Creating Commercial Components

(Microsoft® COM Components)

●

Technical White Paper

View Contents

Date: February 29, 2000
Revised: July 9, 2001

Authors: Andrew Pharoah, ComponentSource
Chris Brooke, ComponentSource

www.componentsource.com

Email: publishers@componentsource.com

US Headquarters

ComponentSource
3391 Town Point Drive,
Suite 350,
Kennesaw, GA 30144-7079
USA

Tel: (770) 250 6100
Fax: (770) 250 6199
International: +1 (770) 250 6100

European Headquarters

ComponentSource
30 Greyfriars Road,
Reading,
Berkshire RG1 1PE
United Kingdom

Tel: 0118 958 1111
Fax: 0118 958 1111
International: +44 118 958 1111

Contents

Prototype Interface

Introduction

This white paper has been constructed to help component authors develop and enhance professional software components for delivery on the 'open market'. Information covered in the document is based on our knowledge and expertise of those component authors who successfully have established themselves in the component marketplace. The content is aimed at developers who wish to create components based on the Microsoft Component Object Model (COM). In the following chapter we discuss the business benefits of using components and identify the functionality suitable for component development. Following this we detail the component architectures and the languages in which components can be written.

Commercial Overview

The market for Software Components is expected to grow to around $4.4 billion by 2002, $1.0 billion from products and $3.4 billion from related services. (Source: PricewaterhouseCoopers)

Software applications are now created as a collection of software components. For example: Microsoft ® Office 2000. Increasingly application developers are employing component-based software development techniques, which enable them to reduce their time to market and improve their software quality. Software authors who are experts in a specific horizontal or vertical market sector are now "componentizing" their applications to meet the increasing demand for sophisticated business components. As such this represents a huge opportunity for you to unlock hidden revenues from years of research and development.

Why is buying a software component a good idea?

Everybody, software developers included, admit that they do something, (write a program or subroutine), better second time around. This is the essence of a "component", built and continuously improved by an expert or organization and encapsulating business logic or technical functionality. By buying a component a developer can add functionality to their application without sacrificing quality. Indeed quality should improve, as the component will have gone through several development iterations and include feedback from 1,000's of users.

What type of components will people buy?

Initially software components were used to provide technical functionality, such as SMTP for email or enhanced user interfaces. Developers are now requesting sophisticated components that solve real business issues from component authors.

What is helping make this happen now?

The open Microsoft Component Object Model (COM), Transaction Server (MTS) and Message Queue (MSMQ) standards have allowed component authors to focus on creating components rather than spending time on building complex proprietary "application framework environments" that "lock-in" users. These standards from Microsoft provide "services for components" and give the user a scaleable and secure deployment environment for his/her component based application. Plus - components can now be easily built using standard development languages like Microsoft Visual Basic ®, Visual C++ or Delphi.

To find out how to create components from your existing applications or as part of your next software development project - read the remainder of this white paper. Many varied technical topics are covered and this paper gives a "best practice" guide to commercial component creation in a Microsoft COM based environment.

Component Overview

Identifying A Component Candidate

How do I identify a component candidate? - Understanding how a component works and how functionality differs from applications is important when identifying a suitable component candidate. In this section we investigate existing applications for potential functionality, consider component reusability and finally discuss the importance of business knowledge and how this applies to the components you write.

a) Analyze Application Functionality

Developers should look at the functions encapsulated in their own applications and others to assess the commercial viability of componentizing particular functions. One of the main characteristics of a component is that the business logic is separate from the data. However, this does not apply to single parameter data that is passed to a methods interface. If you are creating a component it's important to manipulate data in arbitrary collections or streams. Typical examples include Visual Basic property bags and XML documents. (Read the 'White Paper' by Jim Parsons on separating data from programming and business logic) A typical example would be Microsoft's ADO component. This allows application developers to specify any ODBC compliant data source such as SQL server or Oracle, connect to the data source and then manipulate the data through an arbitrary recordset. The component was never programmed to know what data source to connect to or the type of data inside the recordset. This characteristic ensures the component is viable for any developer wishing to handle data located in an ODBC compliant database.

b) Component Reusability

An important factor worth considering is a products commercial viability. Market demand determines whether a component is commercially viable or should be used only within your own organization. Typical examples include components that are directly linked to hardware such as monitoring components for alarm systems. Unless the components can be sold separately from the hardware the ability to sell the product online is greatly reduced.

Components that can be integrated without any consultation will succeed in what's known as the 'Open Market'. This market allows components to be distributed without any consultation or tailoring service. All information regarding the product is supplied in online documentation such as demonstrations, evaluations, help files and sample code.

For more information on the open market browse to:
http://www.componentsource.com/services/cbdiopen_market.asp

c) Expert Functionality

Expertise and knowledge are the two areas you should focus on when writing a software component. If you are developing a component from scratch then consider the components already on the market and assess whether you could offer a different or superior solution. Where possible write components that are related to your core business area. It's likely that these functions will be more valuable than peripheral functionality designed to provide a basic solution. For example, if your core business provides insurance underwriting services then concentrate on these core functions first as opposed to peripheral components such as a basic user interface components for data presentation in a grid or as a chart of graph.

