FACHHOCHSCHULE
TECHNIKUM WIEN
ELEKTRONIK

Diploma Thesis

Universal Dataprovider For Heterogen Systems Based On COM+ And XML

Written At The:

TECHNIKUM WIEN

FACHHOCHSCHULSTUDIENGANG ELEKTRONIK

Written By:

Stefan Domnanovits, Roterdstr 12/3/5 A-1160 Wien, 9710011086

Coaches:

Company: DI Mario Simandl, ace Neue Informationstechnologien GmbH

Academy: DI Dr Robert Pucher

Date: 28.05.2001

Problem

The task was to develop an application server capable of providing information to different clients. This server must be able to retrieve data from many different data sources with different kind and structure. Additionally the results have to be linked together to express related data together in a consistent form. The clients have to be able to use this server even if they are written in a language different as that from the server. Also the client access must be handled in transactions to protect the multiple data sources from inconsistency and to secure the data access from multiple clients accessing the server at the same time.

The server should also be easy to administrate and must have the ability to be easily upgraded for future data sources.

Abstract

This paper develops a server application as a possible solution for enterprises that want to retrieve information from multiple data stores. The server is developed using XML as data transportation format and COM+ as the runtime environment. This gives the server the advantages of usability for many programming languages, transaction safety across multiple data platforms, and scalability. The server is also extensible because of using plugins for data access and it is easy to administrate due the fact that the configuration information is displayed and can be edited in a clear form inside the Microsoft Management Console, the standard configuration environment under Windows 2000.

Diese Diplomarbeit beschreibt die Entwicklung eines Servers für Unternehmen welche Informationen aus verschieden Datenspeichern verwerten wollen. Hierzu wir XML als Datenformat für den Transport verwendet und die COM+ Laufzeitumgebung. Dies gibt dem Server eine Reihe von Vorteilen: Verwendbarkeit durch zahlreiche Programmiersprachen, Transaktionssicherheit selbst über mehrere Datenplattformen hinweg. Der Server ist außerdem erweiterbar durch die Verwendung von Plugins für den Datenzugriff und außerdem leicht zu administrieren, da die Konfigurationsdaten in einer übersichtlichen Form innerhalb der Microsoft Management Konsole, der Standard Konfigurationsumgebung unter Windows, dargestellt und bearbeitet werden können.

Used Abbreviations

ADO	Active Data Object
API	Application Programming Interface
ATL	Active Template Library
CLSID	Class Identifier
COM	Component Object Model
DBMS	Database Management System
DCOM	Distributed COM
DLL	Dynamic Link Library
FSC	Fabasoft Components
GUID	Global Unique Identifier
HTML	Hypertext Markup Language
IID	Interface ID
IDL	Interface Definition Language
IPC	Inter-Process Communication
MMC	Microsoft Management Console
MSDAC	Microsoft Data Access Components
MSMQ	Microsoft Message Queue
MSVS	Microsoft Visual Studio
MTA	Multi-Threaded Apartment
MTS	Microsoft Transaction Service
RPC	Remote Procedure Call
SCM	Service Control Manager
SDK	Software Development Kit
SQL	Structured Query Language
STA	Single Threaded Apartment
STL	Standard Template Library
TNA	Thread Neutral Apartment
UML	Unified Modeling Language
vptr	Virtual Function Pointer
vtbl	Virtual Function Table
XML	Extensible Markup Language

Table Of Contents

1 INTRODUCTION

1.1 *INTRODUCTION*

This paper tries to find a solution for a problem that is common in bigger enterprises. There are multiple departments with information systems and different data. To use this data most effectively the enterprise has to link this different data sources and be able to find and retrieve information in an effective way.

The XML data provider developed with this paper is a possible answer. This XML Server uses, as the name implies, XML as data transportation format. XML is one of the most flexible data representation formats and has already begun to find its way into most bigger software products[1]. The runtime environment of the server is COM+. The server makes extensive use of COM+ features like distributed transactions supported by the Microsoft Transaction Server, Object Pooling or Just In Time activation.

This diploma thesis develops the server by first describing the basic technology issues associated with this project, like object orientation, COM and finally COM+. Next the XML Server implementation is described. This includes the server itself but also the administration tool that had to be build, to support administrators in their task.

Whenever possible the usage of C++ is avoided. Instead UML diagrams are used. This enhances readiness of program capabilities because UML is able to display often the same information but in a graphical view. This enables developers who are not that experienced in C++ to quickly gather an overview of the displayed functionality.

1.2 *HOW TO READ THIS DOCUMENT*

1.2.1 Source Code

In code examples, a proportional font is used for identifiers. For Example:

```
#include <iostream>
int main()
{
        std::cout << “Hello, new world!\n”;
}
```

At first glance, this representation style seem “unnatural” to programmers accustomed to seeing code in constant-width fonts. However, proportional-width fonts are generally

[1] The W3C Consortium has already defined a communication standard based on XML, called SOAP. All new server Products from Microsoft have XML interfaces, like the SQL Server 2000 or the Microsoft Biztalk Server 2000. Also all other major software companies equip their new software with new XML features, including the Oracle Server Database or SAP R/3.

regarded as better than constant-width fonts for presentation of text. Using this font also allows presenting code with fewer illogical line breaks.

1.2.2 UML Diagrams

Unified Modeling Language (UML) diagrams are a standard in describing objects and their relationship between each other. These diagrams are also used in describing object activities on a time base and presenting business workflows.

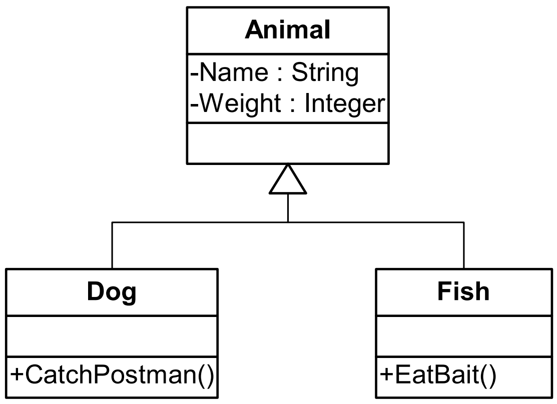


Figure 1.2.2-1 Basic UML Example

There are different ways of interpreting each symbol. This diploma thesis works with the standard described in [2].

1.2.3 XML Representation

XML Data is written in a constant-width font, where the XML data is written in bold letters. This helps the reader to separate the XML tag names from the actual XML content.

```
<Animal>
  <Dog>
    <name>Charlie</name>
    <weight>100</weight>
  </Dog>
  <Fish>
    <name>Susi</name>
    <weight>5</weight>
  </Fish>
</Animal>
```

2 THEORY

2.1 DISTRIBUTED APPLICATION DESIGN

2.1.1 Multilayer Design

Multilayer design means that the whole application is not programmed as one whole block of functionality. Instead the application is split into different service layers each one concentrating on a specific task in the whole application

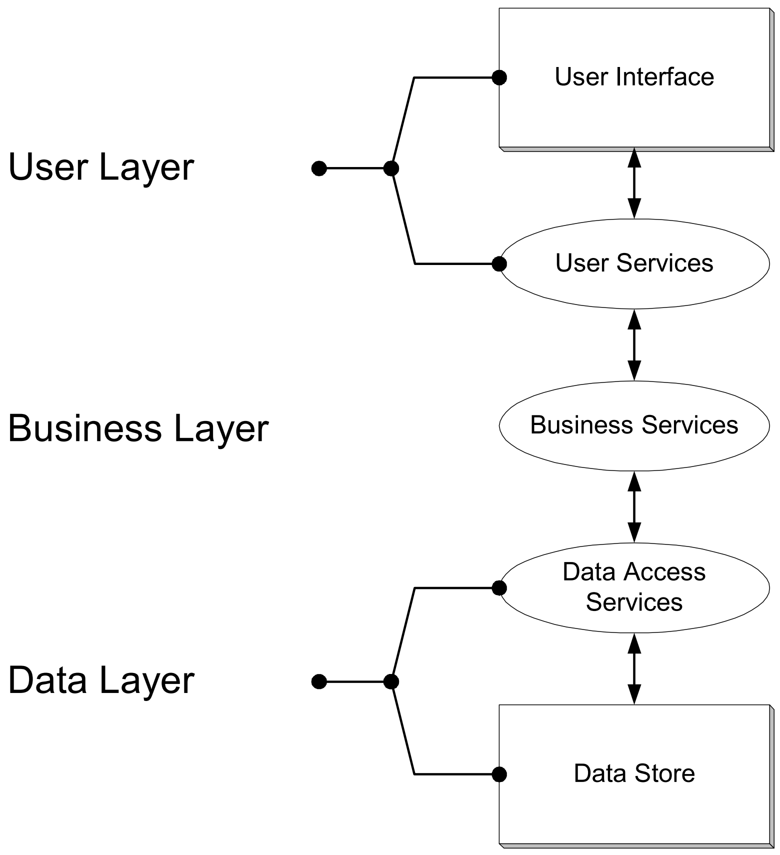


Figure 2.1.1-1 Multitier Application

These services can be distributed across both physical and functional boundaries to support the needs of the solution. The possible breadth of application of each service allows for parallel development, better use of technology, easier maintenance, and increased flexibility in the distribution and deployment of each logical service in the solution.

2.1.2 Benefits of Multilayer Designs

Using a Multitier design approach can provide several benefits to the developer process. These benefits include reusability, flexibility of distribution, and parallelism in the design effort.

- **Reusability** Traditionally, applications have been developed independently, each project focusing exclusively on its own needs. When a business rule or information view changes, each element of the application would have to be modified to work with the new development. Using a services based, modular approach, developers can design, implement and change systems independently of the other service layers.
- **Flexibility of distribution** The service-based approach provides maximum flexibility by allowing developers to deploy logic where it best meets the performance and usage requirements of the application. This approach also supports greater interoperability. With current technologies, services are provided with a transparency of location, enabling them to be distributed in the best configuration for the particular business solution, whether that means all on one client computer or across multiple computers around the world.
- **Parallelism** One of the major advantages offered by the service-based approach is the ability to do more than one development task at a time. Because the application model defines an application structure of five distinct pieces, each of these pieces can be worked on in parallel. In the past, with monolithic implementations, development was a serial process because no distinction was made among different types of functionality.

The abilities of COM described in the later chapters make it an ideal technology for programming applications in a multilayer design. COM shares all major advantages of this development approach.

The XML Server described in this document itself is programmed as part of the data access services, shown in Figure 2.1.1-1. Providing data in XML form to higher application layers.

2.2 XML

Data may have very different structure and form between different data stores and applications. For example the “classical” relational database like the MS SQL Server or the Oracle Database have its data stored in tables connected with foreign keys. But other sources like files located on a hard disk or the new Active Directory from Microsoft have their data stored in a hierarchy like a tree.

It is difficult to find a form, which can represent all these kinds of different data structures. The XML standard is the most pleasant way to represent many different types of data. It is especially an ideal form if it is used for asynchrony transportation over the Internet. Because modern browser like the Internet Explorer are able to display this data with a descriptive XML Style Sheet like any other HTML document. Here are two examples how a XML file may look.

```
<personlist>
  <person id=1>
  <name>Domnanovits</name>
  </person>
  <person id=2>
  <name>Simandl</name>
  <person>
</personlist>
```

```
<message>
  <body>
    Hi Mario, do we meet
  each other tomorrow?
  </body>
  <siganture>
    Stefan
  </signature>
</message>
```

As anyone can see in the XML lines above, XML does not contain any style format information like the fonts to use or to display the data in a table. The task of formatting the data is done in an XML Style Sheet. This allows a clear separation of data and its representation. This is a great advantage to web applications without using XML because there the developer has to build and to format the data into one big HTML document. This takes much more effort than two separate task of retrieving and displaying information.

An XML Style Sheet would display the first XML file as a list or table. The second one would be displayed in a more message like way.

As one can see XML is well structured and very flexible in its data storage. The data is divided into different tags. Each tag starts with a tag name in peaked parenthesis and ends with a tag of the same name containing a slash in front of the name. The actual data is stored between the start and the end tag. Also each tag may contain itself an infinite number of additional sub tags. Additionally it is possible for each tag to be more classified with supplementing attributes to produce more specification about one tag.

One of the major goals of XML Server was to retrieve and transport content and knowledge information, that means presenting larger text data and document attributes. XML is ideal for that task, considering the fact that HTML (the content language of the Internet) is just a sub standard of XML.

2.3 OBJECT ORIENTED PROGRAMMING

Programming in an object-oriented language is not that difficult. The real problem is that it takes a while to understand the advantages in using the new object features. Tom Hadfield gets the point in saying: "Object languages *allow* advantages but don't *provide* them"[2].

2.3.1 Objects and Classes

In all object-oriented applications the basic element of information and action is an object. An object contains of data elements and operations. The data is stored in member variables and the operations are performed in methods. Figure 2.3.1-1 shows an example of such a simple class.

[2] Taken from [16], Page 22

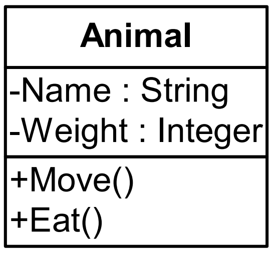


Figure 2.3.1-1 Animal Object

This animal class is described with two data members, a string containing the name of the animal and the weight as a number. The animal is able to perform two actions, to move and to eat. The programmer is now responsible to perform these actions. This means he has to produce code that moves the animal and to perform the animal eating mechanism.

Now the user may create different instances of animals and is able to move them around in its virtual world and to let them eat and become fat.

2.3.2 Inheritance

Unfortunately there are many different animals and a developer would be very hard pressed to implement a moving mechanism for all kind of spiders, animal, fishes etc. On the other hand many creatures share the same moving or eating mechanism. Inheritance is the way to group these bunch of abilities and to add new or specific ones.

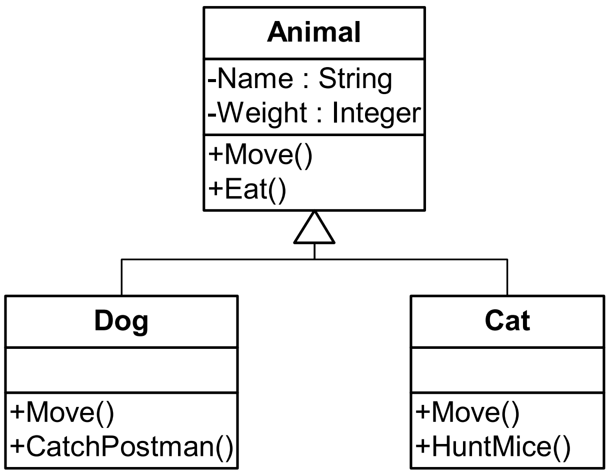


Figure 2.3.2-1 Object Inheritance

In the figure above the animal object is now closer specified with two additional classes. The software developer is now able to write a moving method for all kinds of dogs or all kind of cats. This mechanism may be continued in all directions.

2.3.3 Polymorphism

Another technique is added to objects to make lifer easier for software developers. Until now the full program capability is spread over different objects. This is convenient if every object knows exactly each type of object and how to handle it. Consider the following example: A *Cage* object is added. This *Cage* may contain any type of animal. But what happens if somebody wants to feed the object in the cage.

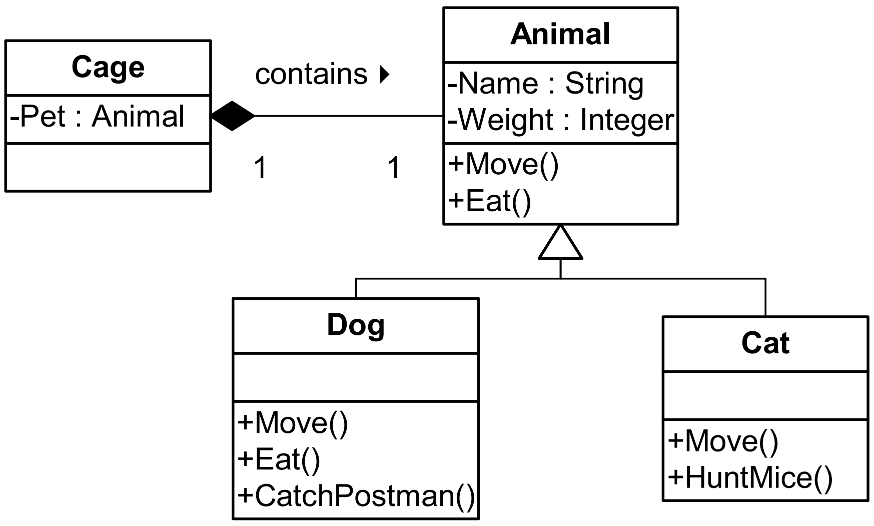


Figure 2.3.3-1 Object Polymorphism

The *Cage* does not know the difference between a cat and a dog. The only information it has is that there is an animal inside. The solution to this problem is polymorphism. The *Cage* calls the *Eat()* method of the animal it contains and the object in the cage itself knows if to call the *Dog::Eat()* method or the default implementation in the animal class. Same goes with the *Move()* method. This behavior is achieved with a virtual function table.

2.3.3.1 Virtual Function Table

Each method with the ability to instantly call the corresponding method of a derived class is a call of a virtual function or method. The runtime implementation of these virtual functions takes the form of Virtual Function Pointers (vptr) and Virtual Function Tables (vtbl).

This technique is based on the compiler silently generating a static array of function pointers for each class that contains virtual functions. This array is called the virtual function table and contains one function pointer for each virtual function defined in he class or its base class. Each instance contains a single invisible data member called the virtual function pointer that is automatically initialized by the constructor to point to the class's vtbl. When a client calls (like the *Cage* class) a virtual function, the compiler generates the code to dereference the vptr, index into the vtbl, and call through the function pointer found at the designated location.

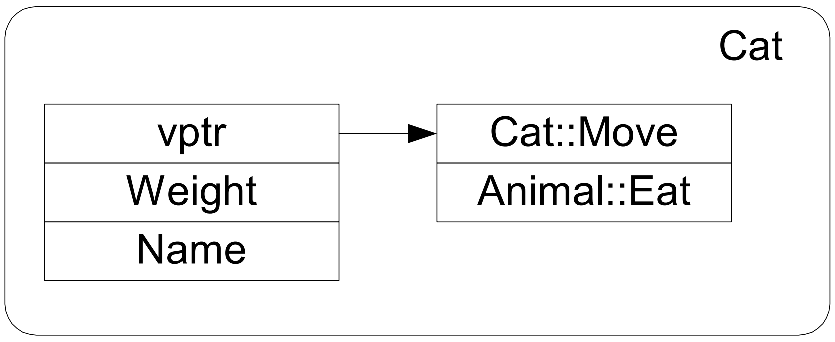


Figure 2.3.3-2 Cat Class Detail

Figure 2.3.3-2 shows this process in the *Cat* class. If a client wants to call the *Move* method the class instance has stored the function pointer to the *Cat* implementation. A default *Eat* function was implemented in the *Animal* class. Therefore the vtbl contains a pointer to *Animal::Eat*.

This mechanism is essential for virtual classes and in consequence the basic abstraction mechanism in COM (s Chapter 2.4).

