A DETAILED ANALYSIS OF THE COMPONENT OBJECT MODEL

By

ARIF MULJADI

Bachelor of Science

Padjadjaran University

Bandung, Indonesia

1989

Submitted to the Faculty of the

Graduate College of the

Oklahoma State University

in partial requirements for

the Degree of

MASTER OF SCIENCE

MAY 2006

A DETAILED ANALYSIS OF THE
COMPONENT OBJECT MODEL

PREFACE

Microsoft's Component Object Model (COM) is a system for reusing software artifacts at the post-compilation level. Several problems have been raised about the Component Object Model [Sullivan et al. 99]. The problems are about a conflict between interface negotiation and aggregation, a conclusion about transitivity, and the identity of the inner components. However, the conflict between interface negotiation and aggregation exists only if a questionable definition is used. The formal model used to depict the conflict is also inadequate.

This thesis concerns an investigation of COM criticisms. A set of programs is used to demonstrate the consistency of the Component Object Model rules regarding interface negotiation and aggregation.

ACKNOWLEDGEMENT

I would like to express my highest gratitude to my thesis advisor Dr. M. H. Samadzadeh for his invaluable support and guidance for the completion of this thesis. With his great patience, spending of his precious time, encouragement, and boundless help, I was finally able to complete this thesis.

I would like to thank the committee members, Dr. Blayne E. Mayfield and Dr. Nohpill Park. I would like to convey my special thanks to Dr. John P. Chandler, the Departmental Graduate Coordinator, for his support.

I would like also to thank Republic Indonesia’s Agency for the Assessment and Application of Technology (BPP Teknologi) for providing me scholarship for part of my study in this Masters program. Also, I thank the Indonesian Aircraft Industries (PT Dirgantara Indonesia, a.k.a. PT Industri Pesawat Terbang Nusantara) where I am still listed as an employee. I would like to thank them for the permission to pursue my studies abroad.

Last, but certainly not least, I would like to express my utmost gratefulness to my mother, my wife, and my sister for their constant support for finishing this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

CHAPTER I

INTRODUCTION

Microsoft's Component Object Model (COM) is a system for reusing software artifacts at the post-compilation level or binary level [Microsoft 05e]. Using COM, one or more computational functionalities of a component can be made available through a system call and through subroutines called interfaces. A client of a component, i.e., a program that needs the functionalities of the component, is served through a protocol called interface negotiation [Microsoft 03]. Several components may be combined into one component using the aggregation method to put all or part of the functionalities of the constituent components into the new component.

Sullivan et al. [Sullivan et al. 99] have shown that there are several problems with the Microsoft's Component Object Model concept. The first problem is a conflict between component aggregation and interface negotiation. This thesis shows that the existence of such a conflict stems from using a definition used by Sullivan et al.

In this thesis, a set of programs that show the consistency of interface negotiation and aggregation are presented.

CHAPTER II

BACKGROUND

Component Object Model provides a facility to reuse software artifacts. Several programs running on a system (a personal computer running an operating system) can share loaded COM software (called a server) to get a number of computational processes. The programs' source codes may be from different programming languages.

There are several ways to reuse software artifacts including reusing source code, reusing flowcharts or diagrammatic expression, and reusing concepts. The stated cases of artifact reuse can be categorized as the pre-compilation ones. Reusing a .dll or a .exe file is a case of reuse at the binary level for software artifacts. A change in a non-COM .dll or .exe library usually requires a significant amount of work for adjusting it to a specific a client. COM, a standard of reusing software artifacts at the binary level, helps reduce the amount of such adjustments at a client side.

A further way of reuse that is utilized by COM is aggregation. Aggregation is reuse of a COM component. An aggregation component consists of one outer component and one or more inner components. At run time, the instance of the outer component, called outer object, is the one that deals with the clients. In turn, the outer object will make a request to the system to generate an instance of the inner component, called an inner object.

The outer object, as coded in the outer component, may give its client a pointer to an interface implemented in the inner object. This is the heart of COM aggregation, i.e., an instance of a component (the outer component) reuses an implementation made in another component (the inner component) or is contained in an instance of the inner component. The outer component may make only a part of the inner component's interfaces available to its client.

Sullivan et al. [Sullivan et al. 99] tried to use COM aggregation to implement an architecture called Mediator-Based Architecture. This architecture requires a client to be able to access any of the inner component's interfaces. This is clearly against the specification of aggregation.

CHAPTER III

COMPONENT OBJECT MODEL

Microsoft's Component Object Model was devised to facilitate the reuse of software artifacts at the post-compilation level or at the binary level. The system utilizes interfaces and components (i.e., subroutines and structures designed in a certain way), an identification scheme for software artifacts and interfaces, and system calls.

For a user of the computational service provided by this reuse system, a component is the unit of reuse. The client of a service makes a system call requesting the operating system to instantiate an object of a component.

A component has a set of interfaces [Microsoft 05b] [Microsoft 05e]. There is an interface that must be implemented by each component, called interface IUnknown. This interface has three methods: QueryInterface, Add, and Release. QueryInterface handles a client's request to a component for an interface. Add and Release keep track of the number of current clients for the components in the system.

Every component and interface has a 128-bit identification number. A client passes a component identification number when it requests the system to instantiate an object of the component. Having the pointer to a component object, the client should pass an interface identification number when requesting an interface from the component object.

3.1 Component

From a client's point of view, a component is a unit of object instantiation at run time. From the point of view of the code, a component is a house for a collection of interfaces.

A compiled and linked component is stored in a container file with extension .exe or .dll that can be accessed during run time by a client. Every container is equipped with an instantiator that is called by the operating system when needed.

3.2 Interface

An interface is a set of method prototypes. To utilize a method of an interface, a client should get a pointer to the interface. This pointer, if available, is returned by QueryInterface upon request by a client.

Every component is required to have an IUnknown interface. A client program, which has requested the system to instantiate a component, will have a pointer to the component. Using this pointer and the QueryInterface method of the IUnknown interface, a client can get a pointer to any interface available from the component.

Calling the QueryInterface method asking for an interface is called interface negotiation. Interface negotiation is reflexive, symmetric, and transitive.

3.3 Global Unique Identification

Since the reusing of the components is at run time (i.e., a component is a run time reuse object) and a reuse object may come from various developers around the globe, a scheme that has a capability to provide strong identification is needed.

Microsoft has chosen a 128-bit identification number for components and interfaces [Microsoft 05a].

3.4 Role of the Operating System

The COM scheme needs operating system support for it to be effective. The operating system keeps the necessary information about the available components. Such information includes container files, the path of container files, and the identification numbers. All components must be registered with the operating system before they are made available to be used by a client.

3.5 Aggregation

A part or all of the functionalities of a component can be made part of the collection of functionalities of another component through a mechanism called aggregation [Thompson 05]. This is a reuse scheme. A component that aggregates is called an outer component and the aggregated component is called an inner component.

There are two kind of IUnknown interfaces for a component that can be aggregated (an inner component): delegating unknown and nondelegating unknown. A delegating unknown is simple, it just returns the pointer to the outer component's IUnknown whenever a client queries it. A nondelegating unknown is just like a regular IUnknown, but only the outer component's IUnknown can access it.

3.6 COM and C++

A COM server may be built using the C++ language [Microsoft 05a]. The following subsections describe C++ materials that are relevant to COM [Microsoft 05c] [Microsoft 05d].

3.6.1 Class and Object

A class is a group of variables and functions [Deitel and Deitel 94]. A class is analogous to a structure in the C language except that it may contain any number of subroutines or functions. An instance of a class at run time is called an object. A class may have more than one instance. An object has an address in the computer memory. This address is used to refer to the object.