Component Architectures

Where are components installed? - Components, unlike applications are deployed in either a client or sever environment. Where they are deployed can depend on the functionality the component is intended to provide. Historically, GUI or visual components were designed to run on the client and were of little use in an environment where servers run without screens. However, the forthcoming .NET Framework and its related services have introduced a new paradigm: Server-side User Interface components. We discuss these types of components more in depth in the following section. For the purposes of this discussion, though, the important thing

to remember is that components without a visual interface can run on either the client OR server machines, although this may be dependent on the usage of the component and other aspects defined below.

a) Client-Side Components

Client-side components can be implemented in a variety of ways depending on the functionality required. Their overall characteristic is that all logic is encapsulated and run on the client as opposed to a server that may serve many clients. Another factor unique to client-side components is licensing. Depending on complexity, client-side components may be restricted with user run-time licenses. Due to the nature of a client component it is possible that unique licenses are required per client machine. Client-side components can be implemented in the form of Presentation, Technical and Business components. Examples of each are detailed in the topic 'Component Types'.

b) Server-Side Components

Server-side components are relatively new to the component market. Benefits enable the developer to provide solutions that run on a per server basis. These components serve many clients simultaneously without significant performance loss. Server-side components can also be upgraded efficiently removing the complexities of updating potentially thousands of desktop machines. Component logic is often run on powerful servers as opposed to a desktop machine. This makes the server-side component an excellent candidate for systems that require efficient throughput and performance.

c) Server-Side User Interface Components

Once available, the .NET Framework will allow developers to build server-side user interface components. These components can be designed to run on either a web server - via Web Forms, or on an application server - via Win Forms. In both cases the component itself resides on the server, enabling it to take advantage of increased resources, scalability, and fault tolerance. Whether you choose to develop Win32 distributed applications or Active Server Pages+ web applications, or both, the .NET Framework will allow you to integrate a rich user interface via server-side components.

d) Exceptions

Where possible you should design components in either a client or server architecture. However, there are a few components that are exceptions to the two definitions above. Typical examples include components that have a user interface that run in an client environment and are tightly coupled to components that run in a server environment e.g. stock/trading systems. These architectures exist for security reasons only i.e. the server component will only communicate with a specific client component and the client component will only communicate to a specific server component.

Component Types

What types of component are there? - Two main types of component exist - visual and non-visual components. Included in the visual components category are server-side UI components and client-side components. Both visual and non-visual components can encapsulate either technical or business knowledge. The differences between the two are dependent on functionality. For example if the component provides only a benefit to the developer e.g. a TCP/IP communication library then the component is categorized as technical. Business components provide a benefit to the developer and end-user by encapsulating business knowledge. Typical examples include address formatting and credit card validation components. Both visual and non-visual components have their benefits and in the following topics we will look at different examples of both in a client and server based environment.

a) Visual Components

COM Visual Components provide a pre-designed presentation interface to the user. Examples include data grids and charting components. These components are traditionally known as ActiveX Controls or OCXs. Visual components appear graphically in a component toolbox in

languages such as Visual Basic and Delphi. These allow the developer to select and physically draw the component onto a form and then manipulate properties via a design interface known as a property page. Another characteristic is that component functionality is run on the desktop machine as opposed to a powerful server. Because of this visual components are relatively lightweight on processing power. Visual components also provide developer licensing built in. This prevents users from copying the component into a development environment and using it at design-time. Developer licenses come as a file that must be on the users system in order for the component to work correctly.

Client-Side Example - Janus GridEX is an example of a 'visual' component that can run on either client or server based environments. The data grid provides the ability for binding, sorting and grouping data visually in the format of Microsoft's Outlook grid. The component has been configured for use in stand-alone applications and Internet browsers. Unlike client-side components that are pre-installed on desktop machines, server-side 'visual' components reside on Web servers and are either automatically downloaded to client browsers on demand or they automatically download the dynamically created content.

Server- Side Example - Chart - FX Internet Edition 2000 by Software FX is an example of a server-side component that dynamically generates content sending it to a client. Dynamic charts and image maps are created on the server and the sent to the client as either image files or as a bitstream directly to the browser client. Server-side components include additional features such as 'Safe for Scripting' and 'Safe for Initialization' properties. These properties protect component functionality from malicious use within a Web page. Without these features the component will not be browser (server-side) compliant. This subject is covered in greater detail in the chapter 'Deploying Commercial COM Components'.

b) Non-Visual Components

Non-visual components do not provide a pre-designed presentation interface to the user. These components are traditionally known as ActiveX DLLs and only contain functionality exposed to the developer through the programming interface. Unlike visual components this interface is not visible through a property page but is available on the standard COM interface. (The COM interface is explained later in 'Designing COM components') Non-visual components are adaptable and can be run in either client or server environments. This allows the functionality to be plugged into any n-tier architecture providing the application developer with a universal solution. Non-visual components do not appear graphically in a component toolbox and do not require a license file for development. However, this security implication is protected by free software provided by ComponentSource. Non-visual components designed to run in a server environment allow many clients to access functionality simultaneously without loss in performance. Typical examples include online housekeeping functions that require the dedicated processing power of a server.

Client and Server Based Example - Address 2000 is an example of a 'non-visual' component that can run in either a client or server environment. The component corrects address layouts to any national postal format in over 200 countries - particularly useful for mailing applications. The component is configured with two main features that allow its functionality to run in a server-based environment as well as a desktop machine. Firstly the component is apartment threaded, which is 'virtual' multi-threading. This enables multiple clients to access component functionality on more than one process thread. Without this, each client would have to wait until each process terminated by other clients. This subject is covered in greater detail in the chapter 'Designing Commercial COM Components'. The component's second feature is Microsoft Transaction Server (MTS) compliance. A scaleable 'Enterprise' solution should include MTS transaction calls within each function. MTS incorporates more efficiencies and transaction handling within a server-side component. This feature is particularly important for components that run in mission critical environments - e.g. banking and insurance systems. Non-visual server components that include these features will run on a client machine without MTS installed. The only difference is that all transaction handling is ignored.

