Kutuphanesi
   Tuzla
   Istanbul, TURKEY

6. Chairman, Code CS 1
   Naval Postgraduate School
   Monterey, CA 93943-5101

7. Prof. Michael Zyda, Code CS/ZK 1
   Naval Postgraduate School
   Monterey, CA 93943-5100

8. Research Assistant Prof. Michael V. Capps, Code CS/CM 1
   Naval Postgraduate School
   Monterey, CA 93943-5100

9. Kent Watsen 1
   108 Cottage Grove Ave.
   San Mateo, CA 94401

10. Yazilim Gelistirme Grup Baskanligi 1
    Deniz Harp Okulu Komutanligi
    Tuzla
    Istanbul, TURKEY

11. LTJG. Mithat Daglar 2
    Yuksek Sokak – Onurkent Sitesi
    D2 Blok D3
    81570 - Kucukyali
    Istanbul, TURKEY