2.3.4 Templates

Microsoft's Active Template Library (ATL) is, as the name mentions, based on templates and is the main object library used to develop COM applications.

Templates are an advanced programming technique in object oriented programming, but only a few programming languages are capable of templates. C++ is one of those.

Imagine that a programmer wants to develop a cage that is not only capable of containing animals. Instead he wants to program a cage that is able of containing any available type even types that are not programmed at this time.

A template achieves this exactly. A C++ class is able to accept a template parameter as future type.

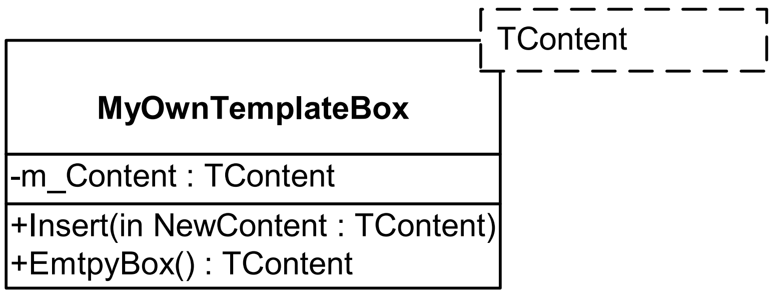


Figure 2.3.4-1 Template Class

The UML diagram above shows such a template. This template is able to insert and remove any type of object in and out of the box. In C++ a class like *MyOwnTemplateBox* would be written like the following.

```
template < class TContent >
MyOwnTemplateBox
{
        TContent m_Content;

        public:
                void Insert( const TContent &newContent );
                TContent EmptyBox();
};
```

A user of that class is now able to include any *Animal* into that Box with the following line of code.

```
MyOwnTemplateBox<Dog> dogCharlie;
```

The C++ compiler now creates a new class at compile time inserting the *Dog* class type instead of the template parameter *TContent*.

2.4 THE COMPONENT OBJECT MODEL (COM)

2.4.1 Basic Idea

COM was designed as a basic technology for MS Windows and multilayer systems. Today all major applications from MS are based on COM like MS Office, MS Project, etc and also major parts of the operating system. Nearly all well known applications provide a COM interface to allow linkage and apply extensions.

2.4.1.1 Reusability

Reusability is a goal every software company wants to achieve. Time is money and every minute counts in the development process. One way to use the already designed and implemented classes is to put them in a static library and every other application may link to this library when creating the final binary file.

Another technique for code reuse is to package a class in a Dynamic Link Library (DLL). The Microsoft (MS) C++ Compiler provides the *__declspec(dllexport)* keyword for just this purpose.

Consider following class *FastString*.

```
// faststring.h version 1.0
class __declspec(dllexport)
FastString
{
        char *m_pszString;
    public:
        FastString( const char *pszString );
        ~FastString();
        int Length() const; //return number of charactes
        int Find( const char *pszSubString ) const; //returns offset
};
```

When the class is exported from a DLL, the *FastString* machine code needs to exist only once on the user's hard disk. When multiple clients access the code from the library, the operating system's loader is smart enough to share the physical memory pages of the DLL between all clients.

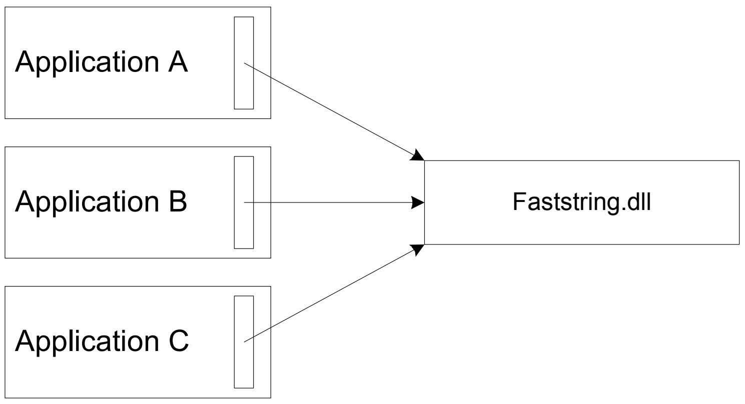


Figure 2.4.1-1 Faststring as a DLL

Now someone wants to enhance the *FaststString* class because it is not fast enough. This developer enhances the class in adding a member, which holds the actual count of characters. Thus acting much faster in returning the length of the string.

```
// faststring.h version 2.0
class __declspec(dllexport)
FastString
{
        const int m_iCount;
        char *m_pszString;
    public:
        FastString( const char *pszString );
        ~FastString();
        int Length() const; //return number of charactes
        int Find( const char *pszSubString ) const; //returns offset
};
```

Now the new DLL is installed, replacing the old one. The new applications using this new *FastString* performs extremely fast and all seems well. Then the user starts a previous application that also happens to use the fastring.dll. Suddenly a dialog appears indicating that an exception has occurred and that all of the end user's work has been lost. What happened? Figure 2.4.1-2 shows the answer.

Version 1.0 of *FastString* required four bytes per instance. Clients written to version 1.0 of the class definition allocate four bytes of memory to pass to the class's constructor. Version 2.0 of the DLL assumes that all clients have allocated eight bytes per instance.

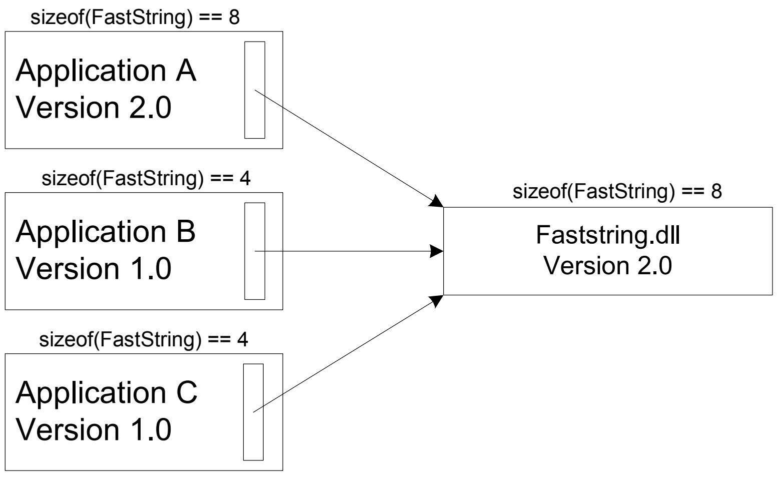


Figure 2.4.1-2 FastSting V2.0

Unfortunately, in version 1.0 clients, the second four bytes of the object really belong to someone else and writing a pointer of a text string at this location is considered rude, as the exception dialog before indicates.

COM provides a solution to this problem. It creates a binary constant interface with the help of two basic object orientation mechanisms – inheritance and polymorphism.

2.4.1.2 Virtual Classes

The first step in building a binary constant interface is to describe the basic object design in the form of virtual classes.

Virtual implies that the class exists only as a definition and has not a working implementation or any data members. The actual implementation is done in derived classes.

The Figure 2.4.1-3 shows a possible implementation with virtual classes. The *AnimalBase*, *DogBase* and *CatBase* classes are pure virtual. This means the base classes contain no implementation.

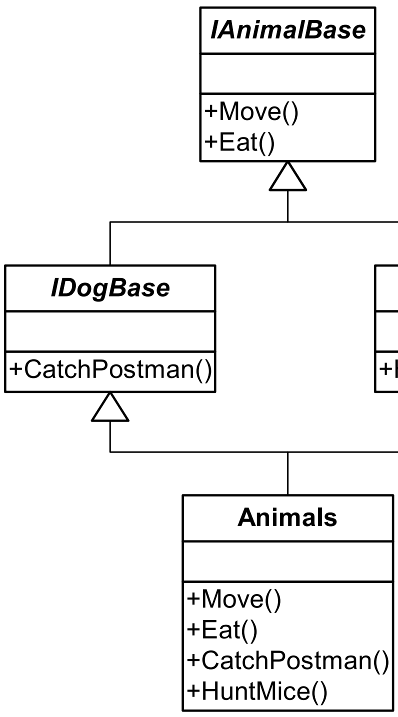


Figure 2.4.1-3 Virtual Class Inheritance

All the working source code is implemented in the *Animals* class that is derived from *DogBase* and *CatBase*. All client methods have to be called on these base classes. There through object polymorphism the implementation method in *Animals* is called instead of the empty one in the base class.

The *Animals* class-implementation may now change at will as long as it contains all methods of all the base classes. A software developer is able to add additional members or methods without causing any changes to the binary class representation or size of the virtual base classes.

2.4.1.3 Binary Interface Across Multiple Programming Platforms

Even though the public operations of the data type have been hoisted to become pure virtual functions in an interface class, the client cannot instantiate *FastString* objects without knowing the class definition of the implementation class. Revealing the implementation class definition to the client would bypass the binary encapsulation of the interface, which would defeat the purpose of using a virtual interface class.

The solution to this problem is to create a public function inside the DLL, which creates the new object.

```
extern "C"
IDog * CreateNewDog()
{
        return static_cast<IDog*>(new Animals);
};
```

One problem resides, each class may contain dynamically allocate memory and wants to free this memory in the object destructor. Unfortunately, this pollutes the compiler independence of the interface class, as the position of the virtual destructor in the vtbl can vary from compiler to compiler. One workable solution to this problem is to add an explicit Release method to the interface as another pure virtual function where the derived class deletes itself in its implementation of this method. This results in the correct destructor being executed.

To handle this self-destroying mechanism the interface should also implement basic resource management. What now follows is an interface that includes just that.

2.4.2 IUnknown

IUnknown is the basic interface of all COM interfaces. All other interfaces used by COM must be directly or indirectly derived from IUnknown. IUnknown is the only COM interface that does not derive from another COM interface. The figure below shows the UML representation of this interface.

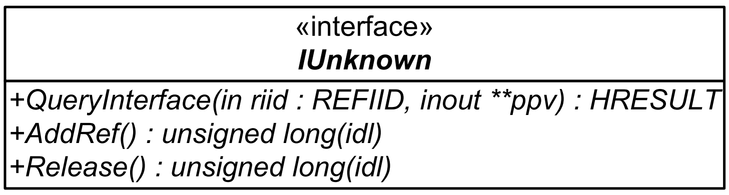

«interface»
IUnknown
+QueryInterface(in riid : REFIID, inout **ppv) : HRESULT
+AddRef() : unsigned long(idl)
+Release() : unsigned long(idl)

Figure 2.4.2-1 IUnknown Interface

This diagram shows that at the binary level all COM interfaces are pointers to vtbls that begin with the three entries *QueryInterface*, *AddRef*, and *Release*. Any interface-specific methods will have vtbl entries that appear after these common three entries.

AddRef and *Release* are used for memory management and *QueryInterface* allows the linkage to other derived interfaces.

Designing a COM object is now fairly simple, as shown below.

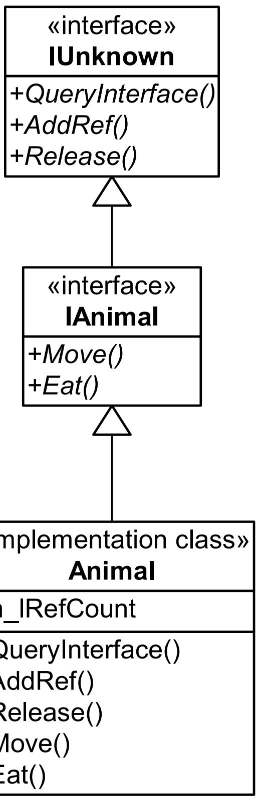


Figure 2.4.2-2 Interface Implementation

The Figure 2.4.2-2 shows the usage of *IUnknown* together with the implementation of an *IAnimal* interface.

Implementing *AddRef* and *Release* is extremely straightforward. *Animal* consists of a counter, which is incremented with each call to *AddRef* and decremented in the *Release* method.

```
ULONG Animal::AddRef(void)
{
        return ++m_lRefCount;
}

ULONG Animal::Release(void)
{
        long res = --m_lRefCount;
        if ( res == 0 )
                delete this;
        return res;
}
```

Release is also responsible for deleting the object itself at the moment the reference counter reaches zero. For a real world COM object, one must implement this reference counting mechanism little bit different, as this implementation is not thread safe. For objects in a multithreaded environment, the Win32 *InterlockIncrement* and *InterlockDecrement* routines should be used to adjust the reference count [5].

With the implementation of *AddRef* and *Release* in place the only remaining IUnknown method to implement is *QueryInterface*. The following is a correct implementation of the remaining method.

```
HRESULT Animal::QueryInterface( REFIID riid, void **ppv )
{
        if ( ppv == 0 )
                return E_POINTER;

        if ( riid == IID_IAnimal )
                *ppv = static_cast<IAnimal*>(this);

        else if ( riid == IID_IUnknown )
                *ppv = static_cast<IAnimal*>(this);

        else
        {
                // unsupported interface
                ppv = 0;
                return E_NOINTERFACE;
        }
        reinterpret_cast<IUnknown*>(*ppv)->AddRef();
        return S_OK;
}
```

The class casts itself to the derived virtual interface class if the client asks for a supported interface. If a specific interface is not supported then the client is informed. Additionally *AddRef* is called because a new reference to this class is accessed by the client. If the client does not need the object any more *Release* must be called.

The object is now completely accessed through virtual methods and classes, even the lifetime and destruction is handled by the object itself. Figure 2.4.2-3 shows the details.

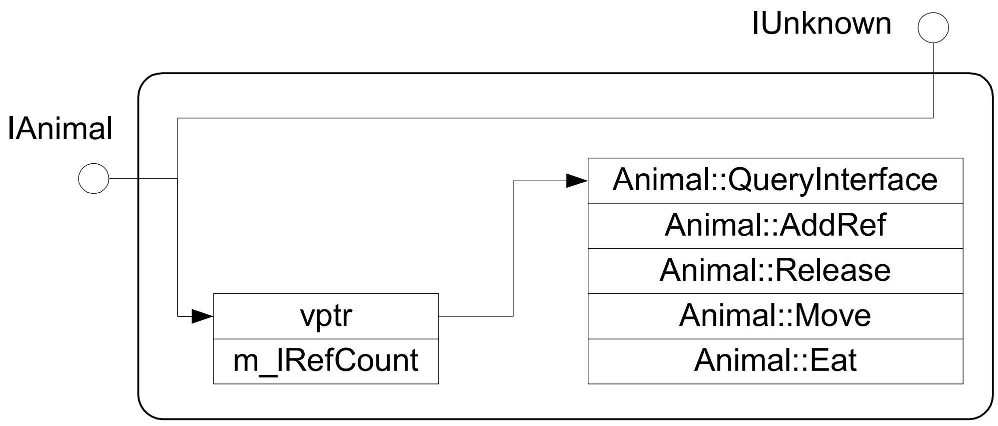


Figure 2.4.2-3 Animal Implementation Detail

All methods from *IUnknown* and *IAnimal* are retransferred to the *Animal* implementation class.

2.4.3 Interface Definition Language (IDL)

The motivation for separating interface from implementation class was to hide from the client all details about an object's inner workings.

Although this last aspect is useful, it does not go far enough in providing a universal substrate for binary components. The important observation is that while clients can use any C++ compiler they choose, ultimately they must use a C++ compiler. The techniques described in the previous chapters provide compiler independence. Ultimately, programming language independence is what is needed for the XML Server. To achieve language independence COM applies the principle of separation of interface from implementation one more time.

To decouple the interfaces from the language used by any particular implementation, one must separate the language used for defining interfaces from the language used to define implementations. If all parties agree on a single language for defining interfaces, it is possible to define the interface once and derive new implementations language-specific mappings, as they are needed. This language is called the Interface Definition Language or IDL.

The Win32 Software Development Kit (SDK) includes an IDL compiler called MIDL.EXE that parses COM IDL files and generates several artifacts.

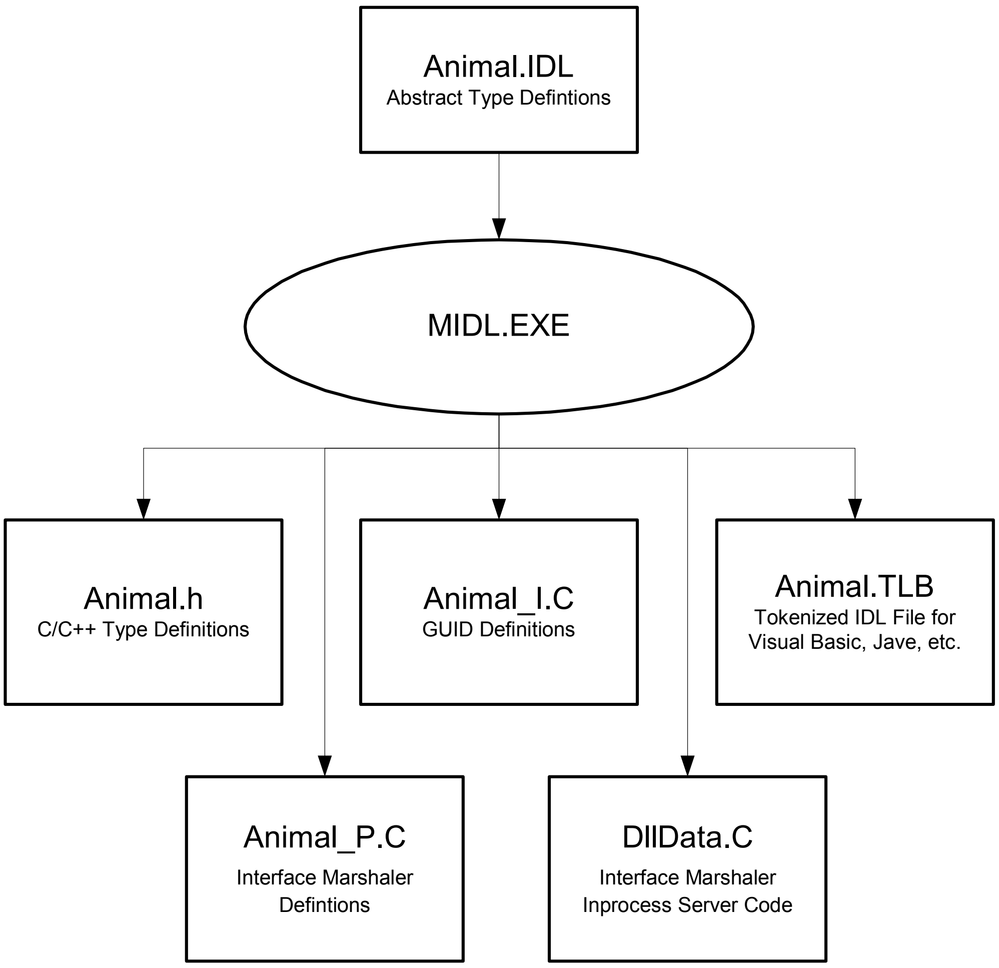


Figure 2.4.3-1 Using MIDL

As shown in Figure 2.4.3-1, MIDL generates C/C++ compatible header files that contain the abstract base class definitions that correspond to the interfaces that are defined in the original IDL file. It also generates source code that allows the interface to be used across thread, process, and host boundaries.