Component Languages

What language do I use? - Practically any. Many development environments support the COM architecture. In addition to this, those environments that do not support COM tend to be bundled with wizards that wrap the compiled code inside a COM interface. In the following topics we look at the mainstream development tools currently used to build COM components.

a) Visual Basic

Visual Basic is the most widely used language for creating software components. This environment provides all the functionality required for developing and compiling an ActiveX DLL, ActiveX EXE or ActiveX Control (OCX) Visual Basic is a language that is easy to use and excellent for rapid development. Because of this developers can use the language for prototyping even if the component is finally written in another language.One of the considerations about using Visual Basic is the need to deploy VB runtime files. These have to be installed on the users system for the component to function properly. A clear understanding of what and when to install these files is covered later on in the topic 'Installation and Registration'. Visual Basic does come with the option to compress these files into to Cabinet files (.CAB) - similar to mini installation routines. More information on cabinet file is covered in the chapter 'Deploying COM Components'.

b) Visual C++

Visual C++ offers greater speed of execution over Visual Basic. Visual C++ should be used to implement components that are executed on a constant basis. This advantage is apparent only when Visual Basic components contain large amounts of functionality. However, this performance issue may be removed depending on the type of component you write. For instance if your component provides data access then it's likely many of the functions would be delegated to a data handling component such as Microsoft's ActiveX Data Object (ADO).The fact that C++ integrates tightly with the main operating system is the main reason for this improved efficiency. Compiling a DLL in C++ can limit the users ability to interface with the component. If you want to create a component with a COM interface then this can be accomplished by using the Active Template Library (ATL).You can use ATL in conjunction with the interface definition language (IDL) to define each function's public interface. This is interpreted by the wizards in C++ to compile a COM wrapped DLL. ATL components can be compiled as 'Minimum Size' or 'Minimum Dependency'. 'Minimum Size' requires only one ATL library whereas a component compiled as 'Minimum Dependency', although larger in size, is stand-alone and has no dependencies.

c) Delphi

Borland Delphi provides the rapid prototyping and development capabilities of Visual Basic along with the execution speeds of C++, using Object Pascal as the underlying programming language. Delphi provides a robust COM framework and wizards for creating a variety of COM components. In particular, the wizard for creating ActiveX controls enables you to automatically encapsulate one of Delphi native Visual Component Library (VCL) controls with an ActiveX control wrapper. While it's possible to write ActiveX controls from scratch, a la Visual C++, it's not recommended except for the most expert of COM developers. Borland choose this route of making ActiveX controls easy to create from VCL control but difficult to create from scratch in order to preserve their customer base's investment in VCL controls and also on the assumption that VCL controls are much easier to write than their ActiveX counterparts. You will generally find that COM components that use VCL are much larger in file size than those developed in Visual C++. This is primarily due to the fact that Delphi links rather large portions of VCL into the resulting executable file. If you wish for a smaller executable file, Delphi supports the Visual Basic style of deployment, which involves compiling with runtime package support and supplying the Borland runtime package(s) to your component end users. For interface design, Delphi provides a unique Type Library Editor that allows you to visually design your COM objects without having to manually write IDL code.

d) C++Builder

Delphi and Borland C++Builder are nearly identical tools, except for the fact that C++Builder uses C++ as its underlying language as opposed to Delphi's Object Pascal. While COM components are created in C++Builder using the same wizards and tools as found in Delphi, the framework

used by C++Builder to implement COM components is very different. Rather than rely on Delphi's VCL-based framework for COM components, which does not leverage the power of the C++ language, C++Builder employs a framework build around Microsoft's ATL. This framework enables the C++Builder COM wizard and IDE support. For ActiveX control creation, C++Builder also steers you toward creating your ActiveX controls by basing them on VCL controls, and the underlying C++ framework provides the connective tissue between the worlds of COM and VCL to produce the ActiveX control. Like Delphi, C++Builder also provides the option to compile with or without package for executable size efficiency. C++Builder also provides options for statically or dynamically linking with the C runtime library (CRTL) for further executable size efficiency.

e) Other Languages

Nowadays components can be written in any language. These often include facilities to incorporate the COM architecture into the component. Non-Microsoft languages supporting COM include traditional languages such as Merant MicroFocus COBOL.

Designing Commercial Components

Component Characteristics

Do components technically differ from applications? - There are various characteristics that differentiate components from applications. The following topics explore the component interface, the Windows registry, component error handling, threading models and the safety aspects of Web-based components. Developing components is not dissimilar from developing applications. An understanding of the fundamental differences will help you convert functionality in stand-alone applications and build new components from scratch.

a) Interfaces

Microsoft's Component Object Model uses interfaces to expose the behavior of COM components to the outside world. You can think of an interface as a list of functions supported by the component. The functions contained in an interface can be methods, property get or put functions, or even events, as described below.

Methods - Methods are similar to functions found in traditional applications. They contain code that can be utilized by the calling application. Components encapsulate methods that are public or private. This allows the component author to provide developers with entry methods only, removing any confusion as to which methods can be used.

Properties - Properties are abstract concepts that enable the developer to work with data elements of an interface as if they were data members on an object. In reality, properties are generally represented in COM as two methods: one to get a property value and one to set, or put, the value (or perhaps just a get method in the case of a read only property). Therefore you could, for example, create a property called "Name" on the interface, which would result in the creation of get and put functions. However, from a developer's standpoint they will simply use the property by name in most cases and be unaware of the underlying implementation details. Once a component is instantiated its properties are persisted until the instance is terminated by the parent application. This allows the properties to be changed either by the parent application or internally by methods or events. In the case of ActiveX controls, property values are usually persisted between design-time and runtime sessions.

Events - Events can be used to provide status information on a methods progress. Event methods in COM are contained in a separate interface from other normal methods and property getters and putters. In most development tools, the handling of events on ActiveX controls is automatic. In Visual Basic the handling of events on other kinds of COM objects is possible if the component is initialized to include 'events' information. In most other tools, events are handled by creating a helper class which handles the event interface. Once an event is fired, the calling

application will receive the notification in the form of a method call, and it can handle the situation as appropriate.

COM components can consist of multiple interfaces, and interfaces can also be inherited from one another.

