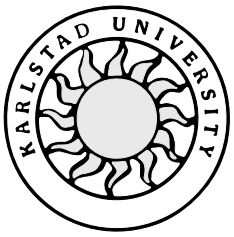

Computer Science

Rickard Holgersson

Compatibility Test: Centura Team Developer vs. Microsoft ActiveX

Bachelor's Project

2000:19

Compatibility Test: Centura Team Developer vs. Microsoft ActiveX

Rickard Holgersson

© 2000 The author(s) and Karlstad University

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Rickard Holgersson

Approved, May 31, 2000

Advisor: Dimitri Ossipov

Examiner: Stefan Lindskog

Abstract

This report presents a method of testing programming tools support for component programming in general, and COM components in particular. The method is then used to evaluate the development environment Centura Team Developer's capacity in this area. This assessment is built up in three steps, where different aspects of component programming are tested.

Contents

List of Figures

List of tables

1 Introduction

This document suggests a method of testing diverse programming tools support for component programming, and shows how it can be used in practice.

So what is component programming? It can be related to conventional programming as building your own stereo versus buying the parts and just plug them together. Because of their well-defined interfaces, components can be used without any demands on the user to know how the components works, or even without having access to the source code.

Section 2 describes the criteria's that should be evaluated for each test, while section 3 describes why they are important. The following section explains the technologies involved in the particular test. Since Centura Team Developer just recently added support for this programming paradigm this will be the tool tested using this method. Section 5 describes what Centura Team Developer is, its parts, and how it is supposed to be used. In the later sections, all three tests are described and evaluated in detail.

2 Criteria for testing

Since this test is meant to generally evaluate the tool's support for component programming, it's important to make the test as complete as possible. It should span over as many aspects as possible, while still being consistent.

The four aspects that I found must be covered is:

1. Integration with the tool
2. Ease of development
3. Ease of maintenance
4. Quality of end product

This is the foundation on which this test stands and I believe that they are included in the items below. Some, like ease of development is covered in many of the items, while quality of end product is mostly covered in 2.5 Performance.

2.1 Development Complexity vs. Component reuse

How many steps are necessary to integrate a component, and how difficult is it to use the same interface but with other components?

2.2 Integration with Legacy Systems vs. Customization

How integrated are the components and how easy is it to customize their behavior?

2.3 Level of Granularity vs. Simplicity

Is it preferable to use this tool with big monolithic components or a set of small ones that can be exported as a framework and used with others?

2.4 Ease of Overview vs. Complexity

Is it possible to show component interfaces and bindings in different views, giving different levels of complexity and overview of them? Is the functionality easy to comprehend?

2.5 Performance

How are the binding of the components created, and does it add something to the performance? Is the component instantiated many times, and how are the interfaces reached?

2.6 Maintenance and Version Handling

What issues need to be handled when you add or remove functionality to components? How does splitting one component into many smaller influence the bindings?

3 Motivation of Tests

3.1 Development Complexity vs. Component Reuse

In a good high-level programming tool it should be fast and easy to build complete applications. Including a component to a project should therefore be a simple and straightforward job and without any demands on the component user to understand the underlying technology. It's much like that you don't need to know how to build a car to drive one, and can use a CD-player even though you have no idea of how a laser works.

At the same time the application must be flexible and easy to change. Replacing a component with another should be simple. Moving logic from the GUI to a component, for example to change it to a three-tier application where presentation, processing, and data is separated, must not be too painful.

This is a bit of a contradiction and it's interesting to see how the development environment balances this.

3.2 Integration with Legacy Systems vs. Customization

To make it as easy as possible to use the components they should be integrated with the development tool so that they look and feel like the other parts of the environment in a consistent way. This makes it easier to use the components and makes the programming tool less complex.

On the other hand, the components will have unique properties that stand out from the rest of the environment. To give the user as much flexibility as possible, these special properties must be exposed to the user.

3.3 Level of Granularity vs. Simplicity

A small very specialized application does not need to be very modularized. As a consequence, the developer may want to make one big component with a complex interface. Big systems on the other hand, should be very modularized, with objects that are easy to replace and re-use. It's virtually impossible to make a language that supports big monolithic applications as easily as object-oriented complex systems.

3.4 Ease of Overview vs. Complexity

It’s vital that you as a component user quickly can comprehend the functionality of the component. To make the development process as fast and simple as possible you should be able to grasp the general idea at a high level of abstraction.

But sometimes you need to get more detailed information of specific parts of a component, especially in the bug-testing phase.

Therefore it’s a crucial aspect of the development environment to give the user a choice of different levels of complexity.

3.5 Performance

In today’s world of distributed applications, performance again becomes an important aspect of programming. For servers to be scalable there must not be too much overhead just to call methods and retrieve and set attributes. Since there’s a number of ways to do that such of thing this becomes a significant part of the test.

3.6 Maintenance and Version Handling

The big software costs lies not in developing new systems. As big an effort that may be, most of the money and resources are used maintaining and updating the system. To be able to change parts of an already functioning application is consequently perhaps the most important part of a development environment.

4 The Technologies

4.1 COM

"The Holy Grail of computing is to be able to put applications together quickly and cheaply from reusable, maintainable code, preferably written by someone else."
The Idea of COM, Julian Templeman, et al [2]

A technology that has become more and more important when developing new software is object oriented programming. Object-oriented programming offers a new model that differs from traditional design, which is based on functions and procedures. The object-oriented programming technology makes it easier to build modules that can be modified and reused. Although this technology has proven to be very useful, it does not address the problems when you want to use different programming language version management and more.

Another way to look at programming is the concept of software components. The big difference between objects and components is that an object is a piece of software source code while a component is an actual working software module. With a precisely defined interface and a supporting system software this technology promises a software market where components can be shared and interchanged without even recompiling the application.

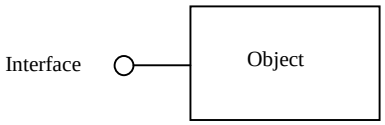

Figure 4.1: A simple object

COM stands for the Component Object Model and tries to deliver that promise. It’s a binary and network standard that allows any two components to communicate regardless of what machine they are running on, what operating systems the machines are running, and what language the components were written in.

COM is based on objects, with all properties you expect from an object (encapsulation, inheritance, polymorphism, etc.). For the user a component is a black box where you don’t know anything about its implementation. Of course, you need some way to interact with the

component and for this interfaces are provided. An interface is a well-defined set of functions that you can call to make the component do something.

Some general way to globally identify a class (object type) and interface is also needed. For this a 16-byte structure called GUID (globally unique identifier) is used. This provides a 2^{128} (ca $3{,}4 * 10^{38}$) sized flat address space.