MIDL additionally generates a binary file that allows other COM-aware environments to produce language mappings for the interfaces defined in the original IDL file. This binary file is called a type library and contains tokenized IDL in an efficiently parsed form.

IDL allows interface designers to work mainly in the logical realm using C-style syntax. However, IDL also allows interfaces to precisely specify any aspects of an interface that cannot be derived directly from its C-style logical description using annotations that are formally called attributes. The next lines show the IDL definition of the IUnknown interface.

```
[
    local,
    object,
    uuid( 00000000-0000-0000-C000-000000000046 )
]
interface IUnknown
{
    HRESULT QueryInterface ( [in] REFIID riid,
                             [out] void **ppv );
    ULONG AddRef(void);
    ULONG Release(void);
}
```

As one can see some elements contain additional attributes like *local* or *object* for the interface or *out* in the *QueryInterface* parameter definition.

The animal definition may look like this:

```
[
    object,
    uuid (62E09C14-9FB1-4aa6-884E-E5DAD1171EA2 ),
    helpstring("Base interface of all animals")
]
interface IAnimal
{
    HRESULT Move(void);
    HRESULT Eat(void);
}
```

The most important attribute is *uuid.* This attribute serves as a unique identifier.

Global Unique Identifiers (GUID) are used to eliminate name collisions. GUIDs are used throughout COM to name static entities, such as interfaces or implementations. GUIDs are 128-bit extremely large numbers that are guaranteed to be unique in both time and space. When used to name COM interfaces, GUIDs are often called Interface IDs (IID).

Implementations in COM are also named using GUIDs, and in this context, GUIDs are referred to as Class IDs or CLSIDs. When represented textually, GUIDS are always displayed in the following canonical form:

```
    62E09C14-9FB1-4aa6-884E-E5DAD1171EA2
```

These 32 hexadecimal digits represent the 128-bit value of the GUID. The algorithm used to create these identifiers includes the local machine's network interface address, wall clock time, and a pair of persistent counters to compensate for clock resolution and abnormal system clock changes.

2.4.4 COM Threading

A thread is a path of execution through the code in a process. It has its own call stack and CPU state. Modern operation systems can juggle hundreds of threads to create the illusion that these threads are running simultaneously, even on a machine that has only a single CPU. The operating system does this by assigning a quantum of CPU time to each thread. When this unit of time is complete, the operating system saves the state of the CPU associated with that thread and then gives another thread a chance to run for a quantum of time.

COM organizes thread safety on an object-oriented level. This means the following. The state of an object is defined through its member variables. These variables are sometimes referred to as the *per-instance* state of a class. COM provides an effective threading mechanism where the developer has not worry about thread safe variable access. Or at least he decides if wants to be responsible for the complete thread synchronization.

For the object synchronization scheme to work, a COM client must have some way of indicating whether it uses COM objects across multiple threads or not. The client thread does this with a call to one of the Win32 API function *CoInitialize* or *CoInitializeEx*. According to the parameter of this functions the COM objects are going to reside in a Single-Threaded Apartment (STA) or a Multi-Threaded Apartment (MTA).

If the object resides in the process of the client, the COM object itself must also indicate if it is thread safe or not. This is done in the registry. The *ThreadingModel* registry key can be found at this location:

```
HKEY_CLASSES_ROOT\CLSID\{CLSID_OF_THE_CLASS}\ThreadingModel
```

It is string type, which may contain the values *Single*, *Apartment*, *Free*, *Both* and *Neutral*.

- **Single** Objects of this class are pathologically unthread-safe. Not only are the objects not thread-safe, the class factory which creates the instances, and the DLL entry functions are also not thread safe. This means a single thread must execute all the code in this server to prevent possible synchronization errors, which sure would happen if variables were accessed from multiple threads.

- **Apartment** Objects in this class do not protect their per-instance state from multithreading problems. Access to each instance must be limited to a single thread. But, the per-class state for this class and the class factory and the DLL entry functions are thread-safe.
- **Free** COM objects of this class protect their per-instance state and per-class state from multi-threading problems using thread synchronization primitives, such as critical sections or mutexes. These objects are safe to use in a multi-threaded environment.
- **Both** Objects of this class are instantiated in the same apartment as their client so they can, for maximum performance, execute on their caller's thread. Because they may run in a multi-threaded enviroment, these classes should protect their per-instance state and per-class state (static member variables) from multithreading problems.
- **Neutral** Objects of this class also execute on their caller's thread, but they reside in their own apartment, the Thread Neutral Apartment (TNA), rather then residing in their caller's apartment. Because any thread (STA or MTA) in a process can enter the TNA at any time, objects marked as Neutral must either use thread-synchronization primitives to make themselves thread-safe, or they can use the thread synchronization service by the COM+ runtime[3].

Once the objects and the threads in a process have indicated which apartment they belong to, the COM runtime can make sure the concurrency requirements for each objects are adhered to by restricting which threads are allowed to call methods on a particular COM object.

2.4.4.1 Marshaling

These restrictions are very strict but COM also introduces a mechanism that allows interface pointers to be passed across apartment boundaries. This technique is called marshaling. Marshaling an interface pointer simply transforms the interface pointer into a transmissible byte stream whose contents uniquely identify the object and its owning apartment. This byte stream is the marshaled state of the interface pointer and allows an apartment to import the interface pointer and make method calls on the object. These marshaled object-references simply contain connection establishment information that is completely independent of the object's state.

When an in-process activation request is made for a class with an incompatible threading model, COM implicitly marshals the interface from the object's apartment. Table 2.4.4-1 list all possible modes and when marshalling occurs.

[3] In the COM+ environment all objects should be programmed stateless (without state defining members) or use the COM+ Shared Property Manager ([4] Page 484).

Threading Model	MTA	Primary STA	Other STA
Single	Marshaled	Direct	Marshaled
Apartment	Marshaled	Direct	Direct
Free	Direct	Marshaled	Marshaled
Both	Direct	Direct	Direct
Neutral	Marshaled	Marshaled	Marshaled

Table 2.4.4-1 COM Marshalling

The Primary STA is the thread that first of all others instantiates the COM object.

The custom marshalling happens in proxy/stub DLLs. These DLLs handle the incoming Remote Procedure Calls (RPC) and translate the interface calls and parameters into byte streams.

For STAs the marshalling is arranged through the standard windows messaging mechanism as shown in Figure 2.4.4-1. [9]

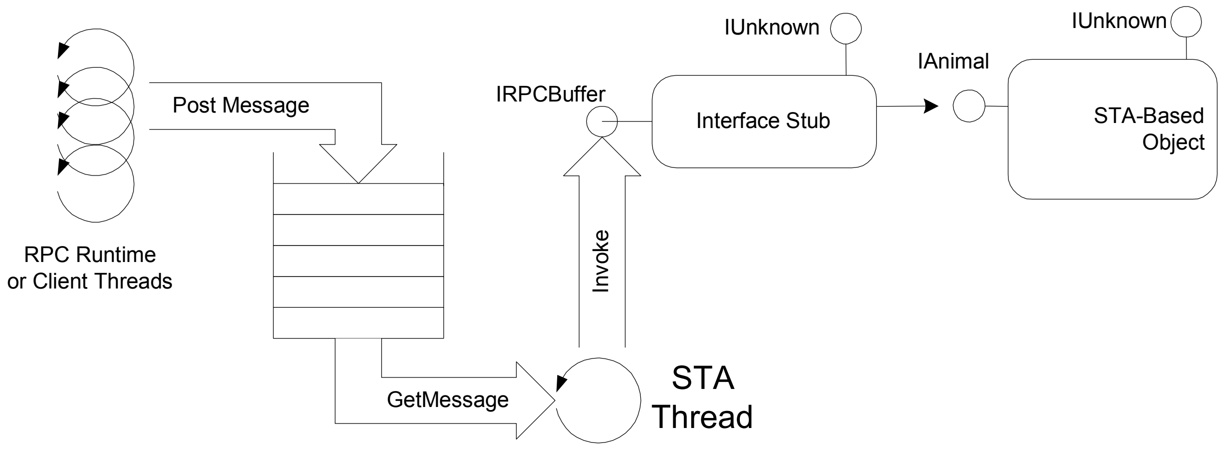


Figure 2.4.4-1 Singlethreaded Apartment Call Dispatching

To enter the STA and dispatch the call to the STA's thread, the RPC thread uses the *PostMessage* API function to enqueue a message into onto the STA thread message queue. This means that to finish dispatching the call, the STA thread must service the queue via some variation of the following code:

```
MSG msg;
while ( GetMessage(&msg, 0, 0, 0) )
        DispatchMessage(&msg);
```

2.4.5 Extended Error Information

The COM standard also consists of a few more interface than IUnknown. Three of them, also used by the XML Server are *ISupportErrorInfo* , *ICreateErrorInfo* and *IErrorInfo*.

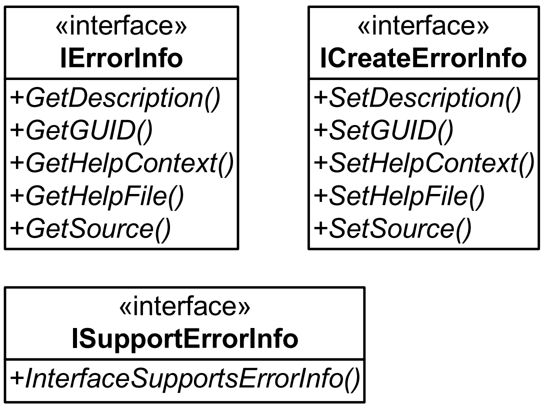


Figure 2.4.5-1 Error Information Interfaces

When an error occurs, a COM object that implements *InterfaceSupportsErrorInfo* may send rich error information to its client by first calling the *CreateErrorInfo* function in the COM API to create a COM exception object. Then COM object can use the *ICreateErrorInfo* interface to poplulate the COM exception object with the source, description, help file, and help context ID for the error that just occurred. Once the exception object is populated with all the necessary error information, the exception object can be send the to its client by first using *QueryInterface* to obtain the *IErrorInfo* interface on the exception object and then passing this interface to the *SetErrorInfo* function as shown in next lines.

```
HRESULT Animal::Eat() {
        HRESULT hr;
        try {
                /* ...*/
        }
        catch ( ...) {
                ICreateErrorInfo pCreateErrorInfo = 0;
                CreateErrorInfo ( &pCreateErrorInfo );
                pCreateErrorInfo->SetGUID( IID_IAnimal );
                pCreateErrorInfo->SetDescription( L"Nothing to eat.");
                IErrorInfo pErrorInfo;
                pCreateErrorInfo->QueryInterface( IID_IErrorInfo,
                                                  (void**)&pErrorInfo );
                SetErrorInfo( 0, pErrorInfo );
                pCreateErrorInfo->Release();
                pErrorInfo->Release();
                return E_NOTHING_TO_EAT;
        }
        return hr;
}
```

2.4.6 DCOM

Distributed COM (DCOM) is that part of COM concerned with enabling COM-based software components to be used over a network. This means that DCOM enables a COM server on one machine and have it used by a client on a different machine. This enables developers to build applications that work together with two, three, four, or even dozens of PC's that exchange and share information. These are enterprise applications that in the not too distant past could only run on mainframe or midrange computers.

Unfortunately, building enterprise class applications introduces new problems, some of them are addressed by DCOM and some are not. Security is an example. These applications need the ability to restrict certain individuals from accessing the information while allowing others to use it. DCOM provides this functionality.

However, the XML Server has not only the goal of distributed processing, it also requires scalability and data-integrity. Scalability means the application should be able to handle simultaneous use by hundreds or even thousands of users.

Data integrity means that updates to data-stores must be made in a consistent way (data must not be corrupted or lost) even in the face of simultaneous use by numerous users. DCOM does not provide this functionality: COM+ and the Microsoft Transaction Server (MTS) do. These technologies are build on top of DCOM. Understanding DCOM is important if one wants to understand COM+ and MTS.

2.4.6.1 The Service Control Manager

Figure 2.4.6-1 shows a diagram of the DCOM network communication. When a client makes an object activation request the Service Control Manager (SCM) is called to locate the object.

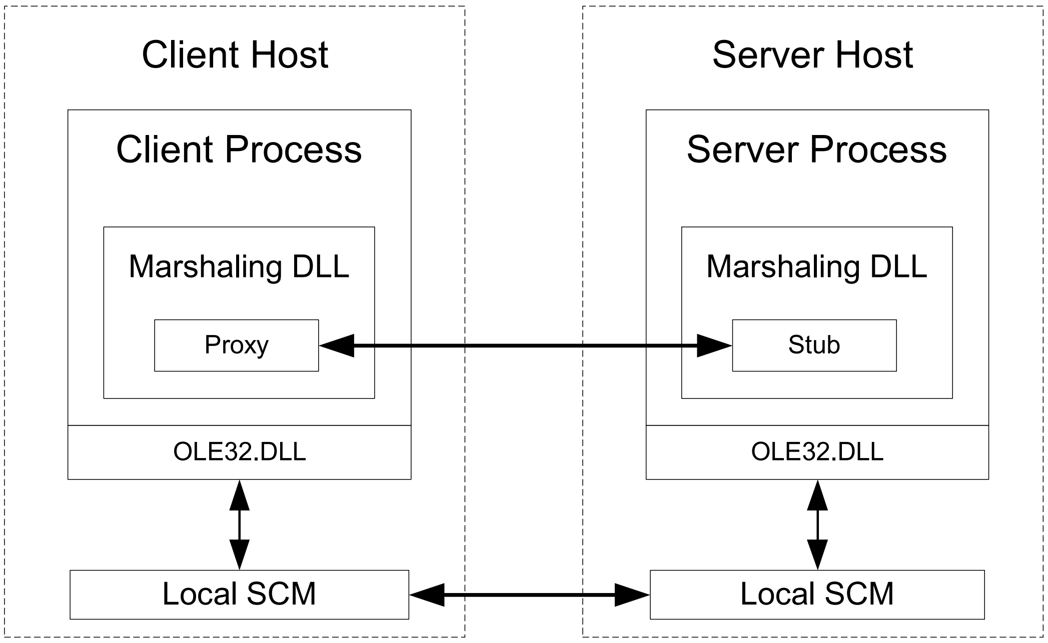


Figure 2.4.6-1 DCOM Communication

The SCM takes the CLSID that is passed to *CoCreateInstance* and searches in the registry for its key beneath `HKEY_CLASSES_ROOT\CLSID\` in the system registry. When the SCM finds the key in the registry it searches for either an *InprocServer32* or a *LocalServer32* key. If it finds *InprocServer32*, it loads the DLL identified in the default for the key into the client process. If it finds a *LocalServer32* key, it starts up the executable and establishes an Inter-Process Communication (IPC) link between the client and the server.

If there is neither of these two keys the SCM next searches for the following key in the registry:

```
HKEY_CLASSES_ROOT\AppID\
```

Each COM Server creates a registry entry below this key. Beneath this AppID security-related information for a server and a key called *`RemoteServerName`* is stored. The *`RemoteServerName`* key identifies a remote machine where the server can be found. This can either be an IP address or the domain name of the remote machine on which the remote server is resident.

If the SCM finds this key, the SCM on the remote machine is contacted and a request is made that this machine starts the server and returns a proxy. The SCM on the remote machine first verifies that the requesting user is authorized to use this server, sets up a stub in the server process and returns a proxy to the client. After the client has the proxy, all further communication occurs directly between the proxy and the stub as shown above.

Because both the client machine and the server machine require marshaling support, the marshaling DLL must be installed and registered on both machines if the COM server wants to implement custom interfaces. If the server is just using standard interfaces, marshaling support is provided by the ole32.dll.

2.4.6.2 Security

The single most important issue that DCOM adds to COM is security. When running a COM client and server on the same machine, the only security needed is that which is required to prevent users from logging into the machine. Once running clients and servers on separate machines several different forms of protection are needed.

- **Authentication** That means it must be able to determine in an accurate and un-subvertible way, the identity of the user who is attempting to use the COM server.
- **Access Control** One may want to give some users full access to the COM server, others may have access to only certain features, but would be restricted from using other ones and some would be denied access for others completely.
- **Security Token Management** Some of the issues involved here are: with whose security token should a COM server process run? Should it run with the clients security token, the security token of whomever is logged in to the server at the moment, or perhaps some other token.

DCOM takes care of all forms of protection. There are two ways of configuring the communication security between the client and the server: via the registry or programmatically. Usually the security is not programmed hard coded into the source code.

Instead the network administrator using the MS Windows tool DCOMCNFG.EXE configures it [13].

2.5 THE NEXT STEP: COM+

COM+ was introduced with Windows 2000 and is a set of system services that are designed to make it easy to develop enterprise-class distributed applications. COM+ is really just a runtime environment for COM objects. This runtime environment provides the following services to COM objects:

- Fine-gained security
- Distributed transactions,
- Thread synchronization,
- Load balancing,
- Asynchronous store and forward method invocation,
- Enhanced scalability through just in time activation
- Pooling of objects and database connections.

In order to use this runtime environment COM objects must be implemented as an in-process server and then configured to run within the COM+ runtime environment. Then it is possible to select services to be applied for that object. Windows 2000 provides a graphical tool called the Component Services Explorer that provides just that.

Each COM+ Server application runs in its own process. Actually the server application runs in a surrogate process that is provided by COM+. Each surrogate process is an instance of DLLHost.exe as shown in Figure 2.4.6-1.

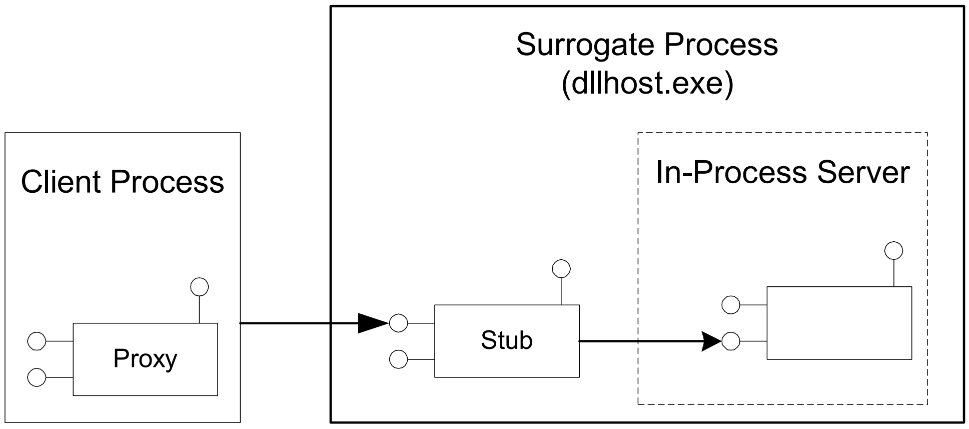


Figure 2.4.6-1 Surrogate Process

As with all out of process COM objects they must be accessed through proxy/stub marshaling.

2.5.1 COM+ Context

The COM+ Context is the runtime environment in which one or more compatible COM+ objects in a particular process execute. Compatible objects are objects that share the same runtime requirements, i.e. they have the same settings of their COM+ attributes. A Context resides in an apartment within a process as displayed in Figure 2.5.1-1. An apartment can contain one or more Contexts and a Context can contain one or more objects.