An instance of a component at a run time is called an object. A call to QueryInterface asking for interface IUnknown returns a pointer that is called an object identity. At run time, after the creation of a component, any call to QueryInterface asking for IUnknown interface by any interface obtained, through an interface negotiation on the object, should always return the same pointer value, which is called the identity of the object [Microsoft 05g].

Modifiers are attribute keywords for further specifying the characteristic of a member of a class. Private and public are keywords for specifying access permissions. A public variable can be accessed by any member of any class in the program. A private variable can be accessed only by a member of the class to which the private member belongs.

3.6.2 Inheritance

Inheritance is inclusion of a class definition in the definition of another class [Deitel and Deitel 94]. The class whose definition is reused in an inheritance is called a base class. The class that reuses another class definition is called a derived class. A derived class can inherit only from one base class. Also, a class can inherit from a derived class.

All variables and functions of a base class become members of the derived class. However, a derived class may modify the attribute of a member that is originally inherited from the base class.

3.6.3 (Pure) Virtual Functions and Abstract Classes

A function that is declared with modifier “virtual” in a base class can be redefined in a derived class. A “pure virtual” function is a function that has no implementation at all. A class that contains at least one pure virtual function is called an abstract class. An abstract class cannot be instantiated, since it has at least one member function that has no definition. The implementation of the declared functions must be provided by a derived class that inherits from that class.

3.6.4 COM Interface Implementation

An interface can be implemented using a class [Microsoft 05c]. This class consists only of function declarations without any implementation. Since an interface consists only of declaration of functions, which specify the return type, number, and type of parameters, an interface is considered a contract. The implementation class should inherit interface IUnknown [Microsoft 05d]. Figure 2.1 illustrates C++ code implementing an interface.

```
/*

These code segments are derived from Rogerson's "Inside
COM" [Rogerson 97]

*/

/*
Interface Interface_A inherits interface IUnknown.
The interface is implemented using an abstract class,
which has at least one pure virtual function/method.
Interface_A has two methods, Method_A_1 and
Method_A_2.
Interface_B has two methods, Method_B_1 and
Method_B_2.
*/

interface Interface_A : IUnknown
{
   virtual void __stdcall Method_A_1() = 0 ;
   virtual void __stdcall Method_A_2() = 0 ;
};

interface Interface_B : IUnknown
{
   virtual void __stdcall Method_B_1() = 0 ;
   virtual void __stdcall Method_B_2() = 0 ;
};
```

Listing 3.1 An Implementation of COM Interfaces Interface_A and Interface_B

3.6.5 COM Component Implementation

A COM component can also be implemented using a C++ class. Since a component is a group of interfaces, a component class must inherit one or more interface classes. A component class must have implementation of the methods defined with modifier "pure virtual" in the interface class. One of the method implementations is for the QueryInterface method.

```
/*
These code segments are derived from Rogerson's "Inside
COM" [Rogerson 97]
*/

/*
COM Component Component_A consists of Interface_A and
Interface_B. In the implementation, it inherits
the classes of Interface_A and Interface_B.
*/

class Component_A : public Interface_A,
                    public Interface_B
{
public:
   //Declaration for methods/functions of IUnknown
   virtual HRESULT __stdcall QueryInterface(const IID& iid,
                                            void** ppv);
   //Methodd from interface Interface_A
   virtual void __stdcall Method_A_1();
   virtual void __stdcall Method_A_2();
   //Methodd from interface Interface_B
   virtual void __stdcall Method_B_1();
   virtual void __stdcall Method_B_2();
}
//Implementation of the QueryInterface method.
HRESULT __stdcall Component_A::
                  QueryInterface(const IID& iid,
                                 void** ppv)
{
   if(iid == IID_Interface_A)
   {
      //Return the pointer to the first interface,
      //i.e. Interface_A
      *ppv = static_cast<Interface_A *>(this);
   }
   else if(iid == IID_Interface_B)
   {
      //Return the pointer to interface Interface_B
      *ppv = static_cast<Interface_B *>(this);
   }
   else
   {
      //Return null pointer for any other query
      *ppv = NULL;
      return E_NOINTERFACE;
   }
   return S_OK;
}
```

Listing 3.2 An Implementation of COM Component Component_A

```
//The implementation of method Method_A_1.
virtual void __stdcall Method_A_1()
{
    .
    .
    .
}

//The implementation of method Method_A_2.
virtual void __stdcall Method_A_2()
{
    .
    .
    .
}

//The implementation of method Method_B_1.
virtual void __stdcall Method_B_1()
{
    .
    .
    .
}

//The implementation of method Method_B_2.
virtual void __stdcall Method_B_2()
{
    .
    .
    .
}
```

Listing 3.2 An Implementation of COM Component Component_A (continued)

3.6.6 COM Aggregation Implementation

Each of the constituent components of an aggregation (the outer and inner components) is implemented with a C++ class. At run time, the creation of an instance of the outer component is requested by a client. The creation of an instance of the inner component is triggered by the outer object. The following listing illustrates an implementation of an aggregation. The code segments in the listing are not complete, some parts that are not relevant to the research are omitted.

```
/*
These code segments are derived from Rogerson's "Inside
COM" [Rogerson 97]
*/

//Interfaces itf1, itf2
interface itf1 : IUnknown
{
   virtual void __stdcall mth1() = 0;
}

interface itf2 : IUnknown
{
   virtual void __stdcall mth2() = 0;
}


/*----------------------------------------------------------
Component Component_Outer
Component_Outer aggregates component Component_Inner
----------------------------------------------------------*/
//This component aggregates component Component_Inner.
class Component_Outer: public itf1
                       //public itf2
                       //This interface is implemented in
                       //  Component_Inner.
{
public:
   // IUnknown
   virtual HRESULT __stdcall QueryInterface(const IID& iid,
                                            void** ppv);

   // Method from interface itf1.
   virtual void __stdcall mth1();

   /*Implementation of mth2 is provided by Component_Inner.
   // Method from interface itf2
   virtual void __stdcall mth2();
   */
}

HRESULT __stdcall Component_Outer::
                  QueryInterface(const IID& iid,
                                 void** ppv)
{
```

Listing 3.3   An Implementation of COM Aggregation

```
   if(iid == IID_IUnknown)
   {
       //Return the pointer to the first interface,
       //  i.e. itf1.
      *ppv = static_cast<IUnknown*>(this);
   }
   else if(iid == IID_Itf1)
   {
       //Return the pointer to the first interface,
       //  i.e. itf1.
      *ppv = static_cast<itf1*>(this);
   }
   else if(iid == IID_Itf2)
   {
      //Return the pointer to interface itf2 which is
      //  actually implemented in Component_Inner.
      //*ppv = static_cast<itf2*>(this);
      return mpUnknownInner0>QueryInterface(iid, ppv);
   }
   else
   {
      //Return null pointer for any other query.
      *ppv = NULL;
      return E_NOINTERFACE;
   }
   return S_OK;
}

//Method mth1
virtual void __stdcall mth1()
{
   printf("This is mth1 of itf1. ");
   printf("Interface itf1 belongs to a component ");
   printf("with identity %p\n", this);
}

/*This is implemented in Component_Inner.
//Method mth2
virtual void __stdcall mth2()
{
   printf("This is mth2 of itf2. ");
   printf("Interface itf2 belongs to a component ");
   printf("with identity %p\n", this);
}
*/
```