There are two types of COM servers: in process and out-of-process. In process servers are dynamically linked libraries (DLLs). They live in the client process' address space, which makes them fast. Out-of-process servers are executable files. The main advantage of out-of-process servers is that they are isolated from the client, so that if they crash it won't affect the client.

4.1.1 Users Point of View

To use the component you need to ask the COM system to give you access to one of its interfaces and then you can send and receive the messages needed.

Let's say we have a component named MsgBox with only one interface, IMsgBox that can be used to show simple message boxes on the screen. The user of MsgBox need not concern about how this is implemented. IMsgBox contains one function: Show(), that displays the message and one property: Message which is a string.

An example in VBScript:

```
Set msgBox = CreateObject("MsgBox.Application")
msgBox.Message = "Hello World!"
msgBox.Show()
```

Figure 4.2: Result of Hello World script

4.1.2 Developers Point of View

While using a COM component is simple and straightforward, creating one is a bit more complicated. COM defines a large set of standard interfaces, and all COM components must implement the IUnknown interface.

It contains three methods:

QueryInterface	Gives access to other interfaces
AddRef	Increments the reference counter
Release	Decrements the reference counter

Table 4.1: Methods of interface IUnknown

The reference counter determines the lifetime of a single instance. Each time a client retrieves a reference to an interface of the component the reference counter is incremented. When the client is done, the counter is decreased. This way each COM object can determine when it is no longer needed and delete itself.

You can then add your own interfaces that are expected to inherit from IUnknown. But how should the interfaces be declared so that they become accessible to the component users? To make the COM component language independent one must have some way to express the properties of the interface in a language independent way. COM uses Interface Definition Language (IDL) for this. It's a language with syntax quite similar to C++. An IDL compiler then creates the source files needed.

```
[ uuid(af7f3e40-07d4-11d4-8b29-c6725a356d37) ]
interface IMsgBox : IUnknown
{
  HRESULT Show();
  HRESULT Message([in,  propput] BSTR *pbsMsg);
  HRESULT Message([out, propget] BSTR *pbsMsg);
};
```

Figure 4.3: Interface definition in IDL

An aspect of COM that has been ignored to now is how to create instances of a class. It is important that COM is able to have a standard way of creating objects of any type without requiring the client to know the details of creation. COM therefore uses a class object (also called class factory) which encapsulates this.

The class object is a COM component in its own right. It implements an interface called IClassFactory, which contains two methods: CreateInstance and LockServer. The important method is CreateInstance which, given the class ID, handles the IUnknown interface of the component requested. CreateInstance can be compared with the C++ operator new.

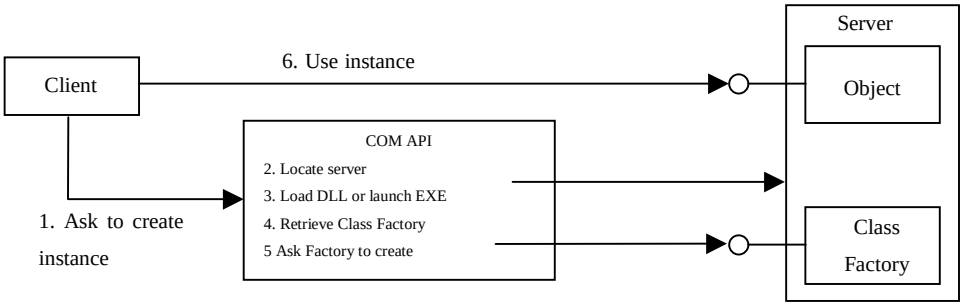

Figure 4.4: Instantiation of a COM component

4.2 Automation

One problem with pure COM is that it does not work well for scripting languages and other programming languages with no concept of vtables. If you look under the hood, a COM interface pointer is a pointer to a location that holds the address of a table of function pointers. But scripting languages does not understand virtual function tables, and therefore some other solution must be made for these languages.

Automation is a mechanism to expose the interfaces for a client so clients can resolve function calls in runtime (late binding). An automation server implements the IDispatch interface.

Invoke	Calls a method or accesses a property
GetIDsOfNames	Returns the ID of a property or a method
GetTypeInfo	Retrieves a pointer to ITypeInfo, if available
GetTypeInfoCount	Returns the number of type info interfaces available (0 or 1)

Table 4.2: Methods of interface IDispatch

Given a pointer to an IDispatch interface, the scripting language can call GetIDsOfNames, and then Invoke. This of course becomes much slower than accessing functions directly, but makes it possible to do quite advanced things with a few lines in a scripting language.

Type libraries accompany most automation servers. They provide information about the interfaces and components of a server. Type libraries can be used in a variety of ways. For example, they can be used to implement dispatch interfaces and to provide information to object browsers. A type library is typically generated from an IDL file.

4.3 DCOM

While COM makes it possible to create software modules that are easy to integrate with other applications, it does not cope with the additional complexity of distributed applications. Distributed COM (DCOM) extends COM with remote method calls, security, scalability, and location transparency. For the communication across the network, DCOM uses an “Object RPC,” extending DCE RPC.

DCOM components are truly location transparent. The client does not need to change any part of its code or even recompile. Some changes in the registry must be made for the underlying COM enabling code to locate the component. DCOM was included in NT 4.0 and Windows 98.

There are implementations of DCOM that runs over other systems than Windows. One example is EntireX DCOM developed by Software AG.

An alternative technology providing the same type of functionality as DCOM, is Common Object Request Broker Architecture (CORBA). CORBA is a standard that was developed by the Object Management Group [4].

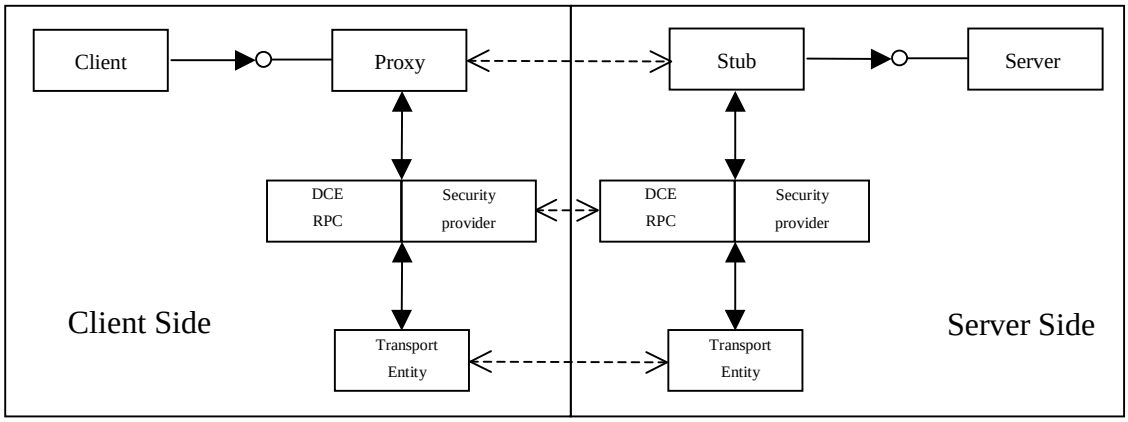

Figure 4.5: DCOM architecture

4.4 ActiveX

ActiveX does not refer to some well-defined technology, instead it’s more like a brand name for a series of objects and technologies based on COM. The history of ActiveX begins with OLE, which is a technology for creating compound documents. It’s successor OLE2 included COM and became more flexible. Shortly after, more technologies took advantage of the COM

architecture. Microsoft therefore chose to call all kinds of COM solutions OLE, but later changed it to ActiveX.