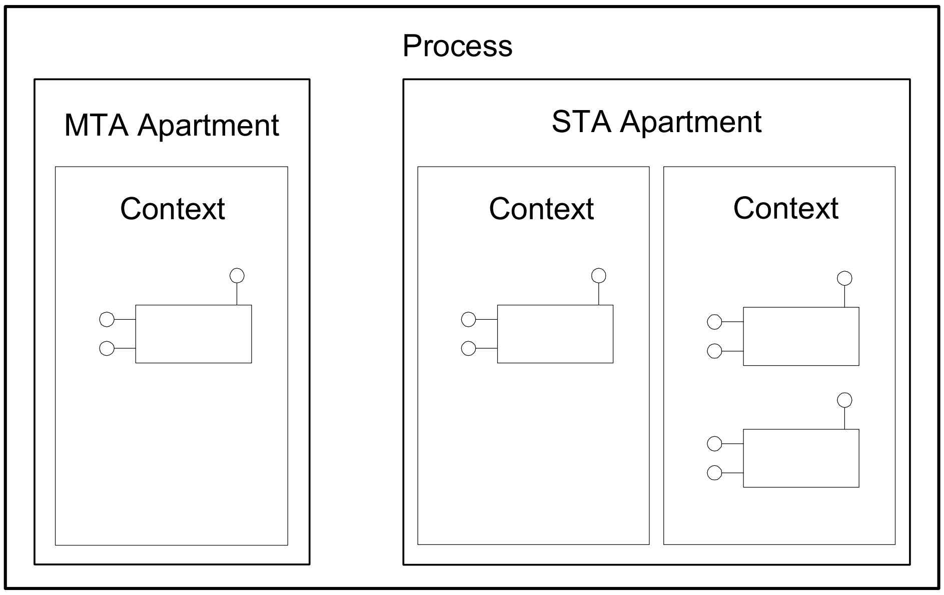


Figure 2.5.1-1 Processes, Apartments, and Contexts

When the COM+ runtime first creates the Context for an object, it reads the settings for the COM+ attributes (security, transaction, object pooling, …) that are configured for the object's class from the COM+ catalog. These settings are stored in a system-created COM object called an Object Context. This Object Context can be used to retrieve and modify context information. Calling the new Windows 2000 API function called *CoGetObjectContext* retrieves this Object Context. Figure 2.5.1-2 shows the interfaces that are supported.

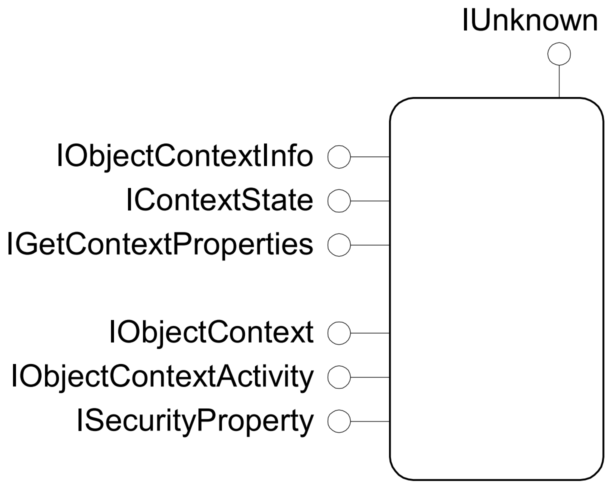


Figure 2.5.1-2 The Object Context

- **IContextState** Controls object deactivation and transaction voting by manipulating context state bits. Calling the methods of this interface sets consistent and done bits independently of each other and get the current status of each bit.
- **IObjectContextInfo** Returns transaction, activity and context information on the current context object. Using the methods of this interface, you can retrieve relevant information contained within an object context.
- **IGetContextProperties** Provides access to object context properties.
- **IObjectContext** Provides access to the current object's context.
- **IObjectContextActivity** Used to retrieve a unique identifier associated with the current activity. This activity identifier is a GUID, and is only valid for the lifetime of the current activity.
- **ISecurityProperty** Used to deremine the security ID of the current object's original caller or direct caller.

2.5.2 Just in Time Activation and Object Pooling

Often it takes a long time to instantiate a server object. Therefore DCOM developers often write client applications where they instantiate all the objects they need when they are started and then hold on to those objects for the life of the application. While this is simple and seems more efficient from the client's perspective this ruins the scalability of the server. A thousand clients may be holding onto their object references but it is likely that only a small number of those are actually executing a method at any given moment.

The idea behind the COM+ services Just in Time Activation and Object Pooling was to make it easier for developers to reuse objects without having to think about object management routines. When an object sets its *Deactivate-On-Return* bit, the COM+ runtime deactivates the object. The object is either destroyed or returned to the object pool (depending on the setting of the object pooling attribute). However, the client still holds the proxy for the object, the RPC Channel is in place an the stub still exits.

With object pooling, the COM+ pool manager maintains a pool of objects, as shown in Figure 2.5.2-1. When a client attempts to activate an object of that type, the COM+ runtime returns an instance from the pool if one is available, instead of creating a new instance from scratch.

The client can use this object for as long as it takes. When the client releases its last reference to the object, or the object returns from a method with its deactivate-on-return bit set, the object is deactivated. Instead of being destroyed, the deactivated object is returned to the object pool. A COM+ component must meet the following requirements before it can be pooled.

- It must reside in the Thread Neutral or Free-Threaded Apartment of a process.
- It must be stateless. That means it must not have any state defining member variables.
- It must not have thread affinity
- It must be aggregateable[4].

[4] See in [5] for a detailed description about COM Aggregation.

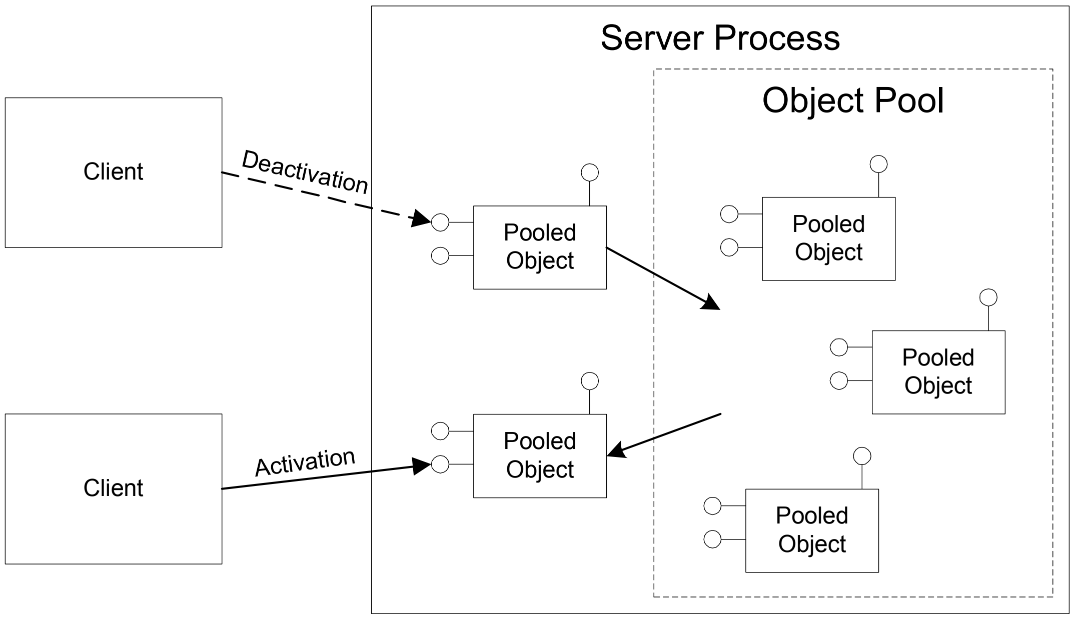


Figure 2.5.2-1 Object Pooling

If an object meets all these requirements a minimum and maximum number of pooled objects can be specified in the COM+ runtime environment.

2.5.3 Transactions

2.5.3.1 What Is A Transaction?

A transaction is a series of operations that have the following properties[5]:

- Atomicity
- Consistency
- Isolation
- Durability

One can understand what these four properties mean by considering what happens when going to the bank and deciding to transfer $500.00 from one bank account to another. There are actually two operations that take place in completing this operation: (1) $500.00 is withdrawn from account, and (2) $500.00 is deposited into the other one. If both operations succeed, everyone is a winner. But what would happen if the bank's computer failed in the middle of the account transfer operation. There is not an acceptable outcome. Both operations must succeed or they both must fail. This is what *Atomicity* means. *Consistency*, in this example, means the amount deposited into one account should match the amount deposited into the other one. Consistency is enforced by application logic with the help of a Database Management System (DBMS) and/or a Transaction Processing Monitor.

[5] Defined by [4], page 472.

Isolation means that a separate transaction that is executing concurrently with another account transfer should not see an invalid intermediate state such as $500.00 has been withdrawn on one account but has not yet been deposited. Isolation is usually implemented using locking.

The *Durable* property of a transaction means that after the transaction is committed, the updates made by the transaction should never be lost. A system crash, network failure, or even someone inadvertently pulling the power cord, should not cause updates to be lost.

2.5.3.2 Transaction Operations

Transactions have several operations to ensure all transaction properties described above are met.

- Begin
- Commit
- Rollbak

The basic steps of using these transaction operations are shown in the pseudo-code below.

The *Begin* operation starts a transaction. When all the operations in the transaction are complete, one can call the *Commit* function to commit the transaction.

```
try
{
        Transaction.Begin();

        // Withdraw $500.00 from savings account
        /* ... */

        // Deposit $500.00 into checking account
        /* ... */

        Transaction.Commit();
}
catch ( ...)
{
        // if anything goes wrong, rollback the entire operation
        Transaction.Rollback();
}
```

If the transaction fails at any time, you can call the *Rollback* function, which undoes everything since the call to *Begin*. One can make several updates to the database, but none of these updates are visible to anyone outside the transaction until the *Commit* function is called. The Commit applies all the updates in an atomic step.

2.5.3.3 Distributed Transactions

Using the transaction management functions that are built into the DBMS will work okay as long as all of the information is stored in a single database. Unfortunately, most enterprises don't store all their important information in a single database. In many cases, the information is spread among many databases. Take an online retailer as an example. A database of customer account information might be stored in an Oracle database at the corporate office.

But many e-commerce businesses do not fill their orders themselves; they simply run the Web site and advertise. One or more partners actually fill the orders. In this situation, when the online retailer receives an order, it must be able to send the order information to its partners. The mechanism used to send this information could be to write directly to the partner's SQL Server database, or they might send a message to their partner using a message queuing product like Microsoft Message Queue (MSMQ). In either case, the placing of the order to be filled and the debiting of the customer's account must be done in a secure transaction way.

Either all of the servers must commit their part of the transaction or, if any of the resource managers are unable to commit their part of the transaction, all of them must rollback.

2.5.3.4 2 Phase Commit

The key to implement distributed transactions is the 2-phase commit protocol. In this protocol, the activity of the resource mangers must be controlled by a separate piece of software that is sometimes called a transaction manager or transaction coordinator. The steps in this protocol are shown in the figures below.

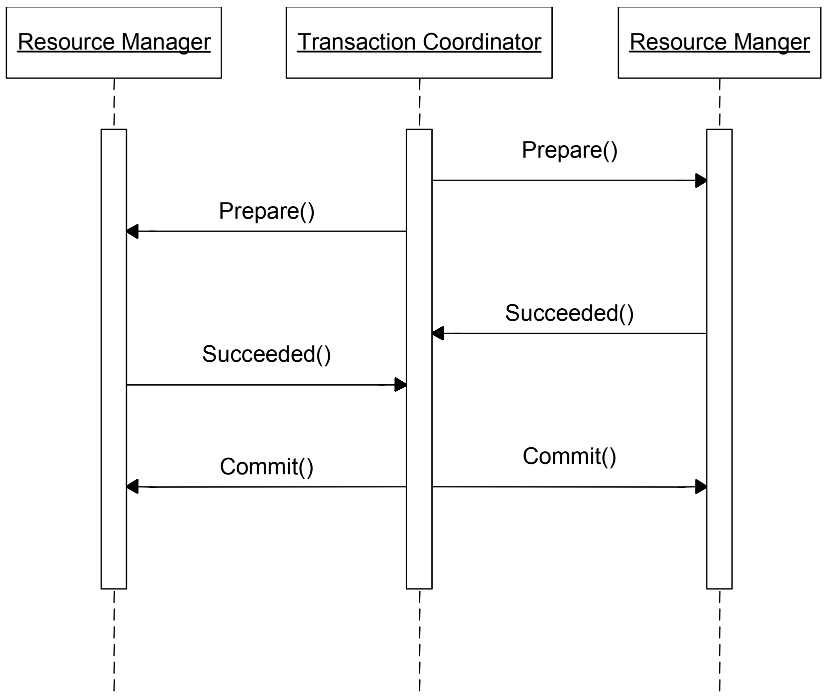


Figure 2.5.3-1 2 Phase Commit Protocol - Transaction Commit

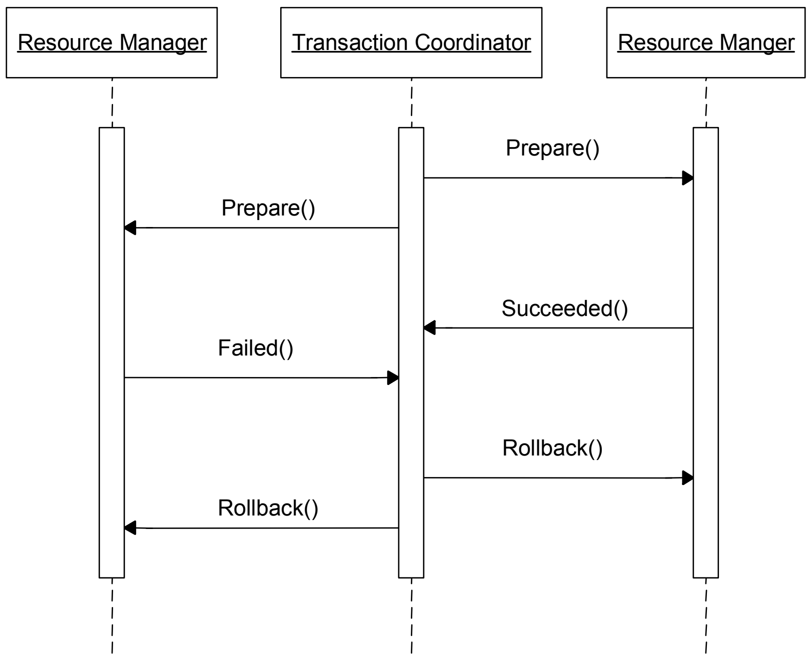


Figure 2.5.3-2 2 Phase Commit Protocol – Transaction Rollback

In the prepare phase the transaction coordinator asks each resource manager to get ready for the commit. Each resource manager is responsible to receive a state where it is able to commit in an atomic operation. Only after all succeed messages have arrived at the transaction coordinator then the transaction is committed. If one operation fails all of the operations that have taken place over several resources are rolled back.

2.5.3.5 Transactions and COM+

In COM+ every COM Component may use the integrated Microsoft Transaction Service (MTS) as transaction coordinator.

Every object is responsible for telling the MTS its current transaction state. To do this an object must obtain its object context interface and call the appropriate methods.

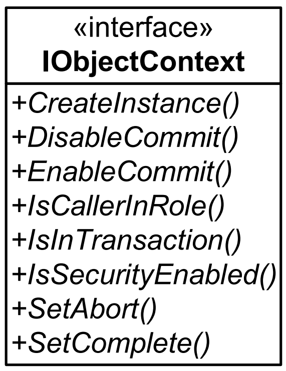


Figure 2.5.3-3 IObjectContext Interface

The methods for transaction voting are *DisableCommit*, *EnableCommit*, *SetAbort* and *SetComplete*.

- **DisableCommit** Tells the MTS that the transaction is in an error state, but this may change in the future.
- **EnableCommit** Tells the MTS that currently the transaction may be commit but work is still in progress that would make a commit impossible.
- **SetAbort** Tells the MTS to rollback the transaction.
- **SetComplete** The object is ready to commit.

The complete transaction behavior like committing or the rollback are performed by the MTS and the object does not have to implement any of the commit or rollback activities, which is a great help for developing transaction safe applications. The next lines show a possible transaction voting implementation.

```
HRESULT Animal::Eat()
{
        HRESULT hr(S_OK);
        IObjectContext pObjectContext=0;

        hr=CoGetObjectContext(IID_IObjectContext, (void**) &pObjectContext();
        try
        {
                // perform work here
                /* ...*/

                pObjectContext->SetComplete();
        }
        catch ( ...)
        {
                pObjectContext->SetAbort();
                hr = E_FAIL;
        }
        pObjectContext->Release();
        return hr;
}
```

In the example above, the object first retrieves its own Object Context. If everything worked out fine the MTS is told to complete the transaction. If something unexpected happens (like an exception) the executions continues in the exception handler and the transaction is forced to abort. At last *Realease* is called to free the Object Context pointer.

3 IMPLEMENTATION

The developed XML Server uses all of the programming techniques described above to implement an engine, which is capable of executing commands on multiple data stores and returning possible results as XML data. The XML Server is placed within the COM+ environment to achieve security goals, transaction safety, scalability and availability for multiple programming languages.

The server was implemented in C++, one of the major languages available for object-oriented programming. The implanted classes and their relations are displayed as UML diagrams because the are much better suited to display the dependency between objects then hundreds of lines of C++ header class definitions.

3.1 A UNIVERSAL DATA ACCESS SERVER

The XML server was developed as a middle tier data platform[6] to make it easier for other applications to retrieve data from multiple data sources where data is linked between each other. In many companies different data platforms are used to store data, which is related to each other, but there is not a tool available to link this data and merge it into one consistent data form.

The data is transferred to the client as an XML file, because this allows structuring the output of nearly every data format. It is also a great help for applications that want to transport this data over the Internet because their already exits internet transportation protocols bases on XML and HTML for internet services, like SOAP. Also modern browser support XML together with a XML Style Sheet as data rendering engine. This means if the data server already returns XML data clients have to spend less effort in data displaying engines.

3.2 IMPORTANT CLASSES AND LIBRARY'S USED

This Server was build from scratch but some additional libraries from Microsoft were used to speed up the development process.

3.2.1 The Standard Template Library (STL)

The STL is part of the ANIS C++ standard[7] this means every new C++ version must include this library and functionality. This allows platform independency as the STL has the same behavior under Linux as under Windows. The following classes where used extensively. For a detailed description of the C++ standard and the STL refer to [1].

[6] N tier design is described in great detail by [3].

[7] The STL was added into the draft standard at the July 14, 1994 ANSI/International Standard Organization (ISO) C++ Standards Committee meeting, and is now included in every major version of C++. The ISO C++ standard (ISO/IEC 14882) was ratified in 1998.

3.2.1.1 string

In C character string are pointers to the first character in a character array. This makes it difficult for developers to handle strings in a complex manner.