When a new interface is created, a 128-bit number called a Globally Unique Identifier (GUID) is created to correspond to the interface. GUIDS are often called Interface Identifiers (IIDs) when used with interfaces. Because every GUID is globally unique, every interface in COM has its own unique IID. By using IIDs only as the internal mechanism to identify interfaces, COM mitigates the potential problems of naming clashes and interface versioning. When a new version of an interface is released, it should have a new name and a new IID.

b) Classes

Classes in COM (also known as coclasses) are somewhat analogous to classes as you would know them in an object-oriented language. A class in COM is the abstract entity that identifies the type of the COM object. Think of the class as the object type and the COM object the implementation of that type. One or more interface is associated with each class, as interfaces are the only way to work with COM objects - classes on their own provide no means of access from the outside world.

c) Registration

Due of the granular nature of components many hundreds can exist on a system at any one time. Therefore to keep track of all these components, the Windows registry is used to store each component under a Class ID. This enables a development language to query which components are on the system and which can be used in development. The Class ID is a globally unique identifier (GUID) that ensures each component is unique within the system. Information on a component is stored in the Windows registry by 'registering' the component either programmatically or with RegSvr32.exe as shown below.

Example: RegSvr32 C:\MyProject\Bin\Example.dll

On registration a ProgID is created which is a readable name for the complex ClassID. This makes finding a component in the registry a lot easier. In addition to the ProgID other details are also included under each ClassID. These include component version, threading model and most importantly the physical location of the component on disk.

d) Error Handling

Handling errors in a component is not the same as handling application errors. Firstly, you need to consider that any error not handled in a component will be raised to the client that called the method. For that reason, you must ensure that the information the client receives is meaningful. A client interface should be totally unaware that a component may be running a process. Therefore any error that occurs should be handled by the client and interpreted in such a way that any error message displayed is generated by the client and is in context with the process that has failed. Below are the main techniques for handling errors in a software component.

Handling Errors Internally - Handling errors within a component is no different to handling errors in a standard application. If a method unexpectedly generates an error then unless an error handling routine is included, the calling application will crash as well as the component. To avoid this situation, intercept the error, assess its severity and take corrective action, either by resuming to a specific line of code or returning a recognizable error code to the calling function.

Passing Errors Back to the Client - To return an error back to the calling client you must raise an error. You can raise an error by invoking the raise (or equivalent) method in the error object of your chosen language. Raising an error will allow you to return a number and error description back to the client. Alternatively ensure that you either set a public error property or error parameter on the methods interface before exiting the method. This will allow the client to interrogate the error property or parameter and take appropriate action.

Raising Errors from Error Handlers - The majority of methods and properties you write will contain error handler routines. Where an error handler receives an unexpected error then returning a generic 'unexpected error' description will not help the client find a solution. A good practice is to return the methods name that failed and the parameters that were passed to it. This information can then be passed back to the component author for investigation.

Handling Errors from Another Component - If your component references a third party component then you must handle all errors (known or unknown) that the secondary component may generate. Developers using your component may have no knowledge of the dependencies your component references. Because of this, you must not raise these errors to your client application.

e) Threading Models

With the advent of more server-based components the need to compile a component with a suitable threading standard becomes increasingly important under a multi-user environment. The following list describes the threading models available for COM components.

Single - The entire COM server runs in a single thread. This makes programming easy because data does not need to be protected from synchronous access, but it can hamper performance, since every method call is serialized into the COM server. When you create a single-threaded component run in a multi-user environment (or single user environment where multiple threads will be accessing the component), the performance at the client end can be extremely slow. On a client the user must wait until the client (or thread) in front has terminated its component connection. In a multi-user environment single threaded components are created per user. Because the server is constantly creating multiple instances all carried in memory the performance of the server can eventually grind to a halt, as all the available memory resources are used.

Apartment, also known as *Single Threaded Apartment (STA)* **-** Each COM object executes within the context of its own thread, and multiple instances of the same type of COM object can execute within separate apartments. Because of this, any data that is shared between object instances (such as global variables) must be protected by thread synchronization objects when appropriate.

Free, also known as *Multithreaded Apartment (MTA)* - A client can call a method of an object on any thread within that apartment at any time. This means that the COM object must protect even its own instance data from simultaneous access by multiple threads.

Both - A hybrid of Apartment and Free that provides the calling efficiency of the Free threading model but the callback efficiency of Apartment. This is done by ensuring that callbacks from the server to the client are serialized on a single thread. If a component marked as both (or MTA) is created from a STA, it is created in a new apartment with a new thread. If created from an MTA, it joins the MTA with its own thread. Creating a component as 'Both' requires extra work on the part of the developer to code in his own synchronization.

Components created in Visual Basic can only be Single or Apartment (STA). In order to created components that are free threaded or 'both' threaded you must use a lower level language such as C++, Visual C++, Delphi, and C++Builder.

f) Safety

Safety is an important factor when writing components for Web development. Unlike components written for traditional environments, developing components extended for use in Web pages requires the setting of two safety properties - 'Safe for Scripting' and 'Safe for Initialization'.

Safe for Scripting - Marking components 'Safe for Scripting' prevents applications

or components accessing functions that could create, change or delete arbitrary files or change system settings.

Safe for Initialization - Marking components 'Safe for Initialization' prevents applications or components from providing initialization data that could create, change or delete arbitrary files or change system settings.

The following details the potential hazards of releasing a component that is not 'Safe for Scripting' or 'Safe for Initialization'.

- Exposure of private information on the local computer or network
- Modification or destruction of information on the local computer or network
- Faulting of the control and the potential crashing of the browser
- Consumption of excessive time or resources such as memory
- Execution of potentially damaging system calls, including execution of files
- Use of the control in a deceptive manner and causing unexpected results

Information on deploying a 'safe' downloadable component is covered in the chapter 'Deploying COM Components'.

Design Considerations

How do I develop a software component? - Before writing a component you should analyze the functionality and architecture first. In this section we discuss components functional boundaries, assess where a component will physically run and how to implement an extensible interface. Considering these elements will prevent the inclusion of unnecessary functions and provide a focused solution for developers.

a) Identify Component Scope

It is important when designing a component to identify the functionality that should be included and the functionality that is best incorporated into another component. A component should allow a developer to integrate a precise solution as opposed to one that provides features over and above a basic requirement. For example, designing a business component that provides addressing services could include various functions such as address deduplication, post coding and address formatting. In this example the three functions are mutually exclusive and should be implemented separately.