Some examples of technologies calling themselves ActiveX are ActiveX controls, ActiveX Data Objects (ADO), and ActiveX Server Pages (ASP).

4.5 ActiveX Controls

What most people think of if you mention ActiveX are ActiveX controls: in process, local COM components that implement interfaces enabling it to look and feel like a control (for example a button, or text field. There are really no required interfaces to implement, except for IUnknown, but for a control that can be used anywhere (in a browser, in a VB-script, C++ application, etc.), it must have an interface to expose its methods to the outside world, to be in-place activated and more. As the figure shows, a typical ActiveX control is a very complex component and nothing you want to implement from scratch.

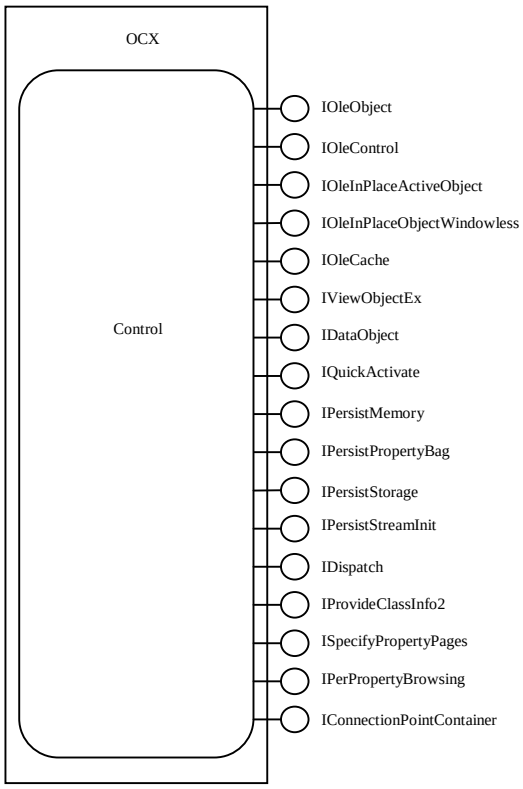

Figure 4.6: A typical ActiveX control

Commonly an ActiveX control has some custom interfaces making it possible for the window it lives in to send messages to it. The control also can send messages to the window in the form of events.

The control needs some context to live in. This ActiveX control container is also a complex COM component, implementing a control site object (one control site object gets instantiated for each control) and takes care of the events the control is sending using a special COM component called an event sink.

4.6 ODBC

ODBC is an industry standard for accessing relation databases. It has become so successful that virtually all databases support the standard. From the programmers point of view it is simply an API that makes database access a bit less demanding. The architecture of ODBC contains four layers. The top layer is the application itself. It makes calls to a dynamically linked library named the Driver Manager. The Driver Manager knows which driver to load and sends the call to that driver. The bottom layer is the data source, which is the combination of the database managing system (DBMS), the operating system, and the network. For more information about ODBC, see Open Database Connectivity Without Compromise, by Kingsley Idehen [3].

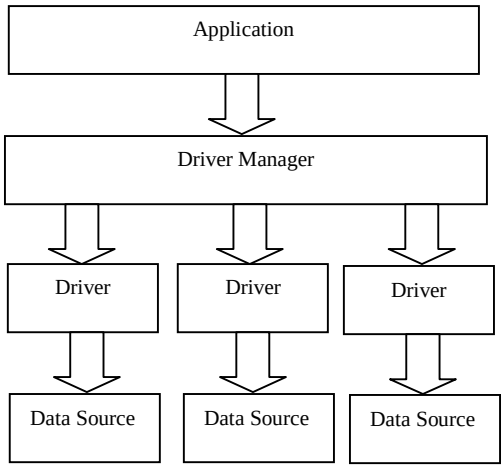

Figure 4.7: ODBC Architecture

5 Centura

5.1 Centura Team Developer

Centura Team Developer (CTD) is a development environment for creating databases and user interfaces to these. There is support to give end users access to a database for example through a web-browser. CTD is not one single tool, but a palette of co-working tools. There are a number of tools to setup and configure database servers, create reports, and debug finished applications. The main tool and the important part of CTD for this essay is SQLWindows. It's a programming environment for creating user interfaces to the databases. It's possible to, within the tool, both make direct database calls and process the data before it's presented to the user.