The *string* class of the STL capsules this character pointer and exports very helpful methods for string parsing. It also handles the dynamic allocation and deallocation of memory while increasing or decreasing the string size.

In the XML Server the UNICODE version of *string*, the *wstring* was used. UNICODE strings store characters in two bytes instead of one. Therefore they also allow the storage of characters outside of the ASCII definitions. This is important in applications that use the Chinese or Japanese character set.

3.2.1.2 vector

The *vector* template of the STL is a great enhancement over the build in C array type of pairs (name, number). Because this array has a fixed size and if a larger size is chosen lots of space is wasted.

The *vector* allocates the memory used to store the information dynamically. Also the *vector* allows data access in a constant time that means it is as fast as a C array without the problem of array overflow when inserting items.

In the XML Server, vectors are used to store a variable parameter list of XML Commands because the *vector* template builds a small-sized parameter list type, with quick access for parsing.

3.2.2 The Active Template Library (ATL)

The Active Template Library is a set of template classes provided and supported by Microsoft. The ATL was primarily designed to speed up the development of COM objects. It contains standard thread safe implementations of standard COM interfaces like *IUnknown*, *IClassFactory*, *IDispatch*, *ISupportsErrorInfo*, etc. They handle the construction and self-destruction of objects and provide a basic object framework for easier access to the Win32 API. For Example the ATL includes the class *CWindow*, which capsules a window handle, and allows basic window manipulation.

However, as with all templates and libraries, they require a good knowledge in C++ to be effectively used and adjusted to each developing task.

The Active Template Library is described in detail in [7].

3.2.3 Additional Helper Classes

Microsoft also provides some additional classes for handling some MS and/or COM specific data types. Without these classes the source code would be overloaded with code fragments just for the basic usage of that types.

3.2.3.1 _variant_t

The *_variant_t* class capsules a VARIANT structure. The VARIANT is used for communication with scripting languages that are only loosely connected to data types. This structure is a union capable of storing multiple data types at the same location in memory. Scripting languages like Visual Basic Script make intensive use of VARIANTs. *_variant_t* also implements conversion methods that allow quick conversion from one type into another.

3.2.3.2 _bstr_t

The *_bstr_t* capsules a BSTR. The BSTR must be used as string parameters in all interfaces that will be used from Visual Basic or Java. This includes the XML Server interfaces.

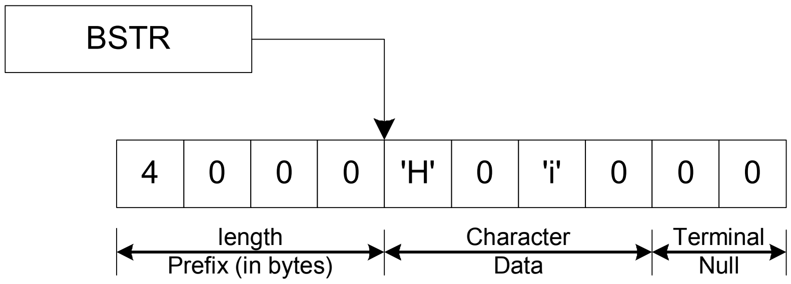


Figure 3.2.3-1 "Hi" as BSTR

As Figure 3.2.3-1 shows BSTRs are length-prefixed, null terminated strings of UNICODE characters. The length prefix indicates the number of bytes the string consumes (not including the termination null) and is stored as a four-byte integer that immediately precedes the first character of the string. To allow BSTRs to be freely returned from methods without concern for memory allocation, all BSTRs are allocated from a COM-managed memory allocator. The _bstr_t class handles these memory API functions.

3.2.3.3 _com_ptr_t

This class is a smart pointer template that manages a COM interface pointer. A smart pointer uses reference counting as a mechanism for freeing the encapsulated pointer[8]. COM pointers already include reference counting therefore this smart pointer uses the IUnknown *AddRef* and *Release* methods for counting and destruction.

The *#import* directive[12] from the MS Visual Studio also uses *_com_ptr_t* as base class for all interface wrappers. The advantage is that the developer is not forced to take care of missing *AddRef* or *Release* calls.

[8] There are other mechanisms of automatically freeing allocated memory. For example the STL auto_ptr uses an ownership indicator, which ensures that the pointer is deleted only once.

3.3 Server Design

3.3.1 Raw Concept

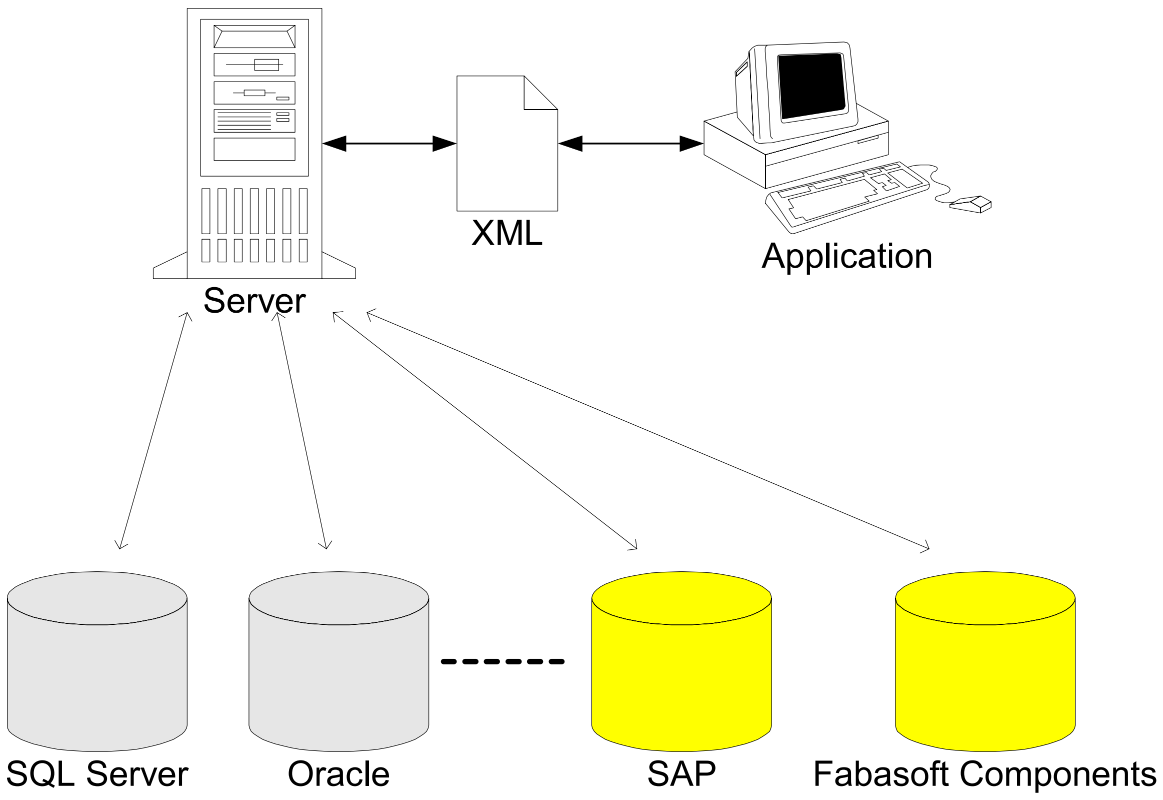


Figure 3.3.1-1 Basic Server Design

Figure 3.3.1-1 shows the part of the server in the information chain. The Server has contact with the different types of data stores. Therefore the server should be able to handle different kinds of data. Like relational data stored in the SQL Server or an Oracle Database. The server should also be able to understand other data sources like the data stored in Fabasoft Components (FSC). This is a middle ware application, which capsules a relational database and enables developers modeling of business objects and workflows. FSC does not store the logical data in tables; instead data is stored as different types of objects.

The Server should also be extensible and be able to handle future types of data. This is achieved by using plugins that can be added into the server. Some plugins have been developed as a standard complement to the server. But developers from other companies or customers may also build their own customized plugins, as the communication interface is freely distributed.

3.3.2 Plugin Design

Figure 3.3.2-1 shows the basic server structure and dataflow of the XML Server. The server talks to the different data stores through special plugins, which are integrated within the server. These plugins communicate with the store and translate the specific data to separate XML streams.

Then these streams are transformed and merged together to one big XML Document.

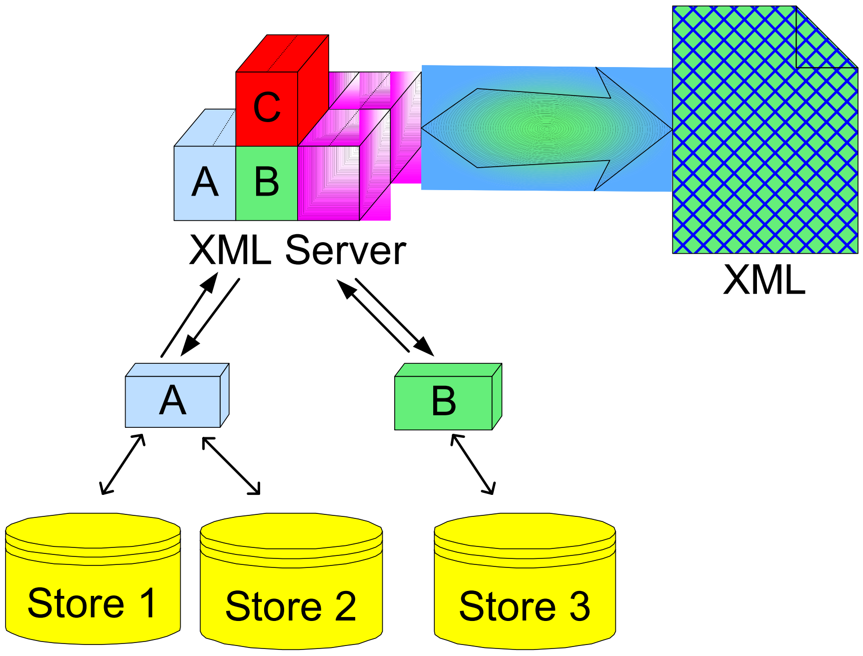


Figure 3.3.2-1 XML Server Function Design

Another feature is that the data flow takes part in a distributed transaction. Meaning that if an error occurs, the whole transaction is rolled back to the beginning and appropriate error handling can be processed. This is extremely important when writing data back to the data stores.

Using plugins to translate the store specific data into the generally used XML format gives the server the possibility to merge data from different stores into one consistent data format. Consider the following XML Files

Store 1 Data

```
<result>
  <droid>
    <type>Repairs</type>
    <name>R2D2</name>
    <owner>Skywalker</owner>
  </droid>
</result>
```

Store 2 Data

```
<result>
  <person>
    <name>Skywalker</name>
    <rank>Commander</rank>
    <locat>Endor</locat>
  </person>
</result>
```

These two results are connected to each other. If one result set comes from an SQL Server database and the other one from an Oracle database it is very different to connect these two records and create a new more informative one.

With the help of the plugins the server is now able to merge these two data elements together into one logically connected form.

The two XML files above could for example build the following new record.

```
<person>
  <name>Skywalker</name>
  <rank>Commander</rank>
  <location>Endor</location>
  <equipment>
    <droid>
      <type>Repairs</type>
      <name>R2D2</name>
    </droid>
  </equipment>
</person>
```

The XML Server achieves this functionality to merge data and build an XML data tree of results with its own XML Command stucture.

3.3.3 XML Server Command Structure

The XML Data can be retrieved with so called XML Commands. Each XML Command can contain any number of additional subcommands, as shown in Figure 3.3.3-1.

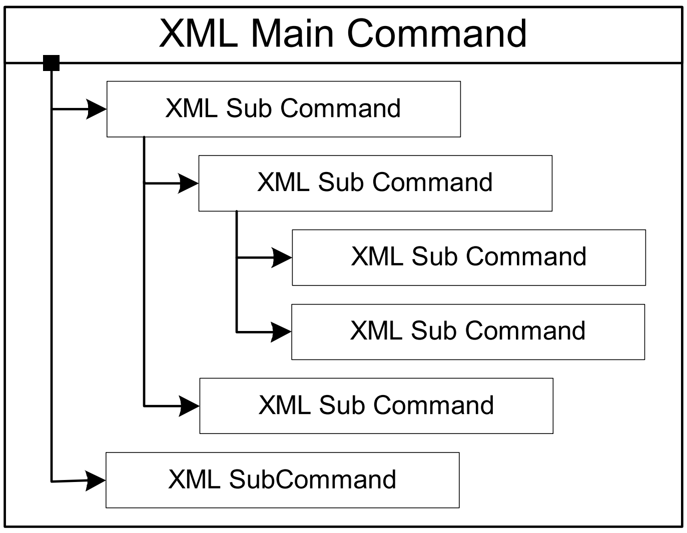


Figure 3.3.3-1 XML Server Command Structure

This XML command tree is put under an XML Main Command. This XML Main Command is identified by a name. The client uses this name as a reference if it wants to execute such commands. Each XML Sub Command contains numerous properties:

- **Command Name** Every command has a name. This name is used as XML tag root name for all data that is returned of this command.
- **Data Source** Each command can have its own data source independent of the data sources of the parent and/or child commands.

- **Command String** This command string is forwarded to the data source plugin. The plugin has to interpret this string and translate it into data source specific commands.
- **Record Name** The XML Server assumes that it receives a list of XML sub data tags. This attribute specifies the name that one of the XML sub elements will receive.
- **Default Parameter List** Each command contains its own list of parameters, which could be parsed into the command string.

The XML Server executes each Sub Command and collects all the XML results from the plugins. These XML results are merged together to one XML Main Command result stream.

3.3.4 XML Command Parameters

Each XML Command can have specific parameters. Every data member already collected from parent XML commands or from the main command can serve as a parameter for each XML Sub Commands.

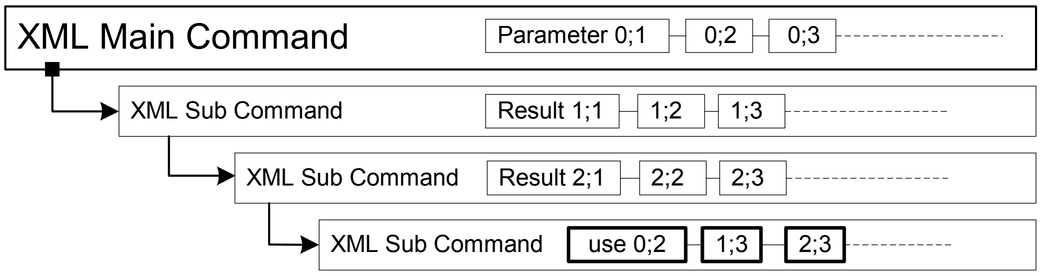


Figure 3.3.4-1 XML Server Parameter Usage

Each main command parameter and each XML result value can be addressed with two values. The first one is the command level starting with zero from the main command level. The second one is the result index starting with one containing each previously returned XML data content.

This parameter structure makes it possible to link results from different data sources with each other. If for example one data source returns the national insurance number as a result then it is possible to search with this number in another database resulting in additional data corresponding to this number.

3.4 SERVER IMPLEMENTATION

3.4.1 XML Server Interface

This is the connection point for each client who wants to communicate with the XML Server. Figure 3.4.1-1 shows all interfaces and their relationship supported by the server. *IUnknown* is as told before the basic interface of all COM objects. *IDispatch* is implemented to enable Scripting Languages like VB Script or JScript to use this server. *IObjectControl* is used by the COM+ environment to signal object pooling events. *ISupportErrorInfo* is implemented for returning textual error information.

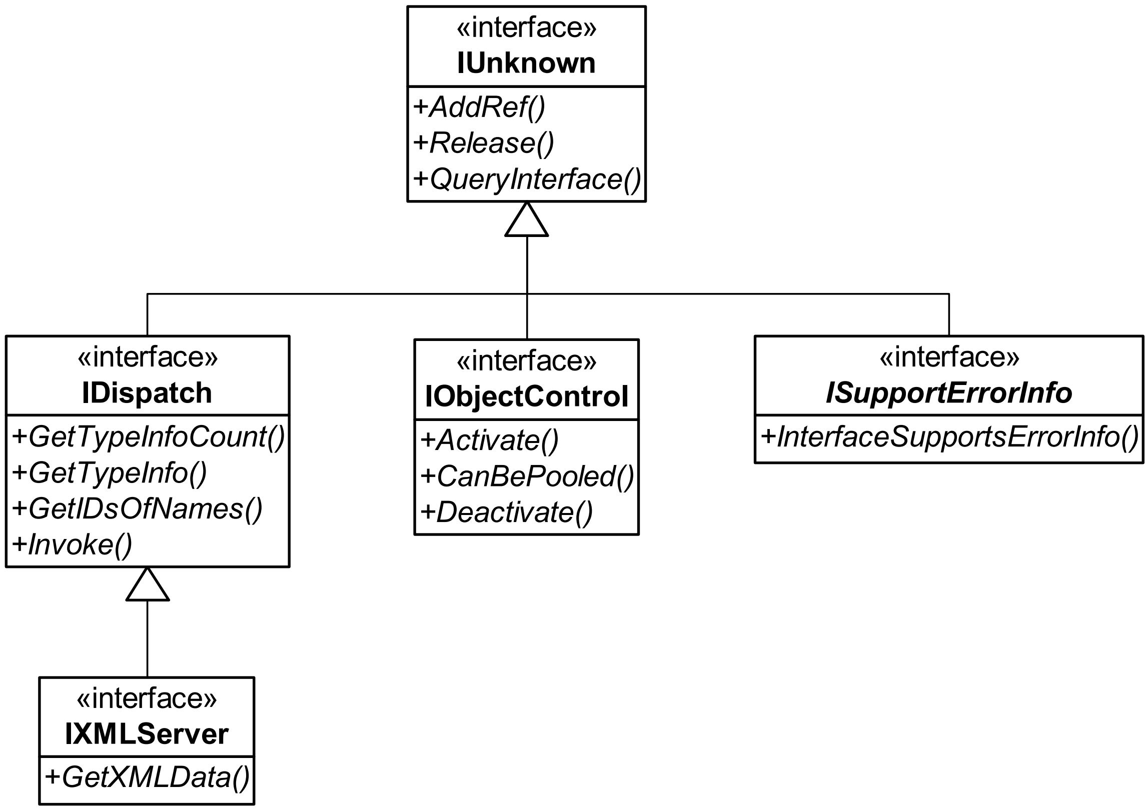


Figure 3.4.1-1 XML Server Interfaces

The interface that gives the server all its power is *IXMLServer*. The figure below shows this interface in detail.