Listing 3.3   An Implementation of COM Aggregation (continued)

```
/*----------------------------------------------------------
Component Component_Inner

Component Component_Inner is aggregated by
  Component_Outer.
Component_Inner has nondelegating iunknown interface
  INondelegatingUnknown as the regular IUnknown
  and delegating iunknown IDelegatingUnknown which
  is called when a client requests an interface
  negotiation through one of its interfaces made
  available to a client by the outer component.
----------------------------------------------------------*/
//This component is aggregated by component
//  Component_Outer
class Component_Inner : public itf2
{
public:
   //IUnknown
   virtual HRESULT __stdcall QueryInterface(const IID& iid,
                                            void** ppv);

   // Method from interface itf2
   virtual void __stdcall mth2();
}

HRESULT __stdcall Component_Inner::
                  QueryInterface(const IID& iid,
                                 void** ppv)
{
   //Since this component is aggregated,
   //  this QueryInterface should just return
   //  the result of the aggregator component's
   //  QueryInterface.

   return m_pUnknownOuter->QueryInterface(iid, ppv);

   /* if(iid == IID_IUnknown)
   {
      //Return the pointer to the first interface,
      //i.e. itf1
      *ppv = static_cast<IUnknown*>(this);
   }
   else if(iid == IID_IUnknown)
   {
      //Return the pointer to interface itf2 which is
      //  actually implemented in Component_Inner.
      //*ppv = static_cast<itf2*>(this);
      return mpUnknownInner0>QueryInterface(iid, ppv);
   }
```

Listing 3.3 An Implementation of COM Aggregation (continued)

```
   else
   {
      //Return null pointer for any other query.
      *ppv = NULL;
      return E_NOINTERFACE;
   }
   return S_OK;*/
}
//Method mth2
virtual void __stdcall mth2()
{
   printf("This is mth2 of itf2. ");
   printf("Interface itf2 belongs to a component with ");
   printf(" identity %p", this);
}

//This is the regular QueryInterface of
//  component Component_Inner.
HRESULT __stdcall Component_Inner::
        NondelegatingQueryInterface(const IID& iid,
        void** ppv)
{
   if(iid == IID_IUnknown)
   {
      *ppv = static_cast<NondelegatingUnknown*>(this);
   }
   else if(iid == IID_Itf2)
   {
      //Return the pointer to interface itf2.
      *ppv = static_cast<itf2*>(this);
   }
}
```

Listing 3.3   An Implementation of COM Aggregation (continued)

CHAPTER IV

AN ANALYSIS OF REACHABILITY, AGGREGATION, AND INTERFACE NEGOTIATION

In an aggregation, an outer component may hide any part of an inner component's interfaces. Sullivan et al. asserted that hiding an inner component's interface in an aggregation causes a conflict with the query interface mechanism of the inner component ([Sullivan et al. 99], Section 3.1). According to Sullivan et al., the query interface mechanism of an inner component should be able to return the pointer to any of the implemented interfaces in the inner component, while an aggregation gives a freedom to the outer component to just selectively expose certain inner component's interfaces.

The assertion is based on what they call the reachability property or rule. The definition of reachability is as follows: "if a component exposes a given type of interface, that type of interface should be accessible from any interface on the component" ([Sullivan et al. 99], Subsection 2.3.1). This property is not a part of the original COM specification. COM specification has some definitions that are similar to, but not exactly identical to, the reachability property.

In the following subsections, a review of component identity, mediator-based architecture, backward transitivity, and the formal model for interface negotiation and aggregation ([Sullivan et al. 99], Section 4) are provided.

4.1 Reachability and Interface Negotiation

4.1.1 Reachability Property/Rule Definition and QueryInterface Method Specification

Reachability property is defined as follows: "if a component exposes a given type of interface, that type of interface should be accessible from any interface on the component" ([Sullivan et al. 99], Subsection 2.3.1).

The reachability rule is defined as the requirement by interface negotiation "that a client with any interface should be able to get interface of all types on a component" ([Sullivan et al. 99], Section 3.1).

Reachability is a property or a rule about QueryInterface method of IUnknown interface. There are several definitions of the approach used that are published in the COM literature available on Microsoft's website (http://msdn.microsoft.com). One of them is that QueryInterface "returns a pointer to a specified interface on an object to which a client currently holds an interface pointer" [Microsoft 05f]. Another specification is "return a pointer within this object instance that implements the indicated interface; answer NULL if the receiver does not contain an implementation of the interface" [Micorosft 95].

4.1.2 A Comparison Between Reachability Property/Rule and QueryInterface Method

Reachability is about availability of an interface from a component, while QueryInterface rule is about availability of an interface from an object. A component and an object are different entities. A component is a pre-compilation entity, while an object is a run-time entity. It is the difference between the way reachability is defined and the way QueryInterface specification is defined that causes the conclusion that aggregation may conflict with interface negotiation.

4.1.3 Reachability and Aggregation

Sullivan et al. state that aggregation with hiding at least one inner component's interface conflicts with interface negotiation. The argument for the existence of the conflict is that aggregation with hiding at least one inner component's interface does not conform with the reachability rule ([Sullivan et al. 99], Section 3.1). According to them, by reachability rule, a client to an aggregation should have access to all the interfaces of any of its inner components.

On the other hand, COM specification does not have any particular requirement about the inner component of an aggregation with regard to a client of an instance of the aggregation. With regard to a client, COM specification requirement is about an aggregation object or an instance of an aggregation component.

4.2 Component Identity and Object Identity

Sullivan et al. state that aggregation compromises an inner component's identity ([Sullivan et al. 99], Section 5.2). On the other hand, COM specification states that what is returned by an identity query, i.e., calling QueryInterface to request the IUnknown interface, is only the pointer to the object, which is a run time entity. COM has no specification on a component identity, let alone providing it at run time. An interface, or an interface ID, should be looked as an entity that is independent from a component or a component ID. COM specification concerns only the static association of an interface, or interface identity, to an object identity at a run time.

4.3 Mediator-Based Architecture

Sullivan et al. tried to implement mediator-based architecture using COM's aggregation ([Sullivan et al. 99], Section 3.2). It turned out that COM's aggregation cannot be used for the implementation. In addition, they concluded that there is a conflict between interface negotiation and aggregation. They provided an example that they claimed shows how aggregation makes "the QueryInterface functions on inner components malfunctioning badly" ([Sullivan et al. 99], Section 3.2).

However, there is a flaw in the example. The example assumes that the client obtains a pointer to an inner component's interface, IRawBits, that is hidden. If all COM specifications are conformed, there is no way for a client to obtain a pointer to a hidden interface. So the premise of the example is invalid, which makes the conclusion in that section of the paper invalid.

4.4 Backward Transitivity

One property of interface negotiation is transitive. Some COM publications use backward transitivity to illustrate the property [Microsoft 95].

It is true that in general, backward transitivity is not equivalent with transitivity ([Sullivan et al. 99], Section 5.4). But, provided that symmetry holds, transitivity and backward transitivity are equivalent. The following arguments in two subsections show the equivalence.

4.4.1 Symmetry and Transitivity Imply Backward Transitivity

Suppose whenever a client holds a pointer to the interface called Interface_A, a call by the client to QueryInterface asking for the pointer to the interface called Interface_B is always successful. Suppose also that whenever a client holds a pointer to Interface_B, a call by the client to QueryInterface asking for the pointer to the interface called Interface_C is always successful. Since interface negotiation is a transitive operation, if a client holds a pointer to Interface_A, a call by the client to QueryInterface asking for the pointer to the Interface_C will always be successful.