5.2 SQLWindows

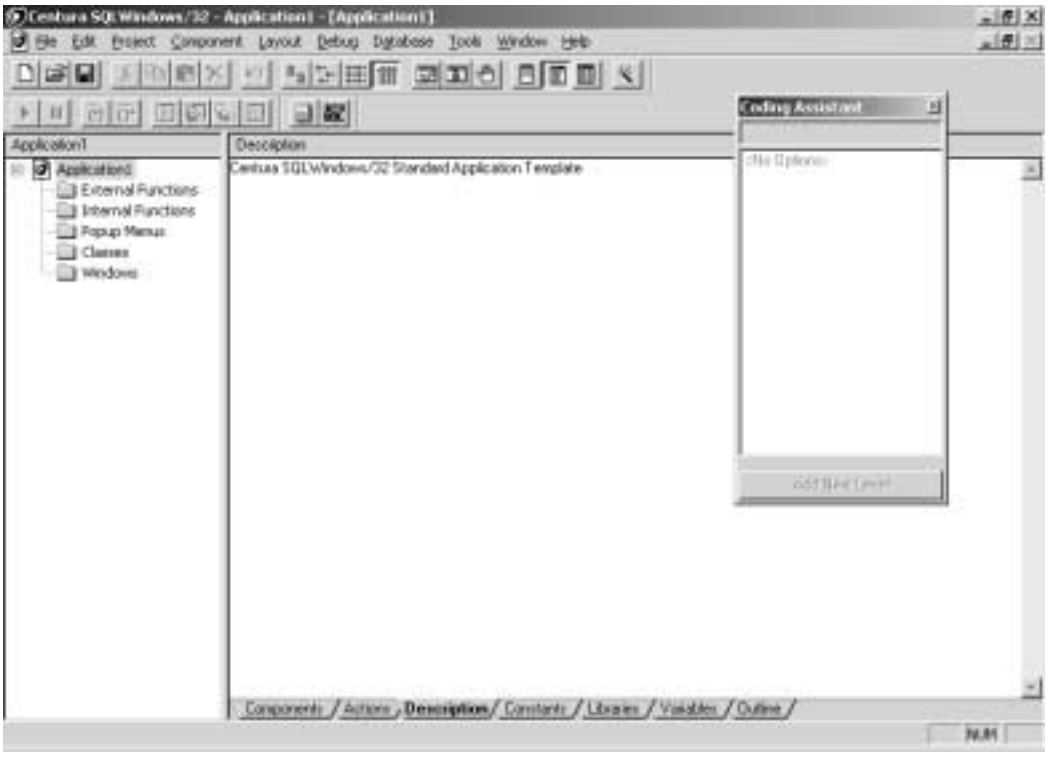

Figure 5.1: Main view of SQLWindows

SQLWindows is a fourth generation language for creating graphical user interfaces (GUIs). These interfaces can be standalone applications or integrated in a web browser.

You start the process of making a SQLWindows application in the layout view. From here you can view the graphical interface of the program. Typically you create a main window (called a form) and drag and drop the controls you need into the window. The controls are collected in a tool called the Controls Palette, and contains various set of controls ranging from conventional items like buttons, to more database-specific like tables with special support to map them to database tables. The attributes of a control can be viewed and edited through a window called Attribute Inspector. There are also a number of wizards to ease the process of making user interfaces.

When you have finished the view of the program you switch to the outline view. Here you can connect your buttons and text fields with code that actually does something. To your help there is a window called the Coding Assistant which shows all options at any given point in the code. Most of the coding is event driven, which means that your program waits until a certain event arrives (for example that a button was pressed) and reacts on that event. Events in SQLWindows are called Message Actions. The actions defined in SQLWindows are called Scalable Application Messages (SAMs), and you can also define your own custom message actions.

```
Pushbutton: pbQuit
  Message Actions
   On SAM_Click
    Call SalQuit ()
```

Figure 5.2: Example of Message Action

To simplify the work there is a huge function library to perform operations like showing dialogs, converting between data types and get data from controls.

SQLWindows is a mix of object based and procedural programming. While there are support for classes and inheritance, there are also functions that manipulates object “from outside”. An example is the function SalScrollGetPos, which retrieves the position of a scrollbar cursor. Most of the time you inherit from a control or window to create new classes. Classes that have no base class and no graphical view are called Functional Classes.

5.2.1 ActiveX Support

All COM components registered are listed in the Control Palette and can be inserted by using drag and drop. This makes it quick and easy to insert new controls to the application. The Attributes Inspector supports ActiveX, so that you can use it to view and modify a control. All events supported by the ActiveX control are listed in the outline as Message Actions and you can therefore easily add the logic needed to respond to events emitted by the control. All forms and dialogs in SQLWindows can act like control containers, so an ActiveX control can be included in any type of window.

The Object class is a functional class that represents a COM component in the project. It's actually a thin layer over the dispatch interface of an automation server (see section 4.2 for an explanation of automation servers).

Create	Creates an instance of the component
Invoke	Invokes a method
PushDate/Number/String, etc.	Push a variable on the stack
PopDate/Number/String, etc.	Pop a variable from the stack
FlushArgs	Reset the stack

Table 5.1: Important methods of functional class Object

There are four steps needed when you invoke a method:

1. Push the parameters onto the stack
2. Execute the method by calling invoke.
3. Pop the parameters off the stack.
4. Flush the parameters from the stack.

```
Call PushString(”datasource”) ! Push the argument on the stack
Set bRet = Invoke(”Connect”, INVOKE_FUNCTION)
If (bRet)
  Call PopBoolean(-1, bRet) ! Pop the return value
  If (bRet)
    ! Connected!
    Call FlushArgs()
```

Figure 5.3: How to call a COM method

As you can see it's quite a lot of job just to call one single method. To make the job easier there is a special ActiveX wizard. The wizard lists all COM components found in the registry and gives you the opportunity of making a functional class representing that component. This class derives from the Object class (see above) and abstracts the Invoke calls, enabling you to use ordinary function calls when you use the component methods. To speed up the generation of the functional classes, SQLWindows adds the information to the include library. This way the application can just include the library each time you re-use a component.

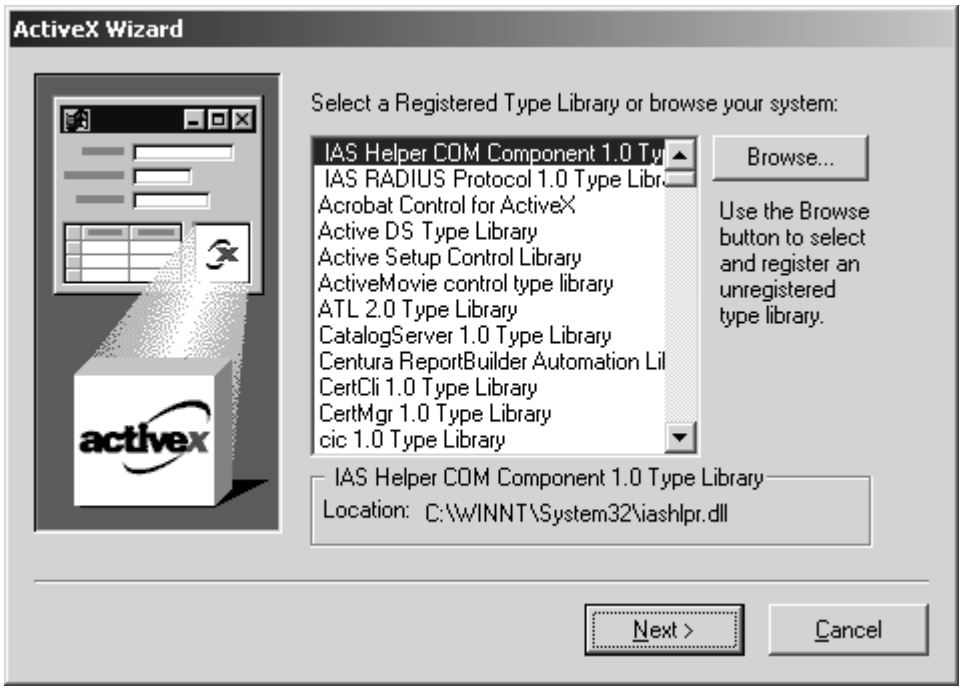

Figure 5.4: SQLWindows' ActiveX wizard

6 Description of Tests

COM is a huge area to test, and to make the test as exhaustive as possible it is separated in three different parts. These parts should represent different uses of COM. Of course, there are millions of ways to setup the test, but this division covers most normal ways to take advantage of the COM technology. The three tests can be summed up as follows:

- How SQLWindows supports a simple in process COM component.
- How SQLWindows supports a component that lives in another address space than the SQLWindows application itself.
- What support SQLWindows has for a complete ActiveX control, acting like one of SQLWindows native controls.