However, if the component was an address deduplication component that incorporated extended functionality e.g. off-line batch deduplication then this functionality should be included. It is possible to create one component that can be sold at three different levels. By using the ComponentSource licensing technology (C-LIC), it is possible to block extended functionality. This allows authors to publish one component but sell a separate standard, professional and enterprise edition.

Defining component scope will help ensure a component does not become monolithic and mimic an application without an interface. Unbundling functionality into separate components will prevent the component from becoming over complex and difficult to maintain. The advent of online purchasing and the removal of packaging and shipping costs has meant there no longer is a need to bundle disparate functionality into one component or to market several components in one suite. Removal of this traditional cost implication will allow authors to publish highly focussed discrete components and provide customers a wider choice.

b) Choose Architecture

Choosing architecture will depend on the functionality the component will provide. As discussed earlier in the chapter 'Component Overview' client components are often visual in some respect such as grids, charting and toolbar components. However, non-visual components may fall into this category if the functionality is 'lightweight' and does not severely impact the processor, typical examples include file encryption and communication components. If the component functionality can be used in a multi-user environment then consider developing a scaleable server based component. This should be compiled either apartment or multi-threaded and preferably MTS enabled for scalability and transaction handling.

Installing components in a server environment is less time consuming than having to install a component on several client machines. The improved performance and upgradeability benefit that server components offer is reflected in the price and provides component authors with an opportunity to generate revenues based on a server architecture. Server based components will provide the backbone to future Application Service Providers (ASP) and consequently developing server components now, will position you for the future growth in this market.

c) Prototype Interface

Prototyping a component interface can be a useful exercise and will help determine the complexity of integrating the component into an application. Component integration should be a relatively quick process. If the interface has hundreds of public properties, methods and events then it's probably too complex and will confuse users and generate support issues. A technique, which can help prevent this problem, is to write the help file before implementation. This will help you detail a functional specification and pinpoint any areas that could be consolidated or improved upon.

Documenting Commercial Components

Documentation Benefits

a) Reduction in Pre/Post Sales Support

Documentation for components sold in the open market is particular important as 'face to face' interaction does not take place between author and customer. Providing a comprehensive set of documentation will ensure that pre/post sales support is kept to a minimum. Providing pre sales documentation i.e. a thorough component specification prevents many of the refund situations common in traditional 'box product' channels.

Traditional channels sell product by providing marketing information but not the finer detail covered in help files and other technical documentation. Providing information such as help files and evaluations enables customers to make an 'informed' purchase decision. Documenting and publishing known issues such as Frequently Asked Questions (FAQ's) on a regular basis will also help reduce technical support after the sale.

b) The Confidence Factor

Components sold on the open market are 'Black Box' i.e. the source code is hidden. Because of this, trust is extremely important between customer and author. Therefore, provision of detailed product information such as evaluations, help files and white papers is essential for building confidence in potential customers.

Typical Documentation

What documentation should I provide? - The following section provides a detailed insight into the different types of documentation that should be provided when selling components in a commercial market. For examples of presenting online documentation in a concise and professional style browse our top selling products.

a) Online Documentation (HTML, HLP and PDF Files)

HTML is probably the best format of documentation you can provide and can be used for displaying information in text and graphical format. Typical examples include product overviews with screen shots and/or related diagrams. Customer can view HTML instantly as opposed to other document formats that must be downloaded first. A new format recently introduced for online help files (CHM) This provides the same search facility as traditional help files but in HTML. Writing a help file is relatively easy and can be achieved using help authoring tools. More information on these tools can be found on our Web site: Help Authoring Tools.

Portable Data Files (PDF) are documents that can be viewed on IBM compatible or MAC platforms. The PDF file enables the creation of technical documentation in a 'book' format. Therefore, converting a published manual into an electronic form is probably the most efficient

way to achieve this. The drawback with PDF files is the requirement of a proprietary viewer that must be downloaded first. To write a PDF file you will need to download the Adobe PDF Writer.

b) Demonstrations

Developing a product demonstration can prove a valuable asset in the documentation you provide customers. Exposing component functions will help users understand the benefits of the product as a component-based solution. Demonstrations are compiled applications assembled with the component. They are not like evaluations that allow developers to use the component in a development environment. More information on evaluations is covered in the following topic.

The objective of a demonstration is to educate users on the functionality incorporated inside the component. The interface should demonstrate the main functions in a format that is understandable for all customers. Because of this it's important to remove industry jargon and acronyms that may confuse users. For data bound components, providing the option of entering a DSN (Data Source Name) could be of benefit. This allows users to connect to internal data sources in their own organization and apply meaningful data in context with the component.

Demonstrations often reference dependencies and therefore testing the demonstration on a clean machine is extremely important. Clean systems contain freshly installed operating systems removing the potential hazards of previously loaded software. If your demonstration references any dependencies then you must create an installation kit. Sometimes it's beneficial to include the demonstrations within the evaluation kit and thus remove the need to write and maintain two separate kits.

Finally, the quality of a demonstration is directly correlated to the quality of the final retail product. Where possible, design your demonstration in-line with an accepted standard e.g. Microsoft standards. This helps build a perception of quality and trust with customers - remember demonstrations can make or break a sale.

For more information on Microsoft standards browse to our Resource Library.

c) Evaluations

Component authors recognize evaluations will help secure a product sale. Once a customer is happy with a specification they often trial the component to check the component will actually provide the functionality they are looking for. Customers do not doubt component based development, but may have concerns with an 'independent' solution, because of this component evaluations are essential. Unlike applications, component evaluations add value and play a significant role in the pre sales process.

Writing an evaluation will require consideration into security. Producing a component that displays a reminder screen or setting time limits hidden in cryptic keys within the registry are just some of the techniques currently used. Setting a 5-10 day trial period for technical components and 10-30 days for complex business components is recommended. This gives the customer enough time to evaluate the product and make a decision whether to buy.