Since interface negotiation is symmetric, if a client holds a pointer to Interface_C, a call by the client to QueryInterface asking for the pointer to the Interface_B will always be successful. Also, if the client holds a pointer to Interface_B, a call by the client to QueryInterface asking for the pointer to the Interface_A will always be successful. And since interface negotiation is transitive, if a client holds a pointer to Interface_C, a call by the client to QueryInterface asking for the pointer to the Interface_A will always be successful. Hence, if symmetry and transitivity are valid, backward transitivity is also valid.

4.4.2 Symmetry and Backward Transitivity Imply Transitivity

Suppose that whenever a client holds a pointer to Interface_A, a call by the client to QueryInterface asking for the pointer to the Interface_B is always successful. Suppose also that whenever a client holds a pointer to Interface_B, a call by the client to

QueryInterface asking for the pointer to the Interface_C is always successful. Since backward transitive is valid, if a client holds a pointer to Interface_C, a call by the client to QueryInterface asking for the pointer to the Interface_A will always be successful. Since interface negotiation is symmetric, if a client holds a pointer to Interface_A, a call by the client to QueryInterface asking for the pointer to the Interface_C will always be successful. This means that transitivity is valid.

4.5 Legal Component and a Formal Model of Interface Negotiation and Aggregation

A legal component is defined as follows: "a component is legal if QueryInterface functions of all of its interfaces follow the COM rules for interface negotiation" ([Sullivan et al. 99], Section 4.6). But actually they do not precisely follow COM specification in using the term legal component. This is clear as they state that "a key property of legal COM components is that their clients always find the same set of interface types on a component regardless of the interfaces through which queries are made" ([Sullivan et al. 99], Section 4.6). The statement says about "finding the same set of interface types on a component", while COM specification on interface negotiation does not have any requirements about finding the same set of interfaces type on a component. COM specification on interface negotiation requires that the interface negotiation on an object (as an instance of a component) should be static during run time [Microsoft 05f] [Microsoft 05g]. To a client, an instance of aggregation is seen as an integral unit, without any consideration of any inner component of the aggregation [Microsoft 05h].

Sullivan et al. constructed a formal model for interface negotiation and aggregation. The model, along with other things, is used to express theorems about the conflict between aggregation and interface negotiation, as the authors of the paper claimed exists ([Sullivan et al. 99], Section 5.1 and Section 3.1), aggregation compromising inner component identity, and sharing interfaces. The proofs of these arguments (as given in the article) use the premise of a legal component. In the previous paragraph, it has been shown that the definition and the specification of a legal component do not follow the original COM specification correctly. Hence, the proofs of the theorems cannot be used to verify any characteristic of COM.

4.6 A Demonstration Program of Interface Negotiation, Aggregation, and Object Identity

Source code segments for demonstration programs of interface negotiation and aggregation are available from the CD-ROM of Rogerson's "Inside COM" [Rogerson 99]. The source code segments are listed in Appendices C through F. The source code segments were modified to also show the object identity in various places in program execution. Another modification made was in order to log messages into a log file called Log.txt.

The original program had three source code files for aggregation and interface negotiation demonstration. They are Client.cpp, Cmpnt1.cpp, and Cmpnt2.cpp. Client.cpp is for the client implementation. The outer component is implemented using class CA in Cmpnt1.cpp. The inner component is implemented using class CB in Cmpnt2.cpp. Outer component has one interface, which is interface IX. Inner component has one interface, which is interface IY.

The values of the object identities should be the same. The value of an object identity is obtained by calling QueryInterface method asking for the IUnknown interface. The value in this execution is 0x00D51280.

The result, as recorded in the Log.txt file, is depicted in Figure 4.1.

```
An object identity is obtained through a call to
QueryInterface asking for IID_IUnknown.

Get interface IX from Component 1.
Create inner component.
Aggregating; delegate to outer IUnknown.
Get the IY interface from the inner component.
Delegate AddRef.
Succeeded creating component.

Object identity.
After outer component creation    : 00D51280

Get interface IY from IX.
Return inner component's IY interface.
Delegate AddRef.
Succeeded getting interface IY from IX.
Delegate QueryInterface.

Object identity.
After getting inner component's IY: 00D51280

Get interface IX from IY.
Delegate QueryInterface.
Succeeded getting interface IX from IY.

Object identity.
After getting outer component's IX: 00D51280

Delegate Release.
```

Figure 4.1 The Result of Aggregation, Interface Negotiation, and Object Identity Demonstration Program

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

It has been concluded that the conflict between aggregation and interface negotiation asserted by Sullivan et al. [Sullivan et al. 99] is based on the specific definition of the reachability property and rule that is provided by them. It turns out that the Reachability property and rule are not parts of the original COM specification.

COM specification does not have anything specific on component identity at run time. So, the notion of compromising an inner component's identity is not relevant.

COM aggregation cannot be used to implement mediator-based architecture, since this architecture may require access to an inner component's hidden interface.

It has been shown that, if symmetry holds, transitivity and backward transitivity are equivalent. Hence, the transitivity of interface negotiation can be illustrated using an example that shows the backward transitivity of QueryInterface.

The proof of most of the theorems, which are expressed using the formal model devised by Sullivan et al. ([Sullivan et al. 99], Section 4), uses the premise of a legal component. This premise is not part of the original COM specification.

5.2 Future Work

The notion of a conflict between aggregation and interface negotiation may arise out of the misunderstanding caused by an ambiguous definition or specification of interface negotiation, such as the one in the Component Object Model Specification, Draft Version 0.9 [Microsoft 95]. A definition that could clearly describe the functionality of QueryInterface for any kind of component (non-aggregation or aggregation) may avoid such misunderstanding.

REFERENCES

[Deitel and Deitel 94] H. M. Deitel and P. J. Deitel, *C++: How to Program*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[Kindel 05] Charlie Kindel, *The Rules of the Component Object Model*, Microsoft Corporation, URL: http://msdn.microsoft.com/library/en-us/dncomg/html/msdn_therules.asp, date created: unknown, date accessed: June 2005.

[Lippman 96] Stanley B. Lippman, *Inside The C++ Object Model*, Addison-Wesley, Reading, MA, 1996.

[Microsoft 95] Microsoft Corp. and Digital Equipment Corp., *The Component Object Model Specification. Draft Version 0.9*, Microsoft Corporation, 1995.

[Microsoft 00] MicrosoftCorp., *C/C++ Language Reference. Virtual Functions*, Microsoft Corporation, URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccelng/htm/deriv_13.asp, date created: May 11, 2000, date accessed: December 2005.

[Microsoft 03] Microsoft Corp., *The Component Object Model*, Microsoft Corporation, URL: http://support.microsoft.com/default.aspx?scid=kb;en-us;104140, date created: unknown, date accessed: January 2005.

[Microsoft 05a] Microsoft Corp., *COM Class Objects and CLSIDs*, Microsoft Corporation, URL: http://msdn.microsoft.com/library/en-us/com/html/0073acdf-38a8-4f1a-aa26-379456a95fca.asp, date created: unknown, date accessed: June 2005.

[Microsoft 05b] Microsoft Corp., *COM Clients and Servers*, Microsoft Corporation, URL: http://msdn.microsoft.com/library/en-us/com/html/5d1d8613-3087-443d-8547-a767c8ba4959.asp, date created: unknown, date accessed: August 2005.

[Microsoft 05c] Microsoft Corp., *COM Objects and Interfaces*, Microsoft Corporation, URL: http://msdn.microsoft.com/library/en-us/com/html/a3b78086-0f02-4b3f-a856-46bfcf4457f4.asp, date created: unknown, date accessed: August 2005.