The first test is needed to secure that SQLWindows can be used at all with the whole tree of ActiveX technology. If it has this basic support then it is theoretically possible to use any technology that builds on COM.

The same line of thinking can be made for the test of Distributed COM. The possibility of creating network applications with COM really stands and falls with the support for DCOM.

The last, and perhaps most important, test checks how SQLWindows can be used with the ever-growing set of finished and hopefully well tested ActiveX controls. Support for this part of COM technology can increase development speed vastly. Instead of having everyone in the whole world making their own implementations of Rich Text editors and calculators, you can just include the needed control to your application.

The simplest possible type of COM component is a collection of methods and data to perform some simple set of operations. An example of when such a component can be useful is in a three-tier application, where presentation, logic and data are separated.

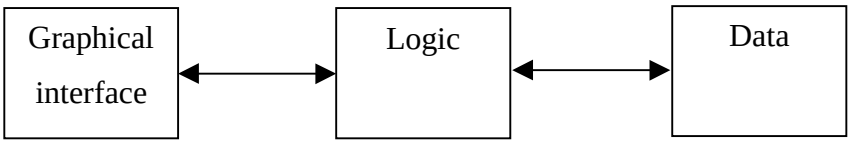

Figure 6.1: A thee-tier application

Here the logic tier can be implemented using COM and the graphical user interface, written in SQLWindows makes calls on that component to process the data. A local COM component which lives in the process space of the GUI has its limitations, but can often be very useful.

One big problem with local components is that they work on the client's machine, and uses the client machines resources to perform its tasks. This is not very suitable if the component performs more complex operations. It may also clog the network if lots of information needs to be exchanged between the data and the logic tier. It's interesting to inspect what kind of DCOM components are supported by SQLWindows, how they must be designed and how they perform.

The first two tests are components without a graphical user interface of their own. The user of the components must add controls that trigger method calls and property retrieval on the component. A complete ActiveX control in turn has its own user interface, and many times the only thing the SQLWindows programmer needs to do is to include the control to the project. This makes for short development time and makes it easy to move the control to some other development tool. Examples of uses are some specialized control to view or edit a table of a database, and to display documents and reports.

6.1 Test of COM

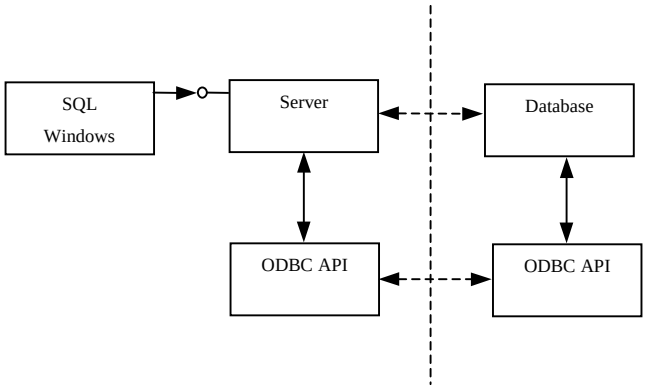

Figure 6.2: Overview of Test 1

6.1.1 Definition

The first test consists of three components. First a GUI is created in SQLWindows. This user interface needs access to a database to perform its actions. It does not access the database directly though, instead it uses a COM component.

The COM component implements one user interface, IDatabase, which can be used to retrieve data from one table. An important part of this test is to make the COM component as simple as possible. This is to enable to really check which interfaces must be implemented.

Connect(BSTR sSource)	Connects to the specified data source
Disconnect	Disconnects from the data source
FetchNext	Fetches next post of the table
FetchPrior	Fetches previous post of the table
FetchLast	Fetches the last post of the table
FetchFirst	Fetches the first post of the table

Table 6.1: Methods of custom interface IDatabase

ID	Long, ID of current post
NAME	String, Name field in current post
CONNECTED	True if connected to data source, else false

Table 6.2: Properties of custom interface IDatabase

The database is very simple, each row of the table contains a name and a unique ID.

ID	Name
1	Adam Anderson
2	Beate Bowman
3	Cinderella Carlisle

Table 6.3: Contents of the Access database

To access the database, the COM component uses ODBC, which is briefly described in section 4.6.

6.1.2 Development Steps

1. Create a dummy user interface in SQLWindows.
2. Create a dummy, local, in process COM component.
3. Connect the user interface with the COM component.
4. Create the database.
5. Connect the COM component with the database.

6.1.3 Installation

It's simple and straightforward to construct the interface. Being a fourth generation language, you just need to point and click to create the GUI. The first thing to do is to add a form (top-

level window) to the project, and then drag and drop the data fields and buttons needed into the form.

The construction of COM component is of course much more complicated. For the first test there is a need of complete control over which interfaces are implemented, and Visual C++ was chosen as a development tool. The component was made from scratch, without any help from a wizard. The class must be registered in the Windows registry. To simplify that part, the helper class CRegObject from MFC (Microsoft Foundation Classes) was used. The next step is to create an IDL-file and use the IDL-compiler MIDL to generate a code skeleton and type library. Then the component and its associated class factory can be implemented.

As a first step the component only supported IUnknown and the database-enabling interface IDatabase.

During the installation phase it was clear that SQLWindows could not handle COM components that are not automation servers (see section 4.2 for information of automation servers). Since SQLWindows has no concept of pointers it cannot reach the methods of IDatabase directly. Therefore a dispatch interface had to be added to the COM component. While it’s possible to compile a SQLWindows GUI using pure automation servers, the end product is very unstable. The component was tested in Visual Basic (which does not use the IDispatch interface) and VBScript (which accesses all methods and properties through IDispatch), but in SQLWindows it was incredibly unstable.

The component triggered a page fault in one of Centura's software modules (CDLLI15.DDL) the second time the application accessed a method on the component. and each time the application was closed it crashed. There was no way to be found around this problem, other than to make a complete, but invisible, ActiveX control of the COM component. Since ActiveX controls are so complicated, you don’t make one from scratch. The only solution was to completely re-make the component as an invisible control using some more advanced tool. For this test the Visual C++ MFC ActiveX Control Wizard was used. This wizard generates most of the skeleton code needed for a simple control.

The database can be created in any type of tool as long as the driver supports ODBC.

Lastly the data source must be connected with the database. For this test the Visual C++ ODBC API was used, which has a fairly complete support for ODBC.