An ideal evaluation is the full retail restricted by a security feature detailed above. This prevents users having to download the evaluation and retail component separately. ComponentSource has developed a license protection facility called C-LIC primarily designed to protect evaluations that can be unlocked into full retail products. C-LIC displays a reminder screen requesting the user to enter a license key provided when the full retail is purchased. More information on C-LIC is covered in the topic 'Deploying COM Components'.

d) Sample Code

Sample code is particularly useful when developers need to prototype and assess component functionality. A good technique is to provide the sample code used in the component demonstration. If possible, this should be provided in a basic, intermediate and advanced version. This will allow the developer to grasp how the demonstration was developed and it's stages of advancement throughout its development cycle.

The provision of sample code for environments such as Visual Basic, Visual C++, Delphi etc will

ensure more developers are aware of compatibility with their chosen environment and that your focused on providing the best solution possible. If you only show VB samples, then only VB developers will buy the component. In this scenario a Delphi programmer may believe support is not available for Delphi users. The more development environments you support with sample code will improve the product's perception and boost sales.

Sample code usually is the final step that customers evaluate before making a decision whether to buy. Therefore its important to maintain a good perception by commenting all code and explaining exactly what happens and why. The quality of sample code will directly correlate to the quality of your final product. Because of this professionally written sample code using correct naming conventions, coding structures and error handling is essential. If the sample code is well structured then it can be reused in actual projects. This makes the whole process of integration far less complex and useful for developer's who need to rapidly assemble a component-based solution.

For more information on Microsoft standards browse to our Resource Library.

e) Readme Files

In this topic we list the various information that a Readme file should contain. Most installation scripts provide users with an opportunity to view a Readme file for last minute changes or errata information once installation is complete. These files should be written in a universal file format i.e. a text (TXT) file or HTML file. This prevents users having to own proprietary applications such as Microsoft Word to view the file. The following list provides an insight into the various information supplied in component Readme files.

Products Changes - this section is extremely important and should note all the functional changes that have been made in comparison to previous versions and any changes to documentation, installation etc.

Bug Fixing - bugs resolved from previous versions should be fully documented. Include the component version that contained the bug and a description of what has changed. This is particularly important if the component's interface has been changed.

System Requirements - Although compatibility information is supplied in our own sales documentation its worth reiterating this information in your Readme file. This should include information such as operating system for deployment, safety levels, threading standards etc.

Service Pack Installation - You should define any services packs that were applied when compiling the component. This often is the reason for components failing to run in a user's development environment.

Definitions of Component Filenames - Listing the filenames of all components (including dependencies) is particularly useful if the user is attempting to identify a problem. Although help and dependency files include this information, Readme files are often browsed as well.

Detailed Installation Notes - This should include information on how to de-install and update previous versions. A troubleshooting section should also be included defining solutions to common installation problems.

Notes on Sample Projects - Document any assumptions, known issues etc. If possible, describe each of the projects and the functions they expose. In addition to this defining a project's complexity i.e. basic, intermediate or advanced can also be of help.

Distribution Information - Particularly useful when a user creates an installation kit. Your component may reference many other dependencies, therefore detailing this information will help the developer create a tailored installation kit and prevent

many of the 'missing dependency' issues when testing.

Known Issues - You must document all known issues. If possible, also explain why the problem arises. If you do not provide this information then it's likely that unnecessary technical support issues will arise. Documenting known issues will demonstrate that you care and are focussed on providing a future solution.

f) Pre-requisites

Pre-requisites provides the customer with details on required software, product size, required memory, service packs where appropriate and publicly available DLLs such as Microsoft's ActiveX Data Objects (ADO) It is worth including the minimum and recommended size when defining memory and hard disk allocation.

g) Dependencies

Dependency files such as Microsoft's .DEP file provide the file information that a component references at runtime. This may include details on which files to register, where each file should be installed and also a URL defining the download location for each .CAB file that the dependent file is reliant on. This URL is particularly useful when your customers create their own installation kit.

Microsoft's Package and Deployment wizard provides functionality to interrogate a component .DEP file, locate the relevant URL and download the correct .CAB file for inclusion within the installation kit. This prevents versioning problems that may arise from integrating independent components that may also reference other dependencies.

Installation tools such as Microsoft's Package and Deployment wizard allow the creation of dependency files. Below is an example of a typical .DEP file.

```
; Dependency file for setup wizards.:
[Version]
Version=6.0.81.69
; Dependencies for MSRdo20.dll

; Default Dependencies ----------------------------------------------

[MSRdo20.dll]
Dest=$(WinSysPath)
Register=$(DLLSelfRegister)
Version=6.0.81.69
Uses1=rdocurs.dll
Uses2=ComCat.dll
Uses3=odbc32.dll
Uses4=
CABFileName=MSRdo20.cab
CABDefaultURL=http://activex.microsoft.com/controls/vb6
CABINFFile=MSRdo20.inf

[ComCat.dll]
Dest=$(WinSysPathSysFile)
Register=$(DLLSelfRegister)
[rdocurs.dll]
Dest=$(WinSysPath)
CABFileName=MSRdo20.cab
CABDefaultURL=http://activex.microsoft.com/controls/vb6
CABINFFile=MSRdo20.inf
```

h) Compatibility

The following topic looks at the compatibility aspects of a software component. Publishing your product on www.componentsource.com will require a comprehensive specification of the component's compatibility. The product submission form that we ask you to complete covers the eight areas detailed below.

Operating System for Development - This section covers the different operating systems that your component can run on. The component may run on Windows 98 and Windows NT however this does not mean that it runs on Windows 2000. Unix is another example of an operating system that exists in many flavors such as Solaris, HPUX and Aix. Because of this, testing component functionality before

labeling an operating system 'compatible' is essential. Any future technical issues that arise due to operating system compatibility will require support.

Architecture of Product - The architecture of a component defines the type of system the component is compatible with. For instance, developers maintaining legacy systems may find it especially important to know a component is '16 bit' as opposed to '32 bit'. The fact a component can run in a '16 bit' environment such as Windows 3.1 may be the reason a customer decides to purchase. Another factor worth considering is consistency. If you label your component '32 bit' then do not check Microsoft Access 2.0 or Microsoft Visual Basic 3.0 as compatible as these are 16 bit products!