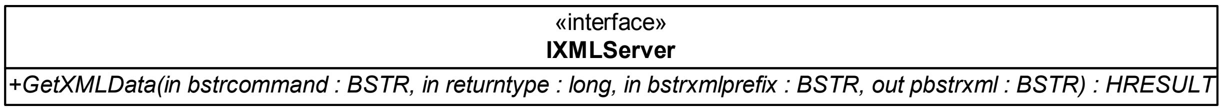

«interface»
IXMLServer
+GetXMLData(in bstrcommand : BSTR, in returntype : long, in bstrxmlprefix : BSTR, out pbstrxml : BSTR) : HRESULT

Figure 3.4.1-2 IXMLServer Interface

The Interface again consists of only one method. This method has the following parameters:

- **bstrcommand**: This is the reference to the Main XML Command as a string. Command parameters may be added as strings separated by space.
- **returntype**: defines the type of the return value. Specifies to return the XML either as a Stream or as the name of a file located on the hard disk.
- **bstrxmlprefix**: The client can specify a prefix, which precedes the XML stream. This is useful for linking to a style sheet or to specify the appropriate character set.
- **pbstrxml**: After command execution this parameter holds the XML stream or name of the file.

3.4.2 Internal Structure

Figure 3.4.2-1 shows an UML diagram of the XML Server. Derived from *IXMLServer* is the implementation class *CXMLServer*. Note that this class does not contain any members. This would prevent the COM+ environment from storing instances of this object in the object pool.

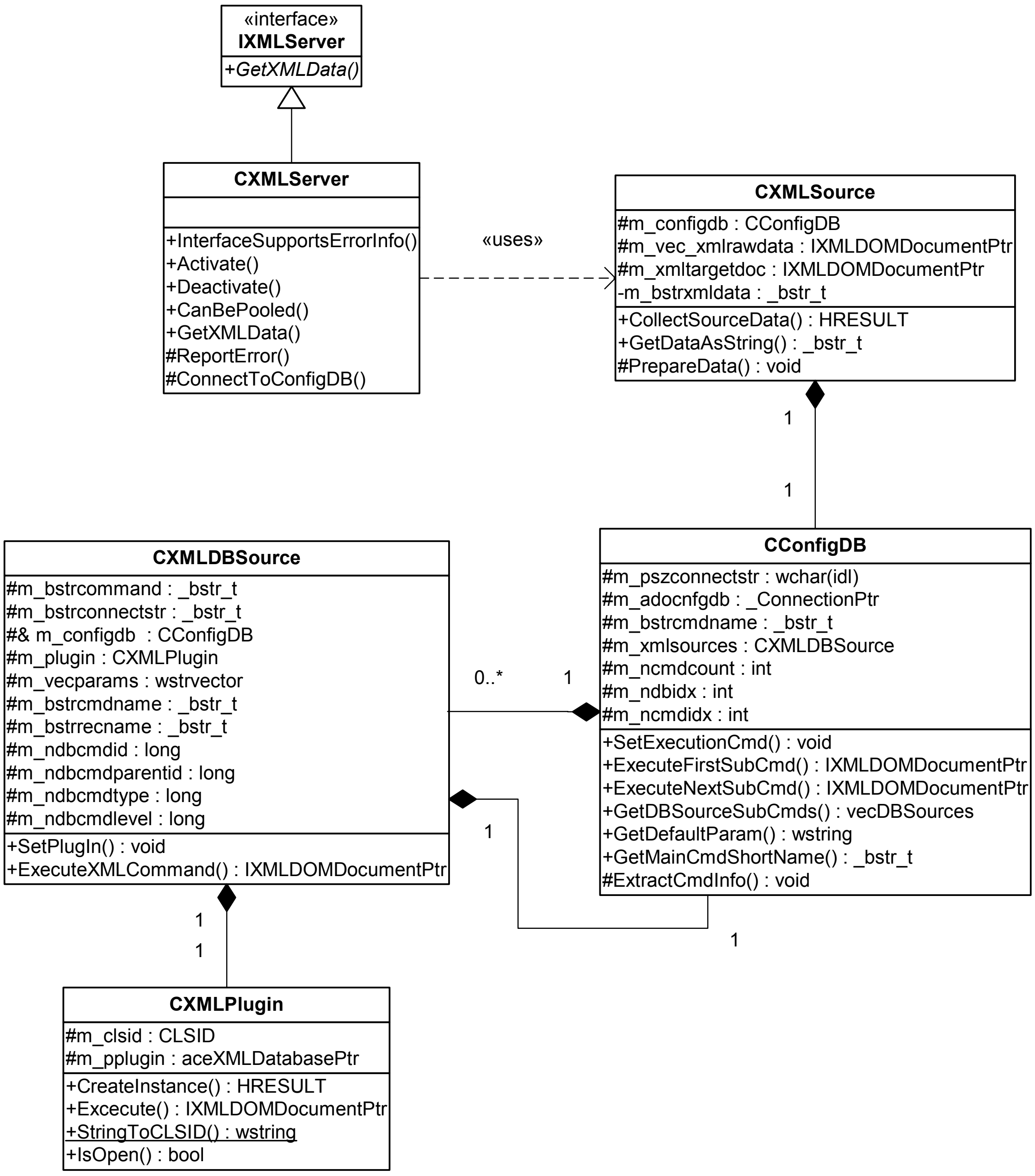

«interface»
IXMLServer
+GetXMLData()
CXMLServer
+InterfaceSupportsErrorInfo()
+Activate()
+Deactivate()
+CanBePooled()
+GetXMLData()
#ReportError()
#ConnectToConfigDB()
«uses»
CXMLSource
#m_configdb : CConfigDB
#m_vec_xmlrawdata : IXMLDOMDocumentPtr
#m_xmltargetdoc : IXMLDOMDocumentPtr
-m_bstrxmldata : _bstr_t
+CollectSourceData() : HRESULT
+GetDataAsString() : _bstr_t
#PrepareData() : void
1
1
CConfigDB
#m_pszconnectstr : wchar(idl)
#m_adocnfgdb : _ConnectionPtr
#m_bstrcmdname : _bstr_t
#m_xmlsources : CXMLDBSource
#m_ncmdcount : int
#m_ndbidx : int
#m_ncmdidx : int
+SetExecutionCmd() : void
+ExecuteFirstSubCmd() : IXMLDOMDocumentPtr
+ExecuteNextSubCmd() : IXMLDOMDocumentPtr
+GetDBSourceSubCmds() : vecDBSources
+GetDefaultParam() : wstring
+GetMainCmdShortName() : _bstr_t
#ExtractCmdInfo() : void
CXMLDBSource
#m_bstrcommand : _bstr_t
#m_bstrconnectstr : _bstr_t
#& m_configdb : CConfigDB
#m_plugin : CXMLPlugin
#m_vecparams : wstrvector
#m_bstrcmdname : _bstr_t
#m_bstrrecname : _bstr_t
#m_ndbcmdid : long
#m_ndbcmdparentid : long
#m_ndbcmdtype : long
#m_ndbcmdlevel : long
+SetPlugIn() : void
+ExecuteXMLCommand() : IXMLDOMDocumentPtr
0..*
1
1
1
1
1
CXMLPlugin
#m_clsid : CLSID
#m_pplugin : aceXMLDatabasePtr
+CreateInstance() : HRESULT
+Excecute() : IXMLDOMDocumentPtr
+StringToCLSID() : wstring
+IsOpen() : bool

Figure 3.4.2-1 XML Server UML Model

CXMLSource is used by *CXMLServer* to establish a connection to the different XML sources and to collect the multiple XML strings. It holds a member of *CConfigDB* to extract the configuration data from the configuration database. *CXMLSource* also starts the execution of all root XML Sub Commands associated with the given main command.

CConfigDB then instantiates a *CXMLDBSource* object for each XML command. These objects receive a *CConfigDB* class as a reference member. This allows the CXMLDBSource objects to start a recursion, which is necessary to navigate through the whole command tree. *CXMLPlugin* is a wrapper class that capsules the COM plugin associated with each specific datasource.

Lets have a look at the XML Server entry point, the *GetXMLData* method of *CXMLServer*. Commentary has been removed to shorten text.

```
STDMETHODIMP CXMLServer::GetXMLData(BSTR bstrcommand,
                                    long returntype, BSTR bstrxmlprefix, BSTR *pbstrxml)
{
        HRESULT hr(S_OK);
        IObjectContext pobjectContext = NULL;
        hr = GetObjectContext( &pobjectContext );
        try {
                // Instantiate CXMLSource and write result data to pbstrxml
                /*...*/
        }
        catch( _com_error &e )   {
                Error( LPCTSTR((e.Description().length()) ? e.Description() :
                                e.ErrorMessage()), e.GUID(), e.Error() );
                hr = e.Error();
        }
        catch( aceXMLException &e ) {
                hr = e.GetHResult();
                ReportError( _bstr_t(e.what()), e.GetGUID() );
        }
        catch( ... ) {
                hr = RPC_E_SERVERFAULT;
        }
        if ( pobjectContext )
        {
                if ( FAILED(hr) )
                        hr = pobjectContext->SetAbort();
                else
                        hr = pobjectContext->SetComplete();
                pobjectContext->Release();
        }
        return hr;
}
```

As anyone can see above the XML Server uses exception handling for error reporting. Exception handling is a common mechanism in C++ to transfer errors. It helps to improve the readiness of source code, as the developer is not forced to write error-handling code at many different positions in the project[9].

[9] For detailed information about how to use exception handling I recommend [15].

The XML Server deals with different kinds of exceptions. First there are the *_com_error* exceptions. This is a Microsoft-specific exception used by *smart_ptr_t*. This exception is thrown to report errors from client COM instances. This is a very powerful class as it also includes functionality to transport rich error information supplied by *IErrorInfo*.

Then there is the aceXMLException. This is a self-defined class to transport error information from inside the XML Server algorithm to a central error handling routine. That means to this exception handler in the XML Server entry method.

There is also a default exception handler included, which catches all other exceptions. In theory this should never happen. But it is difficult to predict that there will never ever be an invalid pointer or something like that.

3.4.3 The Plugins

The plugins are the only communication interface to the different data sources outside of the XML Server. The plugins have to be COM+ objects that have themselves registered in the windows registry. The CLSID of this plugins have to be entered into the server configuration. This CLSID may then be linked to any desired XML Sub Command. During command execution the server reads this CLSID from the configuration database and instantiates the corresponding plugin object.

3.4.3.1 Plugin Interface

The plugins have to support a few specific interfaces to allow communication with the server (Figure 3.4.3-1). The *IUnknown*, *IDispatch*, *IObjectControl* and *ISupportErrorInfo* interface are provided for the same purpose as in the server object.

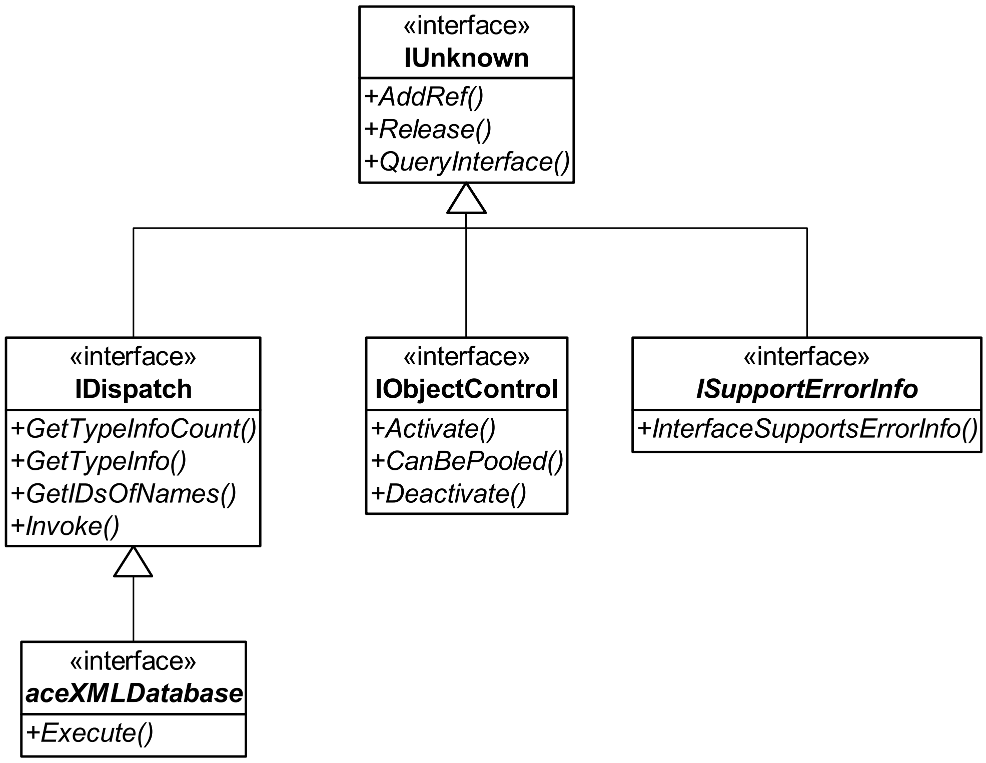
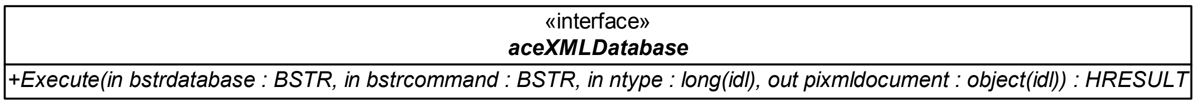


Figure 3.4.3-1 Required XML Server Plugin Interfaces

The main communication interface between the server and the plugins is *aceXMLDatabase*. It currently contains only one method named *Execute*. This method has four parameters shown in Figure 3.4.3-2.



```
+Execute(in bstrdatabase : BSTR, in bstrcommand : BSTR, in ntype : long(idl), out pixmldocument : object(idl)) : HRESULT
```

Figure 3.4.3-2 The aceXMLDatabase Interface

- **bstrdatabase**: Allows the plugin to identify the target database
- **bstrcommand**: A plugin specific command to specify the data to retrieve
- **ntype**: Another plugin specific parameter to give more detailed information of the command type.
- **object:** This is the return parameter. If the execution of the command is a success, this value holds the XML Data in form of the Microsoft XMLDOM Document. It is also possible that is parameter itself holds an XML return stream.

3.4.3.2 Plugin Internals

As a result of the binary encapsulation with COM interfaces the plugins may have any internal structure as long as the interface stays the same. But in order to maintain stability, scalability and data consistency the plugin implementations should follow a few guidelines.

- **Transactions** This is maybe the most important one. It doesn't cost much effort to tell the MTS the current transaction status, but it makes life a lot easier when the data has to be kept consistent.
- **Rich Error Information** The plugin should support the error mechanism shipped with COM. This error messages are forwarded to the server and then to the user or administrator. This makes using the server a lot easier because a message text like "Invalid attribute definition" is more usable then an error code or the default server error message "unspecified error".
- **Stateless Objects** Building stateless classes greatly enhances the server performance when accessed from multiple clients as it is not required to create as many plugin instances as client connections. This reduces main memory usage and increases scalability.
- **Default Exception Handler** Exceptions are not allowed to cross COM interface borders. Therefore it is advisable to implement a default exception handler (*catch(...)*). Otherwise the only information the XML Server receives is from the Service Control Manager that an exception has occurred in one of the COM plugins, but no additional information is available.

3.4.3.3 The Fabasoft Components Plugin

Two plugins have been developed during the XML Server project. One for gaining access to data sources supporting OLEDB and ODBC and another one to another middle tier application called Fabasoft Components (FSC). Developing the OLEDB Plugin was an easy task because OLEDB sources return data in a table. This table can easily be displayed as an XML result set. But he FSC kernel represents data not in tables. Instead it directly exposes objects with members and methods. These objects can be heavily linked with each other through FSC attributes like object references, back links, aggregates etc.

FSC has its own query language to retrieve objects from its object store. These language is similar to SQL but has a little different syntax and FSC specific extensions. For example to get all WinWord documents containing the word "invoice" in its subject the query must look like this.

```
SELECT COOSYSTEM@1.1:objname
FROM COOMSOFFICE@1.1:WinWordObject
WHERE COOSYSTEM@1.1:objname LIKE "*invoice*"
```

The result is a list of COM object pointers that match the above criteria. The result is not a result table like that from a relational database. Instead it is a list of objects with properties and methods.

Figure 3.4.3-3 shows the plugin in more detail. The plugin implementation class is derived from the *aceXMLDatabase* interface. This is the server/plugin communication interface as described above. Another problem is the allocation of memory from each FSC kernel. Each one allocates at least 50 Mbytes of memory, only for startup.

Therefore a developer must be careful. During the development process the programmer tests his software only with one XML Command at the same time and may not recognize what happens if five, ten or more commands are processed at the same time. This can easily put the server to its knees and makes a problem for scalability.

As anyone can see the *FabasoftComponents* member of *CXMLFSCPlugin* is declared static. This means only one instance is created regardless of the number or plugin instances. To do so, this usually requires to implement special safety mechanism to ensure data safety for working with multiple threads. But in this case a COM Interface also encapsulates the target data[10]. COM objects are by default thread safe; every COM object must ensure thread safety by default[11]. Thus a special thread synchronization mechanism for accessing shared data from FSC is not necessary.

[10] Fabasoft Componentes also uses COM as primary communication interface.

[11] As described in chapter 2.4.4 COM Threading.

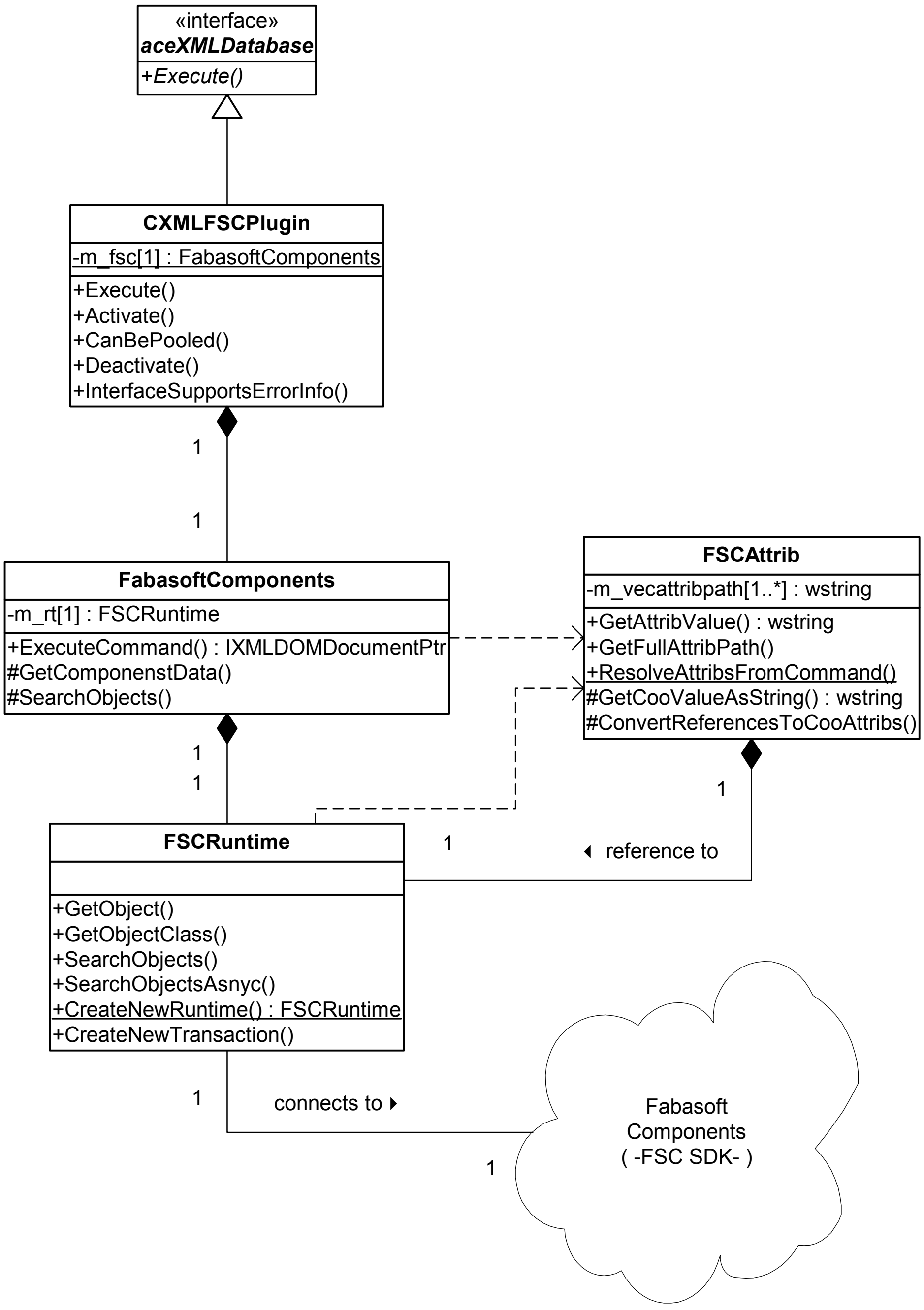


Figure 3.4.3-3 FSC Plugin Internal Structure

The diagram above implies the next steps in gathering the required information. The *FabasoftComponents* class uses the *FSCRuntime* object to instantiate the Components kernel and retrieves the data from the FSC database. The *FSCAttrib* class instances are working a translation objects. This class is capable of translating the requited object and attribute values from FSC into strings that can represent this values in an XML stream.