6.1.4 Test Platform

Component development and testing	Visual C++ version 6.0 Personal Edition
Application development and testing	Centura SQLWindows/32 version 1.5.1
Test system	Microsoft Windows 98 version 4.10.1998 Intel Pentium II processor 192 MB RAM

Table 6.4: Test Platform of Test1

6.2 Test of DCOM

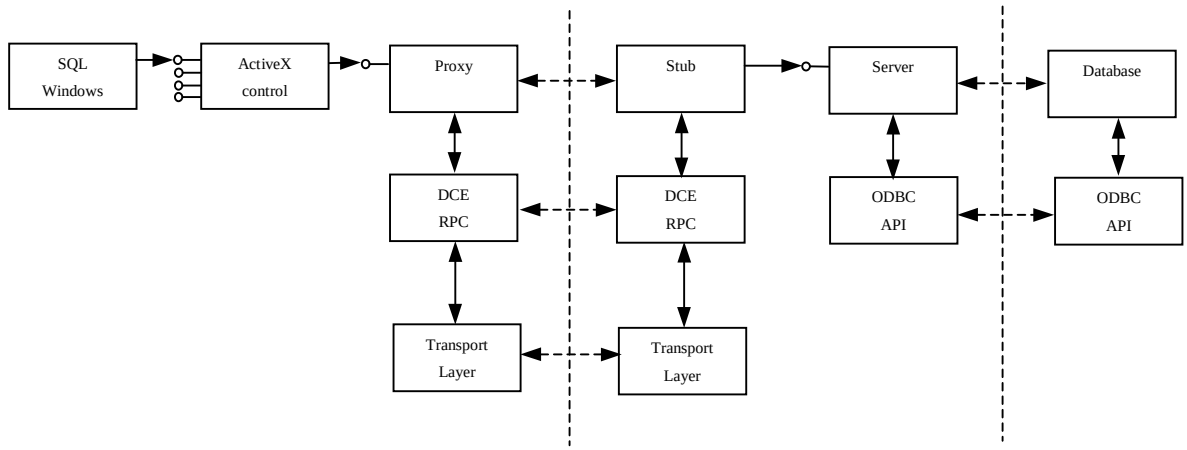

Figure 6.3: Overview of test 2

6.2.1 Definition

The next step in the test was to move the COM component out of the process space of the GUI. The component is reconstructed to become an out-of-process DCOM component. This makes for a real three-tier application where the three parts can be distributed anywhere in the world.

You could imagine that you put the database in USA, the component handling the logic in Sweden, and run the GUI from Japan, and it would still work! It adds new demands on the component though, it must provide stubs and proxies for the COM system to use and preferably be multithreaded. The interfaces needed were unchanged.

One problem arises here. The experience from the first test showed that the component accessed from Centura must be a full-fledged ActiveX control. An ActiveX control is always a local, in process component, and therefore some design changes had to be made.

To give SQLWindows the illusion that it still works with a local component an invisible ActiveX control was added which sends the messages it gets to the DCOM component.

6.2.2 Development Steps

1. Create a dummy user interface in SQLWindows
2. Create a dummy DCOM component implementing IDatabase.
3. Create an ActiveX control that retrieves an interface pointer to the DCOM component's IDatabase interface when constructed.
4. Connect the user interface with the control.
5. Create the database.
6. Change the DCOM component's implementation so that it makes real ODBC calls when it's methods are invoked.

6.2.3 Installation

It's a bit more complicated to make a distributed COM component, mostly because you have to implement the proxies and stubs needed to make the communication location transparent. Microsoft's library ATL was used to ease the process. ATL stands for active template libraries, and is a collection of wrapper classes around the COM system functions, much like MFC abstracts the gory details of the Windows API. For more information of ATL, see The World of ATL [5].

No big changes were needed for the ActiveX control. To retrieve the interface of the remote component, it calls the COM system function CoCreateInstance, which takes a class ID and has an interface pointer as an out parameter.

```
CoCreateInstance(
                CLSID_Server, // Class ID
                NULL,         // Parent
                CLSCTX_ALL,   // Type of component
                IID_IServer,  // Interface expected
                (void **) &_pIEXEServer); // Return parameter
```

After this, the control can call methods on the interface just like with any other object:

```
_pIEXEServer->FetchNext(); // Fetches next record in the database
```

The ODBC calls in the DCOM component are identical to those in test 1 (section 6.1.3).

6.2.4 Test Platform

Component development and testing	Visual C++ version 6.0 Enterprise Edition
Application development and testing	Centura SQLWindows/32 version 1.5.1
Test system	Microsoft Windows 2000 version 5.00.2195 x86 Family 6 Model 3 Stepping 4 64 MB RAM

Table 6.5: Test Platform of Test 2

6.3 Test of ActiveX Control

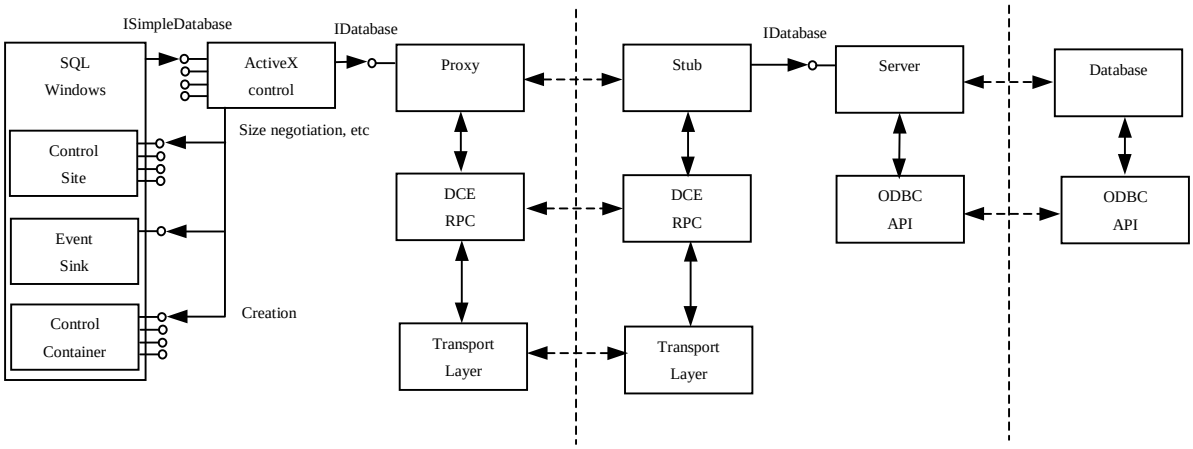

Figure 6.4: Overview of test 3

6.3.1 Definition

The last part tests how easy it is to use complete visible ActiveX controls in Centura Team Developer. Since these types of components sometimes do not even need any extra input from the surrounding application, it's important that it is very simple to include controls to the application. On the other hand, there is a massive increase in complexity, and the tool must be able to make using controls both simple and flexible.