Tool Type - This section defines your software as an application tool, component, add-in etc. Again, selecting the tool type will be directly related to the containers the component can be used in.

Component Type - Defining the component type is particularly important when filtering our product catalog for co-branded sites. Your component will be receiving exposure in different filtered catalogs aimed at specific audiences - there is little point in marketing a VCL (a Delphi only component) to Microsoft Visual Basic audience. Consider each of the types listed - defining a component as a DLL does not define the component with a COM interface. In this particular case the component would not be published on a co-branded site such as Microsoft unless it was labeled as a COM Object/ActiveX DLL/In-Process Server.

Built Using - Stating the framework that a component is built with is especially useful when establishing if the component will run correctly in a design time environment. For example, a Microsoft Visual Basic 6.0 component may not work in a Visual Basic 5.0 environment due to the absence of MSVBVM60.dll. This does not mean the component is not compatible but provides the developer with information on the basic run-time libraries required for the component to run.

Compatible Containers - This section defines each development environment in which the component can be used. Mark only those environments that you have tested and can support your component in. Completing this section will make you eligible for different marketing initiatives and inclusion into catalogs targeted at specific audiences such as 'ActiveX' or 'Delphi' users.

General - This section includes options not easily categorized. Digital signatures, compatibility, scalability and support for apartment model threading are just some of the options that may require inclusion.

Year 2000 Compliance - If the component includes a date function or is anyway related to file storage then test for Year 2000 issues. Although components such as calendar controls are the obvious candidates the application tools can also be eligible for such tests. Tools that compile files such as help authoring tools and installation wizards must state Year 2000 compliance - either 'Yes', 'No' or 'Not Relevant'.

Deploying Commercial Components

In the following chapter we discuss component installation, followed by discussions on testing and component licensing.

Component Installation

How do I install a component? - Installing a component into a system requires more than just copying files into directories on disk. In fact most of the rules and techniques that apply when installing an application apply to the installation of a component. In the following section we

discuss the topics of writing installation scripts, protecting a component from illegal or malicious use and the implications of digitally signing components for use on the Internet.

a) Writing a Script

Creating an installation package is one of the final tasks to complete hen creating a component for commercial reuse. Packaging a software component is no different to any other software application. Nowadays most installations tools come packaged with wizards to help you throughout the process of creating a professional setup kit. There are a number of installation tools available for creating setup kits. More information on these tools can be found on our Web site: Installation Tools

Registering your component and dependent files is extremely important - incorrect registration is the most common cause for damaging a users system. Most installation tools are incorporated with the facility to select how the setup kit will replace a file. All component files contain a version number. This number has greater priority that the file date and therefore analyzing the date only at installation will not suffice. Some files are dependent on each other such as OLE. OLE requires three files (Ole32.dll, Oleaut32.dll and Olepro32.dll) to be installed each with the same file version. Therefore it's important to write an installation kit on a clean machine otherwise any mismatches on your system will be included in the setup kit and installed onto the end users system. By using a clean system will ensure that any dependencies missing will surface when testing the kit. Updating the installation script with missing files, re-compiling and re-testing will ensure the setup kit safely deploys files to users systems.

b) Protection

In this topic we discuss how to protect a component from illegal and malicious use. Protecting a component from illegal use applies to both visual and non-visual components. However, protecting a component from malicious use only applies to components intended for download into an Internet browser. Malicious use is where a component can be scripted to harm an end-users system, and because of this certain protection procedures should be applied.

Illegal Use - Nowadays, customers expect one download that runs in evaluation mode for a set number of days. Once this evaluation has expired, functionality is disabled until a license key is purchased and entered, unlocking the component into a full retail version. The best form of license protection is to use a reminder screen that appears each time the parent application calls the component. This prevents users without a license from releasing an application into a commercial environment.

- **Date Expiry -** How long would it take to evaluate your product? This should be short for non-complex GUI/Technical components 5 to 10 days and longer for complex Technical/Business components - 30 days max.

- **Reminder Screens -** Where the protection is a warning that 'pops up' every time an application is run that is built with the evaluation.

- **Limited Functionality -** This is not popular with customers, as they cannot fully evaluate the functionality.

Malicious use - Setting the safety levels of a component can be implemented one of two ways. The easiest and most common method is using Microsoft's package deployment wizard. The second option, although more complicated, is to incorporate the IObjectSafety interface within component code. This is particularly useful for protecting specific functions within the component. The following details both methods for incorporating safety features inside a component for use in Internet browsers.

- **Using the Package and Deployment Wizard -** The wizard can be launched as a stand-alone application or from the Visual Basic Add-in

Manager. The wizard takes you through the process of creating a cabinet (CAB) file. Once the project and dependent run-time files are defined you can set the safety levels for each component listed.

• **Using IObjectSafety -** This interface exposes functionality to Internet browser 'Safe for Initialization' and 'Safe for Scripting' security features. The advantage of the IObjectSafety interface is that you can protect specific functions within the component unlike the Package and Deployment wizard that protects all functions selected as Safe for Initialization and Safe for Scripting.

c) Component Verification

Digital signatures are used if you are developing a component that will be downloaded on the Internet. This allows the user to verify that a file they download is identical to the file released by the component author. This check is to ensure that the file is from a reputable source. Below is the procedure for digitally signing a component.

Sign Component - Any software that is available for download will require a digital signature. If the component is not signed then by default Internet Explorers will refuse to download the file. Digitally signing a component identifies who is legally responsible for any 'system destruction' at the point the file is downloaded or run. You can package your component and its dependent files into a cabinet (CAB) file. Although separate files can be signed, CAB files allow you to incorporate many files under one digital signature. Files that can be digitally signed are: -

• DLL (Dynamic Link Library)
• EXE (Executable file)
• CAB (Cabinet file)
• OCX (Ole Control eXtension file)

Verify Safety Level - Components used in Internet browsers can be run with a pre-defined script. This means the component could execute potentially hazardous data or run the component using a malicious script. This can range from utilizing a delete method to automatically installing a macro virus. The legal entity on the digital certificate will be held responsible for any malicious damage caused to the users system. Installation tools like Microsoft's application setup wizard require the author to brand a file as "Safe for Scripting" and/or "Safe for Initialization".