[Microsoft 05d] Microsoft Corp., *IUnknown and Interface Inheritance*, Microsoft Corporation, URL: http://msdn.microsoft.com/library/en-us/com/html/c45f0947-

6020-4aa1-9250-561603a46a68.asp, date created: unknown, date accessed: August 2005.

[Microsoft 05e] Microsoft Corp., *The Component Object Model*, Microsoft Corporation, URL: http://msdn.microsoft.com/library/en-us/com/html/f5f66603-466c-496b-be29-89a8ed9361dd.asp, date created: unknown, date accessed: June 2005.

[Microsoft 05f] Microsoft Corp., *IUnknown::QueryInterface*, Microsoft Corporation, URL: http://msdn.microsoft.com/library/en-us/com/html/54d5ff80-18db-43f2-b636-f93ac053146d.asp, date created: unknown, date accessed: August 2005.

[Microsoft 05g] Microsoft Corp., *Rules for Implementing QueryInterface*, Microsoft Corporation, URL: http://msdn.microsoft.com/library/en-us/com/html/6db17ed8-06e4-4bae-bc26-113176cc7e0e.asp, date created: unknown, date accessed: October 2005.

[Microsoft 05h] Microsoft Corp., *Aggregation (COM),* Microsoft Corporation, URL: http://msdn.microsoft.com/library/en-us/com/html/6845b114-8f43-47ad-acdf-b63d6008d221.asp, date created: unknown, date accessed: October 2005.

[Rogerson 97] D. Rogerson, *Inside COM*, Microsoft Press, Redmond, WA, 1997.

[Sullivan et al. 99] K. J. Sullivan, M. Marchukov, and J. Socha, "Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Model", *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, pp. 584-599, July/August 1999.

[Thompson 95] Nigel Thompson, *MFC/COM Objects 4: Aggregation*, Microsoft Corporation, URL: http://msdn.microsoft.com/library/en-us/dncomg/html/msdn_house4.asp, date created: March 20, 1995, date accessed: January 2005.

[Williams and Kindel 94] Sara Williams and Charlie Kindel, *The Component Object Model: A Technical Review*, Microsoft Corporation, URL: http://msdn.microsoft.com/library/en-us/dncomg/html/msdn_comppr.asp, date created: October 1994, date accessed: January 2005.

APPENDICES

APPENDIX A

GLOSSARY

Aggregation	A compound of components consisting of one outer component and at least one inner component.
Backward Transitivity	If A is in relation with B and B is in relation with C, then C is in relation with A [Microsoft 95].
COM	Component Object Model.
Component	A group of interfaces. A unit of instantiation.
Interface	A group of methods.
Interface Negotiation	Calling the QueryInterface method of the IUnknown interface.
Object (COM)	An instance of a component.
Object (C++)	An instance of a class.

APPENDIX B

TRADEMARK INFORMATION

Micorosoft is a registered trademark.

MSDN is a Microsoft Corporation trademark.

APPENDIX C

AGGREGATION - CLIENT CODE

The following aggregation code segments are from Dale Rogerson's "Inside COM" CD-ROM folder \CODE\CHAP08\AGGRGATE which contains the source code for illustration of Chapter 8 "Component Reuse: Containtment and Aggregation" [Rogerson 97]. There are three source code files for the aggregation example. They are: Client.cpp, Cmpnt1.cpp, and Cmpnt2.cpp. Listed below is the modified Client.cpp. It is modified to also demonstrate COM specification on the identity of an object.

```
//
// Client.cpp - client implementation
//
#include <iostream.h>
#include <objbase.h>
#include <string.h>
#include <stdio.h>

#include "Iface.h"

//Modified trace()
//It also puts the msg into Log.txt
void trace(const char* msg)
{
    FILE *fp;

    cout << "Client: \t" << msg << endl ;
    fp = fopen("Log.txt", "at");
    if(!fp)
        return;
    fprintf(fp, "\n");
    fprintf(fp, msg);
    fclose(fp);
}

//
// main function
//
int main()
{
    FILE *fp;

    //Pointer to the outer object, i.e., object identity.
    IX* pObjectID=NULL;
    char str[1024];

    // Initialize COM Library
    CoInitialize(NULL) ;

    //Initialize Log.txt
```

```
fp = fopen("Log.txt", "w+t");
if(!fp)
{
    printf("It failed to create ObjectIdentity.txt\n");
    return -1;
}
fprintf(fp, "An object identity is obtained through a call to\n");
fprintf(fp, "QueryInterface asking for IID_IUnknown.\n");
fclose(fp);

trace("Get interface IX from Component 1.") ;
IX* pIX = NULL ;
HRESULT hr = ::CoCreateInstance(CLSID_Component1,
                                NULL,
                                CLSCTX_INPROC_SERVER,
                                IID_IX,
                                (void**)&pIX) ;
if (SUCCEEDED(hr))
{
    trace("Succeeded creating component.") ;

    hr = pIX->QueryInterface(IID_IUnknown, (void**)&pObjectID) ;
    if (SUCCEEDED(hr))
    {
        sprintf(str, "\nObject identity.");
        trace(str);
        sprintf(str, "After outer component creation    : %p\n",
                pObjectID);
        trace(str);
    }

    pIX->Fx() ;
    trace("Get interface IY from IX.") ;
    IY* pIY = NULL ;
    hr = pIX->QueryInterface(IID_IY, (void**)&pIY) ;
    if (SUCCEEDED(hr))
    {
        trace("Succeeded getting interface IY from IX.") ;
        pIY->Fy() ;

        //Show the identity of the object.
        hr = pIY->QueryInterface(IID_IUnknown, (void**)&pObjectID) ;
        if (SUCCEEDED(hr))
        {
            sprintf(str, "\nObject identity.");
            trace(str);
            sprintf(str, "After getting inner component's IY: %p\n",
                    pObjectID);
            trace(str);
        }

        trace("Get interface IX from IY.") ;
        IX* pIX2 = NULL ;
        hr = pIY->QueryInterface(IID_IX, (void**)&pIX2);
        if (SUCCEEDED(hr))
        {
            trace("Succeeded getting interface IX from IY.") ;

            //Show the identity of the object.
            hr = pIX2->QueryInterface(IID_IUnknown, (void**)&pObjectID) ;
            if (SUCCEEDED(hr))
            {
                sprintf(str, "\nObject identity.");
```

```
                    trace(str);
                    sprintf(str, "After getting outer component's IX: %p\n",
                            pObjectID);
                    trace(str);
                }

                pIX2->Release() ;
            }
            else
            {
                trace("Error! Should have gotten interface IX.") ;
            }

            pIY->Release() ;
        }
        else
        {
            trace("Could not get interface IY.") ;
        }

        pIX->Release() ;
    }
    else
    {
        cout << "Could not create component: " << hex << hr << endl ;
    }

    // Uninitialize COM Library
    CoUninitialize() ;

    fclose(fp);

    return 0 ;
}
```