The project uses the same DCOM component as in section 6.2 Test of DCOM, but this time the interface is used directly by the control and all SQLWindows needs to do is to include the control.

The custom interface of the control is a subset of IDatabase:

Connect(BSTR sSource)	Connects to the specified data source
Disconnect	Disconnects from the data source

Table 6.6: Methods of ISimpleDatabase

ID	Long, ID of current post
NAME	String, Name field in current post
CONNECTED	True if connected to data source, else false

Table 6.7: Properties of ISimpleDatabase

The control consists of two text fields showing the data at the current row in the table, and four buttons to move between rows.

Figure 6.5: ActiveX control interface

To give the container a chance to react on changes in the control, an event DataUpdated was added to the control.

6.3.2 Development Steps

1. Create an ActiveX control with a dummy user interface and ISimpleDatabase.
2. Create a SQLWindows project including the control.
3. Create a dummy DCOM component implementing IDatabase.
4. Connect the control with the DCOM component.

5. Change the interface of the ActiveX control so that it makes calls on the DCOM component.
6. Create the database.
7. Change the DCOM component's implementation so that it makes real ODBC calls when its methods are invoked.

6.3.3 Installation

The ActiveX control was made using the Visual C++ MFC ActiveX Wizard. The control acts as a simple container of MFC controls. The controls get initialized when the ActiveX control receives a CREATE signal from the operating system, and a message map is used to connect "button clicked" messages with function calls. MFC controls and message maps are thoroughly explained in Programming Windows with MFC, by Jeff Prosise [1].

When a button is clicked, one of the Fetch* methods on the DCOM component (see Table 6.1: Methods of custom interface IDatabase) is called, and the text fields are updated.

No changes were needed in the DCOM component. It has exactly the same implementation as in test 2 (section 7.2).

6.3.4 Test Platform

Component development and testing	Visual C++ version 6.0 Enterprise Edition
Application development and testing	Centura SQLWindows/32 version 1.5.1
Test system	Microsoft Windows 2000 version 5.00.2195 x86 Family 6 Model 3 Stepping 4 64 MB RAM

Table 6.8: Test Platform of Test 3

7 Results and Conclusions

7.1 Test of COM

7.1.1 Development Complexity vs. Component Reuse

It's quite easy to add an in process component to a SQLWindows project. To integrate a component you need to execute the ActiveX wizard and select the library needed. If it's not registered you can choose browse and find the type library manually. The wizard will then create the necessary functional classes. After this you must drag the ActiveX control into a window. Finally you retrieve a reference to the dispatch interface by calling SalActiveXGetObject.

Splitting components into many smaller ones is not as easy, since much of the code in the SQLWindows graphical user interface must be rewritten.

7.1.2 Integration with Legacy Systems vs. Customization

Even though the component does not have any user interface, and should not be used as a control, it gets added to the Controls Palette (see section 5.2). This is a bit counter-intuitive, and it does not feel right to add the component this way. To get a dispatch interface to the component you call the function SalActiveXGetObject, which takes a window as an in parameter and an object of type Object as out parameter.

The component would have felt much more as a part of the development environment if you just could add it as a functional class and use it directly.

There is some information loss when the communication between the COM component and the SQLWindows application fails. All methods of the interfaces in COM returns an error code with information about what went wrong, but the Invoke method of the Object class representing a COM component in SQLWindows only returns True or False. So some special properties of the COM component become hidden because of this, and customized behavior on different return values of the component is not easy to implement.

7.1.3 Level of Granularity vs. Simplicity

It's difficult to have many components controlled by the same container control (in other words: the same window), since the only function supported to retrieve controls is

SalActiveXGetObject. This means that it may be better to create a control as a collection of all controls needed in an application and flip through the controls using the Object class' function Next. Hopefully this will become better in a future release of Centura Team Developer.

It is easier to develop user interfaces in SQLWindows that works with only one big component.

7.1.4 Ease of Overview vs. Complexity

You can view the contents of a component by selecting "classes\<name of type library>" in the classes view, or directly when you write the logic in the outline. You can also see all methods and properties supported by using the Coding Assistant.

Something missing is the ability to see help text, even though IDL has extensive support for giving the user information about how the component should be used. In Visual Basic for example auto completion of method names as well as popup windows with helpful information and links to help documents is supported. In this area there's a lot of improvements possible to make SQLWindows easier to use with COM.

To sum up, you get an acceptable overview of the different components, but more detailed information is difficult to get.

7.1.5 Performance

Early binding makes the function calls faster than late binding. Since SQLWindows only supports late binding of components, this slows down the performance when calling methods. The fact that you must create a complete ActiveX control, even if you just make a simple one-interface component, means that there is a huge overhead of both memory and performance. The situation becomes even worse if the user of the component does not have access to the source code. The only solution then is to create a wrapper ActiveX control masquerading as the COM component.

7.1.6 Maintenance and Version Handling

The ActiveX include libraries are a nice addition to the ActiveX support of Centura. It does not work very well when changes are made in the interface of a component though. Centura does not automatically detect that the type library has changed. Instead you have to find the path to "Directories\Locations\ActiveX Libraries" and remove the type library file. To force SQLWindows to create a new include library you must start a project and use the ActiveX Wizard to create a new one.

This means that there's a big risk that the application crashes if you forget to make all these steps when you, or someone else, changes the interface of a component.

If you want to split a big component into many smaller, you should consider adding them all to one collection and add it to the project. A lot of logic must be added to get a dispatch interface to the part needed.

7.2 Test of DCOM

7.2.1 Development Complexity vs. Component Reuse

It's difficult to add a DCOM component to a SQLWindows graphical user interface (GUI). This is because you have no other option than to make an ActiveX control in some other language that can act as a proxy between your GUI and the component.

Since it's the ActiveX control's responsibility to load and unload all components needed by SQLWindows, it really is the development environment of the control that sets the limits to how easy it is to change to other components. If the interfaces differ you can try to tweak the calls of the control to fit that of the components, or change the control's interface and make the changes directly in SQLWindows.

7.2.2 Integration with Legacy Systems vs. Customization

The result of this test is similar to that in test one (see section 7.1.2). The components must be treated as ActiveX controls and added to a window in the SQLWindows project. It would have been much better if you could treat the component just as a functional class (see the description of SQLWindows in section 5.2).

7.2.3 Level of Granularity vs. Simplicity

Each component needs its own ActiveX control as a proxy. It may be better to create a control implementing all interfaces of all components and let it be a gateway to the different types of COM components.

A problem with that approach is that each time one component changes its interface, the type library information of the control needs to be removed and updated on every computer developing the SQLWindows user interface. This becomes really hazardous when the component lies on a server for a big company. It could mean hundreds of changes just for one little change in a distributed component.