Arrange Licensing - Applications compiled with development tools such as Microsoft Visual Basic automatically incorporate component run-time licenses when an application is compiled. However, this does not apply when components are downloaded into an HTML page. If you wish to license a component for use in HTML then the creation of a License Package (LPK) file is required using the Microsoft LPKTool.exe.

Package Component - The next step is to package your component and dependent files into a cabinet (CAB) file. CAB files can be created using many installation tools such as Microsoft's application setup wizard. Typically, CAB files include:

Component file (DLL, OCX or EXE)
License file (optional)
Dependency file (files required to run the control)
Dependent files
HTML page (requires LPK file)

Test Download - Test the component download on a clean version of Windows 95, Windows 98, Windows NT Workstation, Windows NT Server and various editions of Windows 2000. This will allow detection of any operating system specific problems. Ensure the component is not registered on the test system by running

Regsvr32.exe using the following parameter.

Example: RegSvr32 /u C:\MyProject\Bin\Example.dll

Sign Component - To digitally sign a component you must firstly purchase a certificate from a trusted organization called a Certificate Authority (CA) The certificate is issued once the authority has validated your identity and is digitally signed with a their own private key. The Internet browser using the authorities public key can then decrypt the certificate thus preventing third parties tampering with the certificate. The certificate is then applied to the component using a utility supplied in Microsoft's ActiveX SDK.

Component Testing

How do I test a component? - Thorough testing is paramount to the success of a component being excepted in the open market. All evaluations and sample code should be tested in addition to the full retail product for functionality, installation and de-installation. An issue that should be approached with care is the dependencies referenced by your component. Most installation tools require the selection of the original component's project file. This allows the wizard to analyze all references selected at the time the component was compiled. Absence of dependent files referenced by other dependent files is probably the most common installation issue. This is why testing on a clean machine, on all operating systems and all development environments is imperative. If this rigorous testing process is not followed then the likelihood of damaging a customers system is high. Therefore, to create a clean machine you must:

Format Hard Disk - If you only reinstall the operating system then static files that do not require registration may have already been installed. Therefore, without formatting the disk there is no guarantee that the installation will work on all machines.

Install Operating System - Make a note of any service packs applied as this must be included in the component's documentation i.e. the Readme file

Install Development Environment - Again, document any service pack installations. Always select the standard installation otherwise certain files may be missing causing erroneous errors when you test.

Test installation - Although we test the product installation thoroughly we recommend you also test the product to your best ability. This will ensure the swift progress of the component through our QA system.

Once the above steps are complete you can image the disk allowing you to re-clean your environment in minutes. Image applications take a snapshot of your clean system, with operating system and development environment installed. This prevents the long cycle of re-installing everything before testing can re-commence. A good practice is to allocate a hard disk per operating system per development environment. As several disks can be installed in one machine, imaging an environment provides an economical and effective solution.

Component Licensing

How do I license my component? - Unlike application licensing, few licensing utilities exist for component licensing. Those that currently exist often operate a 'two-phase unlock' process which requires manual intervention to generate the retail unlock key. However, this form of licensing does not suit 'mass market' adoption. The development of the C-LIC (common licensing) component enables authors to integrate a DLL providing a 'Try-Before-You-Buy' licensing solution.

a) The Common Licensing Problem

Components sold on the open market have are typically 'Black Box' architecture. This means that all functionality is encapsulated and cannot be adapted by the developer except through the

public interface. Because of this, providing an evaluation that allows the developer to 'road test' a component is important when securing a sale. Nowadays, customers expect one download that runs in evaluation mode for a set number of days. Once this evaluation has expired, functionality is disabled until a license key is purchased and entered, unlocking the component into a full retail version. Often the best form of protection is to use a reminder/nag screen that launches each time the calling application runs the component. This prevents users without a license from releasing an application into a commercial environment.

b) C-LIC - The Common Licensing Solution

C-LIC is the ComponentSource license technology used to adapt a full retail product into an evaluation. However, please note the current version does not support copy protection. The C-LIC DLL can be integrated into a majority of languages that support the creation of software components. Its method of working is similar to that used in application software. C-LIC was developed to enable component authors to create a fully or part functioning evaluation protected by a 'nag' screen reminding users that the component is unlicensed. The nag screen allows customers to browse to the relevant product page and purchase the license key used to unlock the product into the full retail component. The license key is provided by ComponentSource and is generated by our own proprietary encryption.

C-LIC can also protect different levels of functionality. For example if your standard version has 10 functions and your professional version 20 functions then the purchase of a standard license will unlock 10 functions only - the other 10 functions will remain in evaluation mode. Please Note: C-LIC does not provide "copy protection".

Conclusion

Build components and enter the component market now!

Many different companies of various sizes from around the world have already created new COM based components and entered the "open market". For example:

- AFD Software, UK - Components for address formatting and zipcoding/postcoding
- AppSoft, South Africa - Components for integrating accounting systems with order processing systems
- BAI, Belgium - Components for CRM and financial services organizations
- EDS, Plano, TX - Components for security, legacy data access and financial application creation

Developer demand for components is currently outstripping supply - as a result an opportunity exists for experts to create components and enter the "open market" for components.

If you have any feedback on this white paper or questions about creating commercial software components email us on: publishers@componentsource.com

Revision History:

First Published: February 29, 2000
Revised: July 15, 2000 (Updated: Component Architectures; Component Languages)
Revised: September 20, 2000 (Updated: Component Languages; Component Characteristics)
Revised: June 24, 2003

Contributions by Steve Teixeira, CTO, Full Moon Interactive

ComponentSource