APPENDIX D

AGGREGATION - OUTER COMPONENT CODE

The following aggregation code segments are from Dale Rogerson's "Inside COM" CD-ROM folder \CODE\CHAP08\AGGRGATE which contains the source code for illustration of Chapter 8 "Component Reuse: Containtment and Aggregation" [Rogerson 97]. There are three source code files for the aggregation example. They are: Client.cpp, Cmpnt1.cpp, and Cmpnt2.cpp. Listed below is Cmpnt1.cpp.

```
//
// Cmpnt1.cpp - Component 1
//
//   Interesting bits of code marked with @N.
//
#include <iostream.h>
#include <objbase.h>
#include <stdio.h>

#include "Iface.h"
#include "Registry.h"

// Trace function
//void trace(const char* msg) { cout << "Component 1:\t" << msg << endl ;}
//Modified trace()
//It also puts the msg into Log.txt
void trace(const char* msg)
{
   FILE *fp;

   cout << "Component 1:\t" << msg << endl ;
   fp = fopen("Log.txt", "at");
   if(!fp)
      return;
   fprintf(fp, "\n");
   fprintf(fp, msg);
   fclose(fp);
}

///////////////////////////////////////////////////////////
//
// Global variables
//

// Static variables
static HMODULE g_hModule = NULL ;   // DLL module handle
static long g_cComponents = 0 ;     // Count of active components
static long g_cServerLocks = 0 ;    // Count of locks

// Friendly name of component
const char g_szFriendlyName[]
```

```
    = "Inside COM, Chapter 8 Example 2, Component 1" ;

// Version-independent ProgID
const char g_szVerIndProgID[] = "InsideCOM.Chap08.Ex2.Cmpnt1" ;

// ProgID
const char g_szProgID[] = "InsideCOM.Chap08.Ex2.Cmpnt1.1" ;

///////////////////////////////////////////////////////////
//
// Component A
//
class CA : public IX
        // public IY @N
{
public:
    // IUnknown
    virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv) ;
    virtual ULONG   __stdcall AddRef() ;
    virtual ULONG   __stdcall Release() ;

    // Interface IX
    virtual void __stdcall Fx() { cout << "Fx" << endl ;}

    /* @N Component1 aggregates instead of implementing interface IY.
    // Interface IY
    virtual void __stdcall Fy() { m_pIY->Fy() ;}
    */

    // Constructor
    CA() ;

    // Destructor
    ~CA() ;

    // Initialization function called by the class factory
    // to create the contained component.
    HRESULT __stdcall Init() ;  // @N

private:
    // Reference count
    long m_cRef ;

    // Pointer to the aggregated component's IY interface
    // (We do not have to retain an IY pointer. However, we
    // can use it in QueryInterface.)
    IY* m_pIY ;                 // @N

    // Pointer to inner component's IUnknown
    IUnknown* m_pUnknownInner ; // @N
} ;


//
// Constructor
//
CA::CA()
: m_cRef(1),
  m_pUnknownInner(NULL) //@N
{
    ::InterlockedIncrement(&g_cComponents) ;
}
```

```
//
// Destructor
//
CA::~CA()
{
    ::InterlockedDecrement(&g_cComponents) ;
    trace("Destroy self.") ;

    // Prevent recursive destruction on next AddRef/Release pair.
    m_cRef = 1 ;

    // Counter the pUnknownOuter->Release in the Init method.
    IUnknown* pUnknownOuter = this ;
    pUnknownOuter->AddRef() ;

    // Properly release the pointer; there might be per-interface
    // reference counts.
    m_pIY->Release() ;

    // Release contained component.
    if (m_pUnknownInner != NULL)   // @N
    {
        m_pUnknownInner->Release() ;
    }
}

// Initialize the component by creating the contained component.
HRESULT __stdcall CA::Init()
{
    // Get the pointer to the outer unknown.
    // Since this component is not aggregated, the outer unknown
    // is the same as the this pointer.
    IUnknown* pUnknownOuter = this ;

    trace("Create inner component.") ;
    HRESULT hr =
        ::CoCreateInstance(CLSID_Component2,
                           pUnknownOuter, // Outer component's IUnknown @N
                           CLSCTX_INPROC_SERVER,
                           IID_IUnknown,  // IUnknown when aggregating  @N
                           (void**)&m_pUnknownInner) ;
    if (FAILED(hr))
    {
        trace("Could not create contained component.") ;
        return E_FAIL ;
    }

    // This call will increment the reference count on the outer component.
    trace("Get the IY interface from the inner component.") ;
    hr = m_pUnknownInner->QueryInterface(IID_IY, (void**)&m_pIY) ; //@N
    if (FAILED(hr))
    {
        trace("Inner component does not support interface IY.") ;
        m_pUnknownInner->Release() ;
        return E_FAIL ;
    }

    // We need to release the reference count added to the
    // outer component in the above call.  So call Release
    // on the pointer you passed to CoCreateInstance.
    pUnknownOuter->Release() ; //@N
    return S_OK ;
}
```

```
//
// IUnknown implementation
//
HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
    if (iid == IID_IUnknown)
    {
        *ppv = static_cast<IUnknown*>(this) ;
    }
    else if (iid == IID_IX)
    {
        *ppv = static_cast<IX*>(this) ;
    }
    else if (iid == IID_IY)
    {
        trace("Return inner component's IY interface.") ;
#if 1
        // You can query for the interface.
        return m_pUnknownInner->QueryInterface(iid,ppv) ; //@N
#else
        // Or you can return a cached pointer.
        *ppv = m_pIY ;   //@N
        // Fall through so it will get AddRef'ed
#endif
    }
    else
    {
        *ppv = NULL ;
        return E_NOINTERFACE ;
    }
    reinterpret_cast<IUnknown*>(*ppv)->AddRef() ;
    return S_OK ;
}

ULONG __stdcall CA::AddRef()
{
    return ::InterlockedIncrement(&m_cRef) ;
}

ULONG __stdcall CA::Release()
{
    if (::InterlockedDecrement(&m_cRef) == 0)
    {
        delete this ;
        return 0 ;
    }
    return m_cRef ;
}



///////////////////////////////////////////////////////////
//
// Class factory
//
class CFactory : public IClassFactory
{
public:
    // IUnknown
    virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv) ;
    virtual ULONG   __stdcall AddRef() ;
    virtual ULONG   __stdcall Release() ;
    // Interface IClassFactory
    virtual HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,
                                             const IID& iid,
                                             void** ppv) ;
    virtual HRESULT __stdcall LockServer(BOOL bLock) ;

    // Constructor
    CFactory() : m_cRef(1) {}

    // Destructor
    ~CFactory() {}

private:
    long m_cRef ;
} ;

//
// Class factory IUnknown implementation
//
HRESULT __stdcall CFactory::QueryInterface(REFIID iid, void** ppv)
{
    IUnknown* pI ;
    if ((iid == IID_IUnknown) || (iid == IID_IClassFactory))
    {
        pI = static_cast<IClassFactory*>(this) ;
    }
    else
    {
        *ppv = NULL ;
        return E_NOINTERFACE ;
    }
    pI->AddRef() ;
    *ppv = pI ;
    return S_OK ;
}

ULONG __stdcall CFactory::AddRef()
{
    return ::InterlockedIncrement(&m_cRef) ;
}

ULONG __stdcall CFactory::Release()
{
    if (::InterlockedDecrement(&m_cRef) == 0)
    {
        delete this ;
        return 0 ;
    }
    return m_cRef ;
}

//
// IClassFactory implementation
//
HRESULT __stdcall CFactory::CreateInstance(IUnknown* pUnknownOuter,
                                           const IID& iid,
                                           void** ppv)
{
    // Cannot aggregate
    if (pUnknownOuter != NULL)
    {
        return CLASS_E_NOAGGREGATION ;
    }

    // Create component.
    CA* pA = new CA ;
    if (pA == NULL)
    {
        return E_OUTOFMEMORY ;
    }

    // Initialize the component. @N
    HRESULT hr = pA->Init() ;
    if (FAILED(hr))
    {
        // Initialization failed. Delete component.
        pA->Release() ;
        return hr ;
    }

    // Get the requested interface.
    hr = pA->QueryInterface(iid, ppv) ;
    pA->Release() ;
    return hr ;
}

// LockServer
HRESULT __stdcall CFactory::LockServer(BOOL bLock)
{
    if (bLock)
    {
        ::InterlockedIncrement(&g_cServerLocks) ;
    }
    else
    {
        ::InterlockedDecrement(&g_cServerLocks) ;
    }
    return S_OK ;
}


///////////////////////////////////////////////////////////
//
// Exported functions
//

STDAPI DllCanUnloadNow()
{
    if ((g_cComponents == 0) && (g_cServerLocks == 0))
    {
        return S_OK ;
    }
    else
    {
        return S_FALSE ;
    }
}

//
// Get class factory.
//
STDAPI DllGetClassObject(const CLSID& clsid,
                         const IID& iid,
                         void** ppv)
{
    // Can we create this component?
    if (clsid != CLSID_Component1)
    {
        return CLASS_E_CLASSNOTAVAILABLE ;
    }

    // Create class factory.
    CFactory* pFactory = new CFactory ; // No Addref in constructor
    if (pFactory == NULL)
    {
        return E_OUTOFMEMORY ;
    }

    // Get requested interface.
    HRESULT hr = pFactory->QueryInterface(iid, ppv) ;
    pFactory->Release() ;

    return hr ;
}

//
// Server registration
//
STDAPI DllRegisterServer()
{
    return RegisterServer(g_hModule,
                          CLSID_Component1,
                          g_szFriendlyName,
                          g_szVerIndProgID,
                          g_szProgID) ;
}


STDAPI DllUnregisterServer()
{
    return UnregisterServer(CLSID_Component1,
                            g_szVerIndProgID,
                            g_szProgID) ;
}

///////////////////////////////////////////////////////////
//
// DLL module information
//
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD dwReason,
                      void* lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        g_hModule = hModule ;
    }
    return TRUE ;
}
```