After you have made you proxy control its not that big difference between using it with a big monolithic component or many smaller.

7.2.4 Ease of Overview vs. Complexity

Just like in test one (see section 7.1.4), there are improvements to be made in this area. You get about the same level of detail as when developing a GUI to an in process server.

7.2.5 Performance

Out-of-process servers always have bad performance when methods are called. The method calls needs to be marshaled and sent to the other process via some type of inter-process communication. A server that may not event be on the same computer as the client is of course even more affected by this. Because of this, the speed penalty of the call to the ActiveX control, and translation to a call on the client proxy is negligible. A bigger problem is memory usage at the client. A simple DCOM component needs a complete ActiveX control to back it up.

7.2.6 Maintenance and Version Handling

Since the project effectively consists of two components the out-of-process server and the proxy control, it can sometimes be possible, and useful to add some conversion logic in the control, to adapt the call to the component. If someone decided that it would be better to change the Fetch* calls into one Fetch() call with the direction as a parameter, you could translate the calls in the ActiveX control.

```
void CDatabaseCtrl::FetchNext()
{
    int next;
    _pIEXEServer->NEXT_POST(&next); // Get property NEXT_POST
    _pIEXEServer->Fetch(next);
}
```

Figure 7.1: Translation of call in the ActiveX control

This way you get around the hassle of changing the interface of the ActiveX control. If there are any completely new additions to the DCOM component, this does not work.

7.3 Test of ActiveX Control

7.3.1 Development Complexity vs. Component Reuse

It's extremely simple to use a visible ActiveX control in a SQLWindows project. The only thing you need to do is to open the Controls Palette, which lists all controls found. SQLWindows has a special ActiveX Control Palette showing the ActiveX controls. You select the control and click in a form window.

Replacing a control or splitting it into many smaller is a bit more problematic though. It's not enough to delete the old ActiveX control and inserting new ones. You must also remove all references to the old include library and change the outline. See section 5.2 for explanation of the Include Library and the Outline.

7.3.2 Integration with Legacy Systems vs. Customization

The control feels very much like a part of the development environment, and often you just point and click a few times to insert new ActiveX controls. After this you can use it much like one of SQLWindows own controls. The event driven paradigm of SQLWindows is nicely supported. All ActiveX control events are shown as Centura Message Actions (section 5.2), and you can add code to react on events from the control directly in the Outline. ActiveX controls are very seemly integrated with SQLWindows.

Most of the properties of the control can be viewed and changed directly when the control is included, and other types of properties can be reached by simply right-clicking the ActiveX control and selecting "Control Properties". Here the ActiveX control gets a chance to show its properties to the user in a dialog window.

7.3.3 Level of Granularity vs. Simplicity

Since visible ActiveX controls takes care of most of the interaction with its environment without consulting its container, it does not at that much complexity to have many controls in the same container.

A big monolithic control, like a control representing a calendar, is easy to use. You just have to paste it into a window and compile.

7.3.4 Ease of Overview vs. Complexity

There are the same problems with getting a good overview over the control as with Test 1 (section 7.1.4). It's a little bit easier to change and view the properties of the control though,

as you can see all its properties in the Attributes Inspector and that the control is getting a chance to show its features when you right-click it and selects “Control Properties”.

7.3.5 Performance

There is no direct performance penalty when working with ActiveX controls in SQLWindows. Unlike in the test of COM and the test of DCOM there is no need for a proxy class masquerading as the real component. The events of the control are sent through the dispatch interface of the control containers’ event sink (see automation, section 4.2). There has not been any possibility to examine if the conversion from events to Centura Message Actions slows down the performance.

Apart from that SQLWindows calls methods through the dispatch interface of the control, the performance is good.

7.3.6 Maintenance and Version Handling

Since ActiveX controls are more like stand-alone controls, where there is little communication between the control and its container, it’s easy to add new functionality to the control. You don’t even need to re-compile the SQLWindows application to take advantage of the updated control.

If changes are made to the custom interfaces between the SQLWindows application and the control, you still have to take all the actions described in section 7.1.6.

7.4 Conclusions

	1	2	3	4	5	6
COM	Average	Poor	Poor	Average	Poor	Poor
DCOM	Average	Poor	Average	Average	Average	Poor
ActiveX Control	Average	Good	Good	Average	Good	Average

Table 7.1 Test Analysis Table

1. Development complexity vs. Component reuse
2. Integration with Legacy Systems vs. Customization
3. Level of Granularity vs. Simplicity
4. Ease of Overview vs. Complexity
5. Performance
6. Maintenance and Version Handling

As the table above shows, it's easy to see that Centura Team Developers support for pure COM and DCOM is very poor, while ActiveX controls have a much better support. This comes naturally from the fact that COM/DCOM really is not supported without some special solutions and "hacks".

The technology that is most problematic to use with SQLWindows is COM. It is not integrated with the rest of the environment, and maintenance and performance is bad. The only acceptable aspects are that it's fairly easy to get an overview of the functionality of the component within the tool. It's also not too difficult to add to the SQLWindows application once you are sure it behaves, from SQLWindows point of view, as an ActiveX control.

DCOM shares many of its properties with COM, and therefore the test results of using DCOM almost the same as with using COM. When you have made a proxy control masquerading as the DCOM component, this proxy can be expanded to act as a proxy of many, both local and remote components. This makes it easier to add more DCOM components once you have added the first. Since DCOM and out-of-process components have bad performance anyway, the added performance loss from using the components in SQLWindows is negligible.

The test with ActiveX controls gave far more positive results. It's apparent that Centura had ActiveX controls in mind when they designed the ActiveX support for version 1.5.1 of Centura Team Developer. The controls feel completely integrated with its environment and it's at the same time easy to get and set custom properties. It's almost as easy to use many small controls with the graphical user interface as one big, and there is no big performance losses compared with other tools, like Visual Basic. There are some areas in which improvements can be made. Maintenance of controls is problematic since you need to make changes in the SQLWindows application every time an interface of the controls is updated. Splitting a big control into many smaller is not that easy. There isn't much support for getting different views of the controls.

References

[1] Jeff Prosise. Programming Windows with MFC, Second Edition. *Microsoft Press*, 1-57231-695-0, 1999.

[2] Julian Templeman, Ivor Horton, George Reilly, Alex Stockton. The Idea of COM. *Wrox Press*, http://www.comdeveloper.com/articles/COMIdea.asp.

[3] Kingsley Idehen. Open Database Connectivity Without Compromise. *OpenLink Software*, http://www.openlinksw.com/info/docs/odbcwhp/tableof.htm.

[4] The Object Management Group, http://ww.omg.org.

[5] The World of ATL. *Wrox Press*, http://www.worldofatl.com.