3.4.4 Parsing the XML Commands

As told in 3.3.4 XML Sub Commands may contain any number of additional parameter arguments. The XML Server parses the plugin specific command line and is searching for the parsing argument character '%'. The next line shows a possible database command.

```
SELECT
    EmployeeID, LastName, FirstName, Address
FROM
    Employees
WHERE
    LastName like '%(0,1)'
```

Then the parser reads the parameter level and index and replaces the whole argument with the appropriate data. For example, assume the XML Main Command name is Northwind[12]. The XML Server is called with the command `"Nortwind D%"`. The plugin would receive the following database command:

```
SELECT
    EmployeeID, LastName, FirstName, Address
FROM
    Employees
WHERE
    LastName like 'D%'
```

This may result in the next XML Server output:

```
<Northwind>
  <Employees>
    <Employee>
      <EmployeeID>1</ EmployeeID>
      <LastName>Davolio</LastName>
      <FirstName>Nancy</FirstName>
      <Address>507 - 20th Ave. E.Apt. 2A</Address>
    </Employee>
    <Employee>
      <EmployeeID>9</EmployeeID>
      <LastName>Dodsworth</LastName>
      <FirstName>Anne</FirstName>
      <Address>7 Houndstooth Rd.</Address>
    </Employee>
  </Employees>
</Northwind>
```

To send a single '%' character to the plugin the database command string must write '%%'. This sequence is replaced with single '%'.

[12] The Nortwind database is an example database included with the Microsoft SQL Server.

3.4.5 Merging the XML Information

This parameter mechanism can now be used to link different result-sets together. As an example a Subcommand is added to the command used above:

```
SELECT
    TerritoryDescription
FROM
    EmployeeTerritories as et
    INNER JOIN Territories as t ON
         et.TerritoryID = t.TerritoryID
WHERE
    et.EmployeeID = %(1,1)
```

This Sub Command now asks the server for the value with command level and index one. As anyone can see this is the *EmployeeID* from the *Employee* table. In each command of the result tree, this parameter is replace with the corresponding value. The returned XML stream looks like this:

```
<Northwind>
  <Employees>
    <Employee>
      <EmployeeID>1</ EmployeeID>
      <LastName>Davolio</LastName>
      <FirstName>Nancy</FirstName>
      <Address>507 - 20th Ave. E.Apt. 2A</Address>
      <Territories>
        <Territory>Wilton</Territory>
        <Territory>Neward</Territory>
      </Territories>
    </Employee>
    <Employee>
      <EmployeeID>9</EmployeeID>
      <LastName>Dodsworth</LastName>
      <FirstName>Anne</FirstName>
      <Address>7 Houndstooth Rd.</Address>
      <Territories>
        <Territory>Hollis</Territory>
        <Territory>Portsmouth</Territory>
        <Territory>Southfield</Territory>
        <Territory>Troy</Territory>
        <Territory>Bloomfield Hills</Territory>
        <Territory>Roseville</Territory>
        <Territory>Minneapolis</Territory>
      </Territories>
    </Employee>
  </Employees>
</Northwind>
```

Every employee has now additional sub tags containing the territories each person is responsible for.

3.4.6 Communication With The Plugins

The preferred communication between the XML Server plugin is by passing a COM reference pointer of *IXMLDOMDocument*. This is an interface published by the Microsoft MSXML type library. This library is included with the Microsoft Internet Explorer since Version 5.0. This means the library is free and already installed if using Windows 2000.

The XML Server prefers a reference to this type of object because this object and library is also used internally to browse and modify XML contents.

As an alternative the plugin may also send the database command result as an BSTR to the server. This may be useful if the plugin resides on another machine where the MSXML library is not available. The XML Server then sends this XML string to the *XMLDOMDocument,* which automatically parses the string and reports if it is not well formed. Which means it analyses the XML stream and reports possible syntactical errors. This new XML document object is then used for further internal operations.

3.5 *SERVER CONFIGURATION*

3.5.1 Configuration Database Design

The configuration of the XML Server is stored in a relational database. In theory this could be any relational database that supports SQL queries. During the project the Microsoft SQL Server 7.0 and SQL Server 2000 were used. Figure 3.5.1-1 shows the internal organization of the configuration database tables.

The *Command* table stores all Main XML Commands of the server. This table is linked over an intermediate table with *DBCommands*. This *DBCommands* table is responsible for holding information about all XML Sub Commands. Also the *DBCmdParameter* table refers to *DBCommands*. *DBCmdParamater* contains the default parsing parameters for the commands that will be forwarded to the different data sources. *DataSource* and *Plugin* are tables, which store the available data sources and their plugin. Two tables are used for saving this information to allow the administrator to register many different data sources that are of the same type therefore may use the same plugin for data retrieval.

The connection to this configuration database is established with the Microsoft Active Data Object (ADO) provided by the Microsoft Data Access Components (MSDAC). This allows access via the OLEDB interface of the SQL Server. But if someone wants to store the configuration information in an Oracle Database and wants access via ODBC no changes to the source code would have to be made.

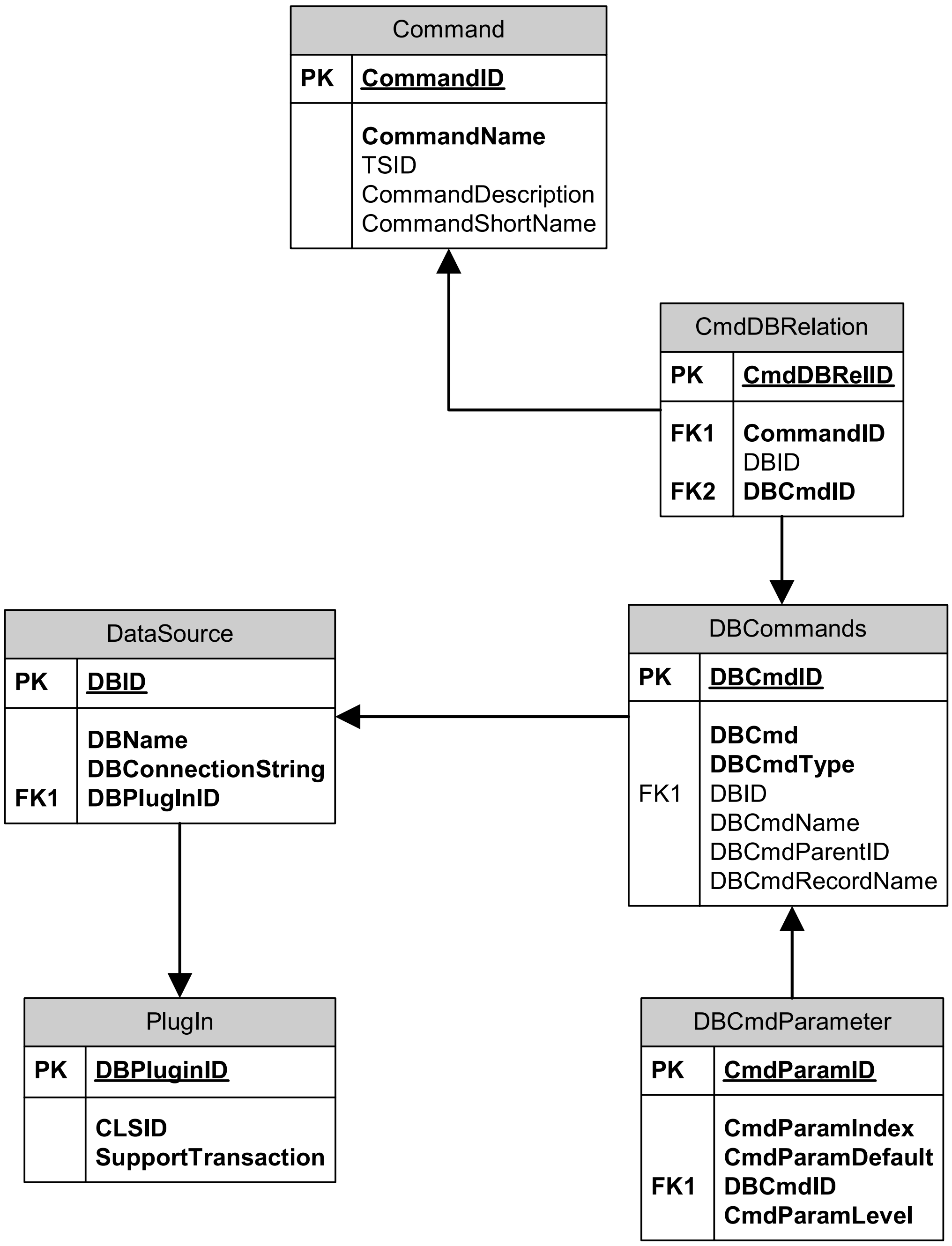


Figure 3.5.1-1 Configuration Database

As can be seen in Figure 3.5.1-1 the configuration data is heavily connected with each other through foreign keys. Configuring the server on the database through editing the tables directly or using SQL commands would ask too much from an average administrator. These tables are also invalid in showing an overview of the commands, as they are structured in trees. This tree view is completely lost in this database.

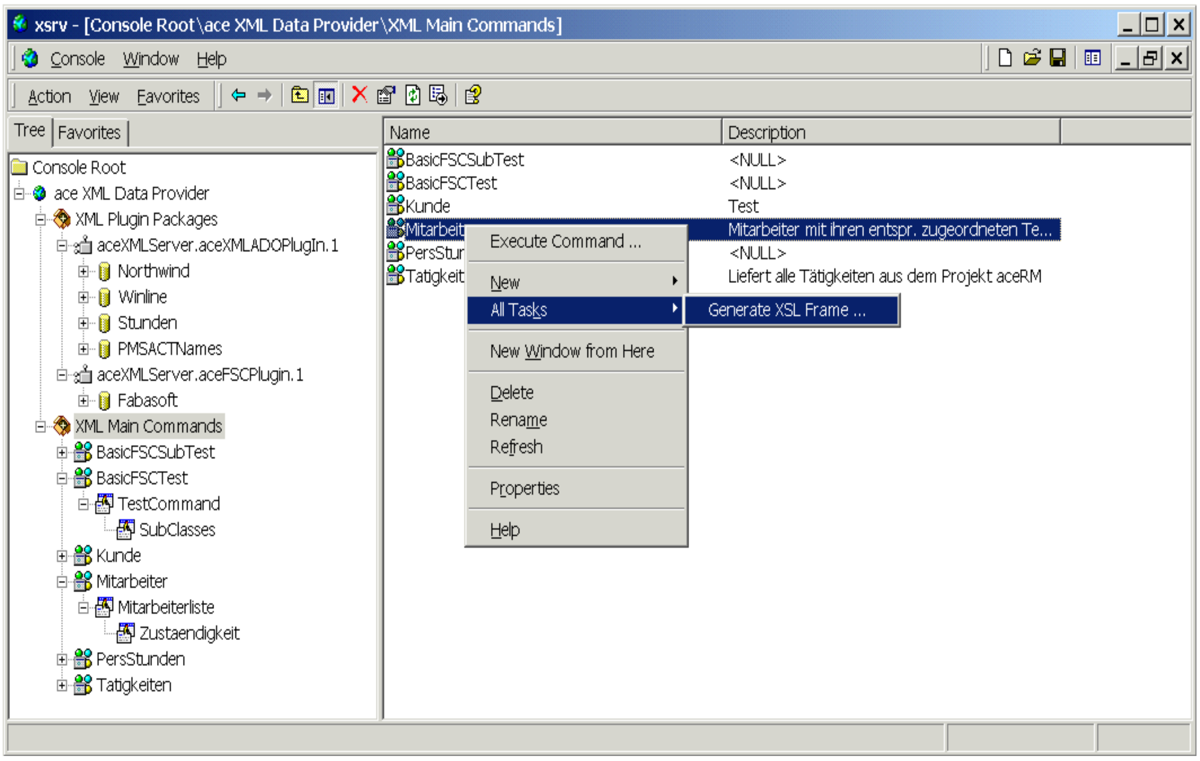
As a result an own administration tool was developed to simplify the administration of the configuration database and provide a quick an logical overview of the available command structure.

3.5.2 The Microsoft Management Console (MMC)

The MMC is a common tool that is used to administrate all of Microsoft's Server Applications. The latest versions of Internet Information Server, SQL Server, and Index Server are all managed using the MMC. Therefore, to provide a consistent administration tool, to which current administrators are familiar with, the MMC was chosen as the configuration environment for the XML Server.

3.5.2.1 Snapin Overview

To handle the data in the database and to give the server administrator a quick overview about the registered server plugins, the connected databases and the available XML commands a configuration tool was developed to display this information.



Name	Description
BasicFSCSubTest	<NULL>
BasicFSCTest	<NULL>
Kunde	Test
Mitarbeit	Mitarbeiter mit ihren entspr. zugeordneten Te...
PersStur	<NULL>
Tatigkeit	Liefert alle Tätigkeiten aus dem Projekt aceRM

Figure 3.5.2-1 XML Server Configuration Tool

As one can see in the figure above the tool was implemented as a Microsoft Management Console (MMC) snapin. The MMC is the new standard configuration instrument for administration purposes. In MS Windows 2000 the MMC takes nearly all administration tasks of the windows system control. This has the advantage that the system administrators is not confronted with as many different user interfaces as the network has programs installed.

As shown in Figure 3.5.2-1 the first major snapin branch displays the currently installed server plugins with their registered data sources. The second one gives the administrator the possibility to add new main commands with subcommands, manipulate these, define default command parameter and execute the commands to test the XML commands.

Every record of the configuration database has a corresponding representation in the MMC. This allows an easy and comfortable way of changing the XML Servers configuration data. This includes the following elements:

- Available plugins for the server.
- Register new or delete data-sources corresponding to each plugin
- Edit each XML Command which can be executed on each data source.
- Add and Remove XML Main Commands including their subcommands.
- Add and Remove XML Subcommands from their parent commands.
- Edit XML Subcommand specifications like the command string or the data store on which the command is executed.
- Specify default values for each XML command.

3.5.2.2 MMC Internals

Programming an MMC Snapin is not an easy task. This is because of the MMC's data organization, shown in Figure 3.5.2-2.

The MMC is divided into two different panes, the result and the scope pane. Each pane may contain any amount of items. The items in the scope pane are organized as a tree. The items in the result view are flat but may contain additional data columns.

The difficulty is that these items have no relation to each other. The scope items are not linked in any way to the result items. This may be helpful if one wants to display data in the result pane which is only loosely dependent of the current scope item. It is for example possible to display any HTML document in the result pane.

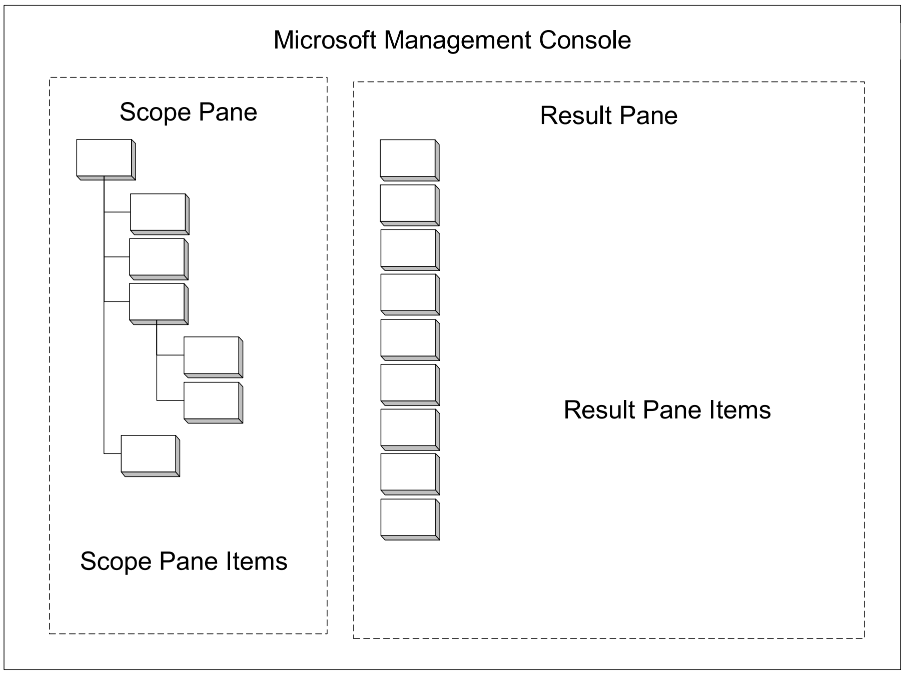


Figure 3.5.2-2 MMC Object Organization

On the other hand usually the developer needs the items within a close relation because when selecting an item in the tree view the user expects the result view to change like in the windows file explorer. And if the Tree is expanded the user estimates to see the result items now as a part of the tree in the scope pane.

This was one of the many reasons why the configuration tool of the XML Server took nearly as long to develop as the server itself.

3.5.2.3 Snapin Internal Implementation

Figure 3.5.2-3 shows a brief overview about the internal class organization. Much information is hidden to make the diagram more readable. The heart of the snapin is the *CMMCBaseItem* template. Every item displayed in the MMC is derived from this template class, except the root item of the configuration tree.