APPENDIX E

AGGREGATION - INNER COMPONENT CODE

The following aggregation code segments are from Dale Rogerson's "Inside COM" CD-ROM folder \CODE\CHAP08\AGGRGATE which contains the source code for illustration of Chapter 8 "Component Reuse: Containtment and Aggregation" [Rogerson 97]. There are three source code files for the aggregation example. They are: Client.cpp, Cmpnt1.cpp, and Cmpnt2.cpp. Listed below is Cmpnt2.cpp.

```
//
// Cmpnt2.cpp - Component 2
//   Note the changes in the class factory marked with @N.
//
#include <iostream.h>
#include <objbase.h>
#include <stdio.h>

#include "Iface.h"
#include "Registry.h"

//void trace(const char* msg) { cout << "Component 2:\t" << msg << endl ;}
//Modified trace()
//It also puts the msg into Log.txt
void trace(const char* msg)
{
   FILE *fp;

   cout << "Component 2:\t" << msg << endl ;
   fp = fopen("Log.txt", "at");
   if(!fp)
      return;
   fprintf(fp, "\n");
   fprintf(fp, msg);
   fclose(fp);
}

///////////////////////////////////////////////////////////
//
// Global variables
//

// Static variables
static HMODULE g_hModule = NULL ;   // DLL module handle
static long g_cComponents = 0 ;     // Count of active components
static long g_cServerLocks = 0 ;    // Count of locks

// Friendly name of component
const char g_szFriendlyName[]
    = "Inside COM, Chapter 8 Example 2, Component 2" ;
```

```
// Version-independent ProgID
const char g_szVerIndProgID[] = "InsideCOM.Chap08.Ex2.Cmpnt2" ;

// ProgID
const char g_szProgID[] = "InsideCOM.Chap08.Ex2.Cmpnt2.1" ;

///////////////////////////////////////////////////////////
//
// Nondelegating IUnknown interface  @N
//
struct INondelegatingUnknown
{
    virtual HRESULT __stdcall
        NondelegatingQueryInterface(const IID&, void**) = 0 ;
    virtual ULONG __stdcall NondelegatingAddRef() = 0 ;
    virtual ULONG __stdcall NondelegatingRelease() = 0 ;
} ;

///////////////////////////////////////////////////////////
//
// Component
//
class CB : public IY,
           public INondelegatingUnknown
{
public:
    // Delegating IUnknown
    virtual HRESULT __stdcall
        QueryInterface(const IID& iid, void** ppv)
    {
        trace("Delegate QueryInterface.") ;
        return m_pUnknownOuter->QueryInterface(iid, ppv) ;
    }

    virtual ULONG __stdcall AddRef()
    {
        trace("Delegate AddRef.") ;
        return m_pUnknownOuter->AddRef() ;
    }

    virtual ULONG __stdcall Release()
    {
        trace("Delegate Release.") ;
        return m_pUnknownOuter->Release() ;
    }

    // Nondelegating IUnknown
    virtual HRESULT __stdcall
        NondelegatingQueryInterface(const IID& iid, void** ppv) ;
    virtual ULONG   __stdcall NondelegatingAddRef() ;
    virtual ULONG   __stdcall NondelegatingRelease() ;

    // Interface IY
    virtual void __stdcall Fy() { cout << "Fy" << endl ;}

    // Constructor
    CB(IUnknown* m_pUnknownOuter) ;

    // Destructor
    ~CB() ;

private:
    long m_cRef ;
    IUnknown* m_pUnknownOuter ;
} ;

//
// IUnknown implementation
//
HRESULT __stdcall CB::NondelegatingQueryInterface(const IID& iid,
                                                  void** ppv)
{
    if (iid == IID_IUnknown)
    {
        // !!! CAST IS VERY IMPORTANT !!!
        *ppv = static_cast<INondelegatingUnknown*>(this) ;  // @N
    }
    else if (iid == IID_IY)
    {
        *ppv = static_cast<IY*>(this) ;
    }
    else
    {
        *ppv = NULL ;
        return E_NOINTERFACE ;
    }
    reinterpret_cast<IUnknown*>(*ppv)->AddRef() ;
    return S_OK ;
}

ULONG __stdcall CB::NondelegatingAddRef()
{
    return ::InterlockedIncrement(&m_cRef) ;
}

ULONG __stdcall CB::NondelegatingRelease()
{
    if (::InterlockedDecrement(&m_cRef) == 0)
    {
        delete this ;
        return 0 ;
    }
    return m_cRef ;
}

//
// Constructor
//
CB::CB(IUnknown* pUnknownOuter)
: m_cRef(1)
{
    ::InterlockedIncrement(&g_cComponents) ;

    if (pUnknownOuter == NULL)
    {
        trace("Not aggregating; delegate to nondelegating IUnknown.") ;
        m_pUnknownOuter = reinterpret_cast<IUnknown*>
                          (static_cast<INondelegatingUnknown*>
                          (this)) ;
    }
    else
    {
        trace("Aggregating; delegate to outer IUnknown.") ;
        m_pUnknownOuter = pUnknownOuter ;
    }
```

```
}

//
// Destructor
//
CB::~CB()
{
    ::InterlockedDecrement(&g_cComponents) ;
    trace("Destroy self.") ;
}

///////////////////////////////////////////////////////////
//
// Class factory
//
class CFactory : public IClassFactory
{
public:
    // IUnknown
    virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv) ;
    virtual ULONG   __stdcall AddRef() ;
    virtual ULONG   __stdcall Release() ;

    // Interface IClassFactory
    virtual HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,
                                             const IID& iid,
                                             void** ppv) ;
    virtual HRESULT __stdcall LockServer(BOOL bLock) ;

    // Constructor
    CFactory() : m_cRef(1) {}

    // Destructor
    ~CFactory() {}

private:
    long m_cRef ;
} ;

//
// Class factory IUnknown implementation
//
HRESULT __stdcall CFactory::QueryInterface(const IID& iid, void** ppv)
{
    if ((iid == IID_IUnknown) || (iid == IID_IClassFactory))
    {
        *ppv = static_cast<IClassFactory*>(this) ;
    }
    else
    {
        *ppv = NULL ;
        return E_NOINTERFACE ;
    }
    reinterpret_cast<IUnknown*>(*ppv)->AddRef() ;
    return S_OK ;
}

ULONG __stdcall CFactory::AddRef()
{
    return ::InterlockedIncrement(&m_cRef) ;
}

ULONG __stdcall CFactory::Release()
```

```
{
    if (::InterlockedDecrement(&m_cRef) == 0)
    {
        delete this ;
        return 0 ;
    }
    return m_cRef ;
}

//
// IClassFactory implementation
//
HRESULT __stdcall CFactory::CreateInstance(IUnknown* pUnknownOuter,
                                           const IID& iid,
                                           void** ppv)
{
    // Aggregate only if the requested iid is IID_IUnknown.
    if ((pUnknownOuter != NULL) && (iid != IID_IUnknown)) //@N
    {
        return CLASS_E_NOAGGREGATION ;
    }

    // Create component.
    CB* pB = new CB(pUnknownOuter) ; // @N
    if (pB == NULL)
    {
        return E_OUTOFMEMORY ;
    }

    // Get the requested interface.
    HRESULT hr = pB->NondelegatingQueryInterface(iid, ppv) ; //@N
    pB->NondelegatingRelease() ;
    return hr ;
}

// LockServer
HRESULT __stdcall CFactory::LockServer(BOOL bLock)
{
    if (bLock)
    {
        ::InterlockedIncrement(&g_cServerLocks) ;
    }
    else
    {
        ::InterlockedDecrement(&g_cServerLocks) ;
    }
    return S_OK ;
}


///////////////////////////////////////////////////////////
//
// Exported functions
//

STDAPI DllCanUnloadNow()
{
    if ((g_cComponents == 0) && (g_cServerLocks == 0))
    {
        return S_OK ;
    }
    else
    {
        return S_FALSE ;
    }
}

//
// Get class factory.
//
STDAPI DllGetClassObject(const CLSID& clsid,
                         const IID& iid,
                         void** ppv)
{
    // Can we create this component?
    if (clsid != CLSID_Component2)
    {
        return CLASS_E_CLASSNOTAVAILABLE ;
    }

    // Create class factory.
    CFactory* pFactory = new CFactory ; // No Addref in constructor
    if (pFactory == NULL)
    {
        return E_OUTOFMEMORY ;
    }

    // Get requested interface.
    HRESULT hr = pFactory->QueryInterface(iid, ppv) ;
    pFactory->Release() ;

    return hr ;
}

//
// Server registration
//
STDAPI DllRegisterServer()
{
    return RegisterServer(g_hModule,
                          CLSID_Component2,
                          g_szFriendlyName,
                          g_szVerIndProgID,
                          g_szProgID) ;
}


STDAPI DllUnregisterServer()
{
    return UnregisterServer(CLSID_Component2,
                            g_szVerIndProgID,
                            g_szProgID) ;
}

///////////////////////////////////////////////////////////
//
// DLL module information
//
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD dwReason,
                      void* lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        g_hModule = hModule ;
    }
    return TRUE ;
}
```