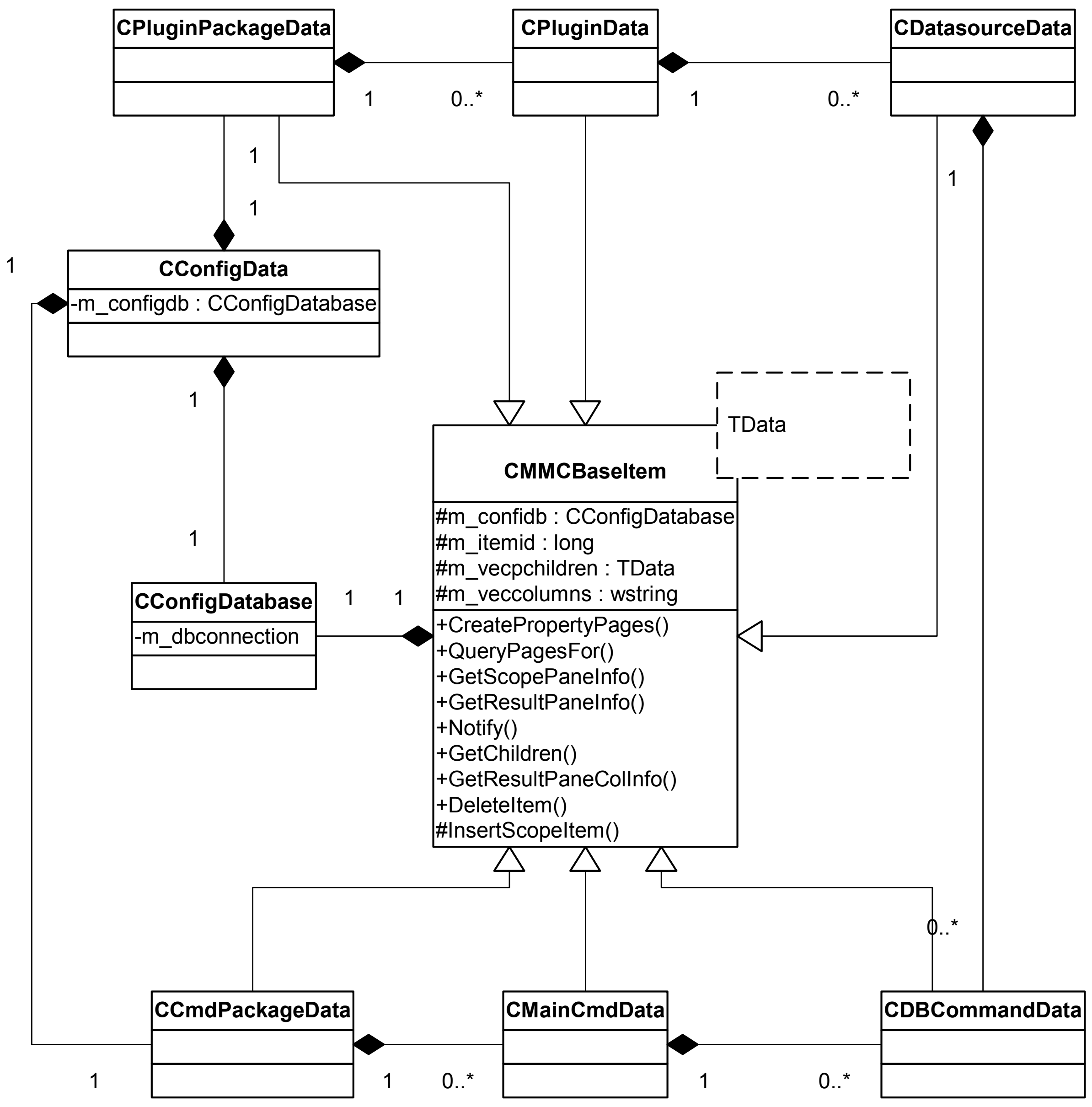


Figure 3.5.2-3 Snapin Internal Structure Overview

The MMC wants every item, which is put into one of the views to expose a specific COM Interface, *IResultData* for result items and *IComponentData* for the scope items. The *CMMCBaseItem* exposes both of these two interfaces at the same time, but hides this from all other classes. Building a tree where the result pane contains the child items of the corresponding scope item becomes now fairly simple. All that *CMMCBaseItem*-derived classes have to do, is to implement the *GetChildren* method. The natural polymorphism of all C++ objects takes care, that the correct implementation method is called.

The figure above shows the final structure overview of the configuration snapin. *CConfigData* represents the root element. This is also the class that holds the only instance to the Configuration Database object *CConfigDatabase*. All other classes receive only a reference to this class instance. Then the tree is split into two branches. One branch that holds the plugin and data source structure and another one, which is responsible for the management of the XML Main Commands and XML Sub Commands.

3.6 THE SERVER CLIENTS

Writing a client is extremely simple, because all Microsoft development tools support COM objects and allow easy usage.

3.6.1 C++ Client

A C++ client is an easy task if using the Microsoft Visual Studio because it creates wrapper classes for the COM interface pointer with a simple link to the type library. This is done with the keyword *#import*.

Using #import informs the compiler to create wrapper classes of derived of _com_ptr_t which uses exceptions as error reporting mechanism. This simplifies the error handling as the error routine can easily be placed on the end of the program without having the developer to worry about any invalid method results.

The source code below shows such a possible client.

```
#include <iostream>
#import "aceXMLServer.tlb" no_namespace named_guids

void main()
{
        CoInitialize();          // init COM apartment

        try
        {
                // create an instance of the xmlserver
                aceXMLServer::IXMLServerPtr xmlserver;
                xmlserver.CreateInstance( CLSID_XMLServer );
```

```
            // execute command and write output to stdout
            std::cout << xmlserver->GetXMLData( "Nortwind", 0, "" );
            std::cout  << std::endl;
        }

        // error hanling
        catch ( _com_error &e )
        {
            std::cout << "Error: " << e.Description() << std::endl;
        }

        CoUninitialize();
    }
```

3.6.2 Visual Basic Client

A Visual Basic (VB) client becomes even smaller. Just add the "aceXMLServer Library" to the project references. It then becomes possible to instantiate the server.

```
Private Sub Form_Load()

    Dim server As New aceXMLServer.xmlserver

    strXML = server.GetXMLData("Northwind", 0, "")
    Print strXML

End Sub
```

By default all errors reported by the object are handled by the VB runtime environment which displays the error message directly to the user.

3.6.3 Client Results

These two clients are now accessing the server and requesting XML Data.. The client wants to access information from the Northwind Database. The client wants to know all employees and the territory each employee is responsible of.

This XML command contains 2 subcommands

The first one is

```
SELECT
    EmployeeID, LastName, Address, Title
FROM
    Employees
WHERE
    LastName like '%(0,1)'
```

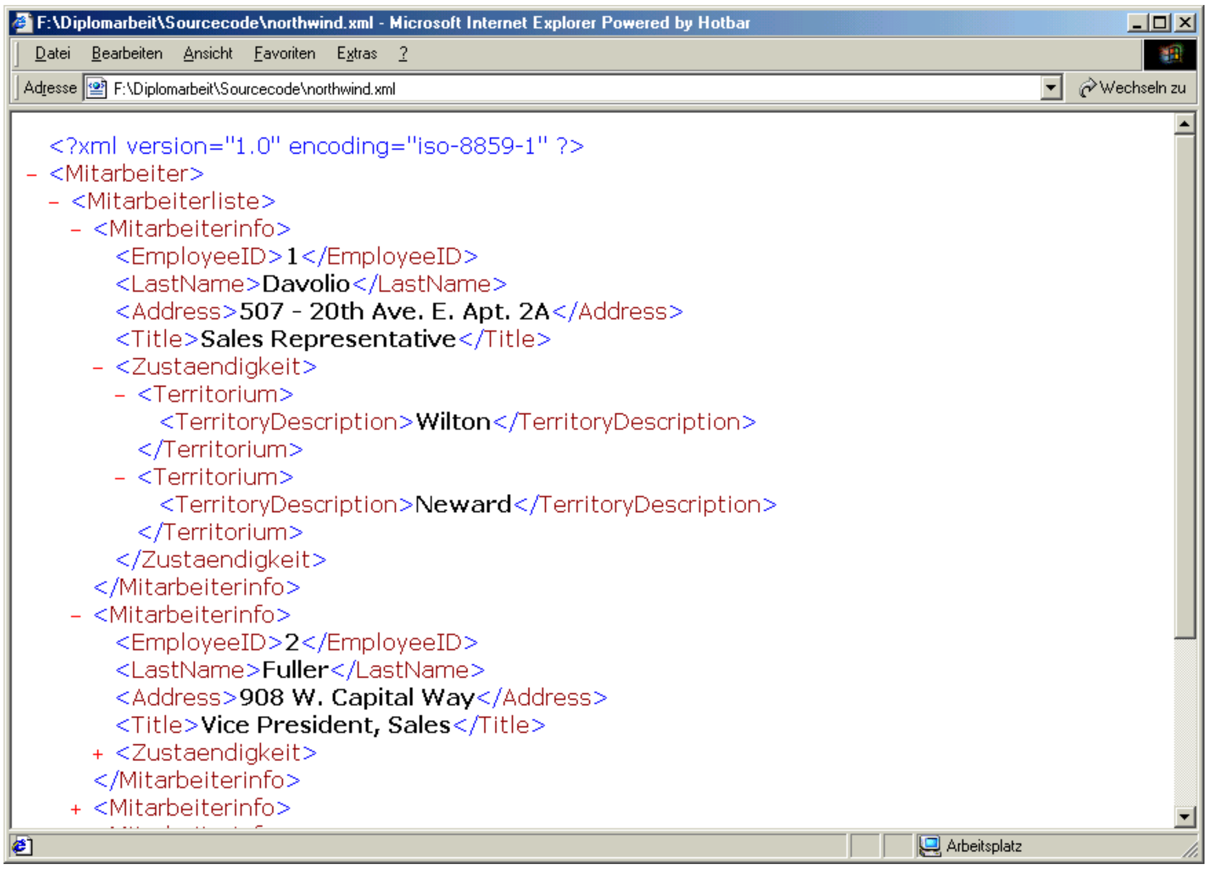
Containing another subcommand

```
SELECT
    TerritoryDescription
FROM
    Territories
WHERE
    EmployeeID = %(1,1)
```

Assume that the server administrator has assigned the character '%' as default parameter for the main command. This means if no main parameters are supported "`%(0,1)`" is replaced by '%'.

The "`%(1,1)`" part of the second command means: Take the result of the command in the first command level and the first result index. During the execution runtime the XML Server parses this command line and replaces this section with the corresponding value.

The server then displays the following XML output.


F:\Diplomarbeit\Sourcecode\northwind.xml - Microsoft Internet Explorer Powered by Hotbar
Datei Bearbeiten Ansicht Favoriten Extras ?
Adresse F:\Diplomarbeit\Sourcecode\northwind.xml
Wechseln zu
Arbeitsplatz

```
  <?xml version="1.0" encoding="iso-8859-1" ?>
- <Mitarbeiter>
  - <Mitarbeiterliste>
    - <Mitarbeiterinfo>
        <EmployeeID>1</EmployeeID>
        <LastName>Davolio</LastName>
        <Address>507 - 20th Ave. E. Apt. 2A</Address>
        <Title>Sales Representative</Title>
      - <Zustaendigkeit>
        - <Territorium>
            <TerritoryDescription>Wilton</TerritoryDescription>
          </Territorium>
        - <Territorium>
            <TerritoryDescription>Neward</TerritoryDescription>
          </Territorium>
        </Zustaendigkeit>
      </Mitarbeiterinfo>
    - <Mitarbeiterinfo>
        <EmployeeID>2</EmployeeID>
        <LastName>Fuller</LastName>
        <Address>908 W. Capital Way</Address>
        <Title>Vice President, Sales</Title>
      + <Zustaendigkeit>
      </Mitarbeiterinfo>
    + <Mitarbeiterinfo>
```

Figure 3.6.3-1 Northwind Result Set

Note that the XML Server has not only returned an XML file. It has also transformed the data stored in linear tables of the Northwind database in an information tree with a tree-depth of 4 Layers (not including the root one).

4 CONCLUSION

The XML Server is a very powerful tool to collect data from multiple data sources. But it is not a replacement of functionality of the used data stores.

For example a customer wanted to have a full text document search over several data sources. During the design phase at the beginning of the project several terms where specified:

- The result should be returned as XML to allow easy representation in a Web Browser. No problem, XML is the natural language of the server.
- The communication should follow transaction rules. Again no problem, the XML Server uses the COM+ transaction services for transaction safety.
- An average skilled administrator must be able to control and supervise the server. The XML server ships with an MMC snapin that allows the administrator full control of the server in a clear structure.
- The full text search has to include support for the Microsoft SQL Server, Fabasoft Components, ordinary files in the file directory and Oracle databases.

The last point is the tricky one. SQL Server has implemented an internal full text search engine. Also a full text search with OLEDB access can be build for standard files on a hard disk using the Microsoft Index Server. Fabasoft also stores its documents as ordinary files on the server machine that can also be indexed by the Microsoft Index Server. The problem results, as this customer has no license for full text indexing of an Oracle database. This means the XML Server cannot retrieve results from a data source that is not capable of providing the appropriate data. Therefore we had to build a new search engine from scratch to support full text searching for all of these data providers.

On the other hand the XML Server has many advantages and supports some interesting features.

- **XML** The usage of XML as data format provides a flexible way of transporting many different kinds of data.
- **COM** The XML Server provides a COM interface, which enables a number of languages to integrate this server to its data access layer.
- **Transactions** All Commands are handled in transactions. The usage of the MTS enables the server to work with multiple data sources in distributed transactions.
- **Scalability** The server and the currently provided plugins support new COM+ features like Just in Time Activation and Object Pooling. This results in quicker object access and less memory usage. Therefore a higher number of clients may use the server simultaneously.
- **Easy Administration** With the MMC as an established management tool to which administrators are already familiar with, an MMC snapin was developed to provide an easy way of administrating this server.

5 BIBLIOGRAPHY

[1] C++ Programming Language, The; Special Edition — Bjarne Stroustrup
Addison Wesley, 2000 — ISBN 0-201-70073-5

[2] Unified Modeling Language, The — Rumbaugh, Jacobson, Booch
Addison Wesley, 1999 — ISBN 0-201-30998-X

[3] Distributed Applications with Visual C++ 6.0 — Scott F.Wilson
Microsoft Press, 2000 — ISBN 0-7356-0926-8

[4] COM and COM+ Programming Primer, The — Alan Gordon
Prentice Hall PTR, 2000 — ISBN 0-13-085032-2

[5] Essential COM — Don Box
Addison Wesley, 1998 — ISBN 0-201-63446-5

[6] COM Programming by Example — John E. Swanke
Publisher Group West, 2000 — ISBN 1-929629-03-6

[7] Inside ATL — George Sheperd, Brad King
Microsoft Press, 1999 — ISBN 3-86063-463-1

[8] Inside Visual C++ — David J.Kruglinski
Microsoft Press — ISBN 1-57231-565-2

[9] Windows Programmierung 5. Auflage — Charles Petzold
Microsoft Press, 1999 — ISBN 3-86063-487-9

[10] C++ Standard Library, The — Nicolai M. Josuttis
Addison Wesley, 1999 — ISBN 0-201-37926-0

[11] XML The Annotated Specification — Bob DuCharme
Prentice Hall PTR, 1999 — ISBN 0-13-082676-6

[12] MSDN Subscription Library — Microsoft Corporation
April 2001 — CD 0401 Part No. X08-3708

[13] Inside Distributed COM — Guy Eddon, Henry Eddon
Microsoft Press, 1998 — ISBN 3-86063-459-3

[14] Unified Software Development Process, The — Jacobson, Booch, Rumbaugh
Addison Wesley, 1999 — ISBN 0-201-57169-2

[15] Exceptional C++ — Herb Sutter
Addison Wesley, 2000 — ISBN 0-201-61562-2

[16] UML Konzentriert — Martin Fowler, Kendall Scott
Addison Wesley, 1998 — ISBN 3-8273-1329-5

6 APPENDIX

6.1 PROJECT SUMMARY

This XML Server was a one-person project and has cost me a lot of work and resources, so here is a little statistic about the project (not included is the work on this diploma thesis). It took me →

9283	lines of code (not included headers and self-written commonly used classes)
742	hours of work (official counting)
428	online minutes with a 56.6kBit modem at home
145	cups of coffee
36	bottles of beer (0.5l)
14	Puten Rio Grande mit Pommes & 2xSalat (Schnitzelhaus)
3	text markers
2	hours in a bookstore in New York to find one of the used books ([6])
1	Windows 2000 Server installation.
1	keyboard
1	mouse

→ to write this server but there are still new features in my mind, which will surely find a way into the XML Server.

6.2 SERVER IDL FILES

The XML Server consists of nearly 10,000 lines of code. In exchange of hundreds of pages of C++ source code only the IDL definitions are added.

6.2.1 XML Server

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

[
    object,
    uuid(A3ED42B5-1C11-42CC-87EF-14F12403F8D0),
    dual,
    helpstring("IXMLServer Interface"),
    pointer_default(unique)
]

interface IXMLServer : IDispatch
{
    [id(1), helpstring("Receive XML data from one or
                        more predefined databases.")]
         HRESULT GetXMLData(
                        [in] BSTR bstrcommand,
                        [in] long returntype,
                        [in] BSTR bstrxmlprefix,
                        [out, retval] BSTR *pbstrxml
                        );

    };

[
    uuid(D282855E-D89C-4BB7-B7C5-1FC3793556DF),
    version(1.0),
    helpstring("aceXMLServer Library")
]
library aceXMLServer
```

```
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    [
         uuid(923D64B8-70DB-4F56-A74D-E7BA5CC94444),
         helpstring("XMLServer Class")
    ]
    coclass XMLServer
    {
         [default] interface IXMLServer;
    };
};
```

6.2.2 Plugin Interface

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

[
    object,
    uuid(5B26D37F-474E-11D4-B5C3-00104BE4D4DD),
    pointer_default(unique),
    oleautomation, dual
]
interface aceXMLDatabase : IDispatch
{
    [helpstring("Execute a Command")]
         HRESULT Execute(
              [in]BSTR bstrdatabase,
              [in]BSTR bstrsqlcommand,
              [in]long ncommandtype,
              [out, retval] VARIANT* pixmldocument );
};
```

6.2.3 Administration MMC Snapin

```
import "oaidl.idl";
import "ocidl.idl";
[
    uuid(14E1BB3B-3142-4D6D-AE63-E24D460544C0),
    version(1.0),
    helpstring("ace XML Server Configuration")
]
library aceXMLServer
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    [
         uuid(95A5F955-3EEC-409C-82F8-3BC26925B2C8),
         helpstring("ace XML Server Config Class")
    ]
    coclass Config
    {
         [default] interface IUnknown;
    }
    [
         uuid(D988AA9F-4647-45AE-B7A0-D4638BAC31B6),
         helpstring("Config Class About")
    ]
    coclass ConfigAbout
    {
         [default] interface IUnknown;
    }
};
```