APPENDIX F

AGGREGATION - OTHER SOURCE FILES

The following aggregation code segments are from Dale Rogerson's "Inside COM" CD-ROM folder \CODE\CHAP08\AGGRGATE which contains the source code for illustration of Chapter 8 "Component Reuse: Containtment and Aggregation" [Rogerson 97]. Listed below are Iface.h and Guids.cpp. The header file Iface.h is included by the three aggregation source codes listed in the previous appendices. Iface.h contains the declaration of interface and component GUIDs. Guids.cpp constains the definition of interface and component GUIDs.

\CODE\CHAP08\AGGRGATE\Iface.h:

```
//
// Interfaces.h - Shared header
//
interface IX : IUnknown
{
    virtual void __stdcall Fx() = 0 ;
};


interface IY : IUnknown
{
    virtual void __stdcall Fy() = 0 ;
};


interface IZ : IUnknown
{
    virtual void __stdcall Fz() = 0 ;
};

//
// Declaration of GUIDs for interfaces and components
//
extern "C" const IID IID_IX ;
extern "C" const IID IID_IY ;
extern "C" const IID IID_IZ ;

extern "C" const CLSID CLSID_Component1 ;
extern "C" const CLSID CLSID_Component2 ;
```

\CODE\CHAP08\AGGRGATE\Guids.cpp:

```
//
// GUIDs.cpp -
//   All GUIDs are defined in this file.
//
#include <objbase.h>

// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
extern "C" const IID IID_IX =
    {0x32bb8320, 0xb41b, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

// {32bb8321-b41b-11cf-a6bb-0080c7b2d682}
extern "C" const IID IID_IY =
    {0x32bb8321, 0xb41b, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

// {32bb8322-b41b-11cf-a6bb-0080c7b2d682}
extern "C" const IID IID_IZ =
    {0x32bb8322, 0xb41b, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

// {0c092c24-882c-11cf-a6bb-0080c7b2d682}
extern "C" const CLSID CLSID_Component1 =
    {0x0c092c24, 0x882c, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;

// {0c092c25-882c-11cf-a6bb-0080c7b2d682}
extern "C" const CLSID CLSID_Component2 =
    {0x0c092c25, 0x882c, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;
```

VITA

Arif Muljadi

Candidate for the Degree of

Master of Science

Thesis: A DETAILED ANALYSIS OF THE COMPONENT OBJECT MODEL

Major Field: Computer Science

Biographical:

Personal Data: Born in Malang, Jawa Timur, Indonesia, on November 16, 1963, son of Ibnu Sutopo and Aini.

Education: Received the Bachelor of Mathematics degree from Padjadjaran University in August 1989; completed the requirements for the degree of Master of Science in Computer Science at the Computer Science Department of Oklahoma State University in May 2006.

Experience: Worked with Industri Pesawat Terbang Nusantara, Indonesia, as a programmer from August 1989 to July 1992. Employed by the Computer Science Department as a Graduate Teaching Assistant from January 1995 to December 2000. Worked with the REN Corporation, Stillwater, Oklahoma, as a systems analyst from January 2001 to present.

Name: Arif Muljadi Date of Degree: May 2006

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: A DETAILED ANALYSIS OF THE COMPONENT OBJECT MODEL

Pages in Study: 49 Candidate for the Degree of Master of Science

Major Field: Computer Science

Scope and Method of Study: Microsoft's Component Object Model (COM) is a system for reusing software artifacts at the post-compilation level. Several problems have been raised about the Component Object Model [Sullivan et al. 99]. The problems are about a conflict between interface negotiation and aggregation, a conclusion about transitivity, and the identity of the inner components. However, the conflict between interface negotiation and aggregation exists only if a questionable definition is used. The formal model used to depict the conflict is also inadequate.

Findings and Conclusion: This thesis concerns an investigation of COM criticisms. A set of programs is used to demonstrate the consistency of the Component Object Model rules regarding interface negotiation and aggregation.

ADVISOR'S APPROVAL: Dr. M. H. Samadzadeh