Creating Commercial Components

(Microsoft® COM and Microsoft® .NET Framework)

●

Technical White Paper

[View Contents]

Date: December 14, 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

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 .NET Framework. 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 Microsoft .NET Framework is an integrated and web-enabled platform that allows component authors to focus on creating reusable components rather than spending time on building complex proprietary "application framework environments" that "lock-in" users. This architecture from Microsoft provides "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 readily available development languages like Microsoft Visual Basic.NET ®, Visual C++.NET, C#, or J#.

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 .NET 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 product's 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 server 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 .NET Framework and its related services introduces 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

The .NET Framework allows 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 Windows Forms Controls. 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.NET web applications, or both, the .NET Framework allows 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

.NET Framework Visual Components is the architecture from Microsoft that allows developers to build server-side user interface components. Utilizing this technology, you are able to create a complete user interface on either a web page or within a Windows application. Let's compare a server-side UI component to a standard client side component:

> **Client-Side Example -** Consider a button bar. Each button has properties - such as color, image, 3D/Not 3D, etc., methods and events - such as a method to switch from 3D to Not3D in the case of a Button_Click event. Buttons can even be combined to form a ToolBar. Since the component resides on the client, it responds quickly to create a rich user interface. However, since they must be installed on each client machine, developing applications that use these components means taking into consideration the size, footprint, and memory and system requirements of the component.

> **Server-Side Example -** A .NET Framework server-side UI component will be able to create the same button bar used above, but the component resides on the server. It can still respond to click events and it can still set component properties. The benefits inherent to this implementation are significant. First, the client application needs only reference the component on the server. This eliminates the need to install a .DLL on every client machine. Second, by offload the component to the server, the requirements for client machines are less stringent. The server is built to be robust with sufficient memory and processor speed to support multiple clients. Additionally, this server-side implementation can be deployed in a distributed application - via Win Forms, or as part of an Internet application via Web Forms and Active Server Pages.NET.

b) Non-Visual Components

Non-visual components do not provide a pre-designed presentation interface to the user. These types of .NET components are known as .NET Components or Classes and only contain functionality exposed to the developer through the programming interface. Unlike visual components this interface is not visible through a property page such or the developer toolbox. 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. 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.

Component Languages

What language do I use? - Practically any. Many development environments support the .NET Framework. There are two classes of languages: .NET Consumers and .NET Extenders. .NET Consumers are able to be deployed on the .NET Platform. They are required to compile to Microsoft Intermediate Language (MSIL is described in depth in the next section). There are many languages that have been updated to be .NET Consumers including: COBOL, APL, Pascal, RPG, FORTRAN, and more. .NET Extenders are able to extend the .NET Framework. Currently, the only development languages that are classified as .NET Extenders are the Visual Studio Suite of programming languages.

a) Visual Basic.NET

Microsoft Visual Basic is the most widely used language for creating software components. Visual Basic.NET will likely be just as popular for creating commercial .NET components. This environment provides all the functionality required for developing and compiling a .NET Assembly. Visual Basic.NET is a language that is easy to use and excellent for rapid development. Additionally, since all .NET languages compile to MSIL (see below) and offer comparable execution speed and capabilities, VB.NET emerges as a first-class player in the .NET arena.

b) Visual C++.NET

Microsoft Visual C++.NET offers greater power and speed of execution over Visual Basic.NET through the use of unmanaged code. Although Visual C++.NET leverages managed code, the developer has the power to choose not to use it in certain circumstances. If the developer chooses to use unmanaged code, he faces the same restrictions as he has always faced in C++, such as security, resource allocation, garbage collection, etc. Security is the most significant consideration. Unmanaged code won t be able to run on many web servers, ASP Service Providers, etc. However, these considerations may be an equitable trade-off for the added speed or flexibility depending upon the requirements of the project. Unless a compelling reason can be found to use unmanaged code, Visual C++.NET developers should use managed code, thus reaping all of the benefits of the .NET Framework.

c) C# (C Sharp)

Microsoft C# is a new development language introduced as part of the .NET Framework. It combines the low-level functionality of Visual C++ with the productivity of higher-level languages like Visual Basic, and all of the advantages of using managed code. C# is designed to allow developers to build robust, object-oriented applications with fewer lines of code. It was designed from the ground up to integrate completely with new web standards such as HTML, XML, and SOAP. It offers a clear advantage over existing development tools that were introduced before the Internet was widely accepted and building "Internet-enabled" applications that leverage this technology became such a major consideration for developers.

d) J# (J Sharp)

Microsoft J# is a new development language that allows Java developers to build components and applications for the .NET Framework. It integrates Java syntax directly into the Visual Studio.NET development environment, and continues to support Visual J++ functionality such as JavaCOM and JDirect. However, J# is not interchangeable with other Java IDEs. Components and applications written in J# will only run in the Microsoft .NET Framework. They will not run on a Java Virtual Machine. However, it does give Java developers a straightforward process with which to port existing Java/Visual J++ components to the .NET Framework, as well as the means to create new .NET components and applications without requiring them to learn a new development language.

e) MSIL

All .NET Consumers and Extenders initially compile to Microsoft Intermediate Language. It is the language spoken by the .NET Common Language Runtime (CLR). This offers the advantage of potentially expanding .NET to other platforms. Indeed, a CLR for Unix is already under development. At Runtime, MSIL is compiled to native code using Just In Time Compilers (JITers). These JITers can be configured for a combination of speed and portability. Depending upon the JIT settings, the MSIL code is compiled to native code sometime between install-time and runtime.

f) Other Languages

As mentioned at the beginning of this section, virtually any language can be modified to act as a .NET Consumer. Compilers are already available for COBOL, APL, RPG, FORTRAN, and more. A distinct advantage to the .NET Common Language Runtime is that developers no longer need to be tied to a single IDE or Platform.

Designing Commercial .NET 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) Code and Metadata

The Microsoft .NET Framework uses metadata to describe a component s entry points. When a

.NET Assembly is compiled to MSIL, the metadata is stored with the component s code inside the .DLL or executable. It describes the component in much the same way as interfaces describe COM components. It is stored in a compact binary format, but can be converted to/from XML Schema or COM Libraries. The functions contained in an the components 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 used to persist data. This persistence can last for only the life of the object, or it can be persisted to a database. Properties are generally represented 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.

Events - Events are messages sent by an object to signify that an action has occurred. They can be triggered by user interaction (such as a mouse click) or in response to program logic, such as providing status information on a method s progress. In most development tools, the handling of events is automatic. .NET uses three elements to provide event functionality: a class providing event data called *Event*EventArgs; an event delegate called *Event*EventHandler, which holds a reference to a method; and a class that raises the event which must provide an event declaration and a method called On*Event* to raise the event.

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

b) Inheritance

The .NET Framework offers true inheritance, whereby a component is defined to inherit the definitions of one class. If the characteristics of the parent change, the subclass that inherits from it will also be changed. For example, if you add a property IsEncrypted to a File component, any subclass based on that component will have an IsEncrypted property. Classes can be created that are incomplete by themselves and are solely for the purpose of being inherited by others. These are called abstract classes. To give further extensibility and customizability to your components, interfaces can be implemented. Interfaces do not support implementation inheritance, and are best used in classes that already have established base classes.

c) Strong Naming

When a .NET Assembly (component) is created, it consists of a strong name. Strong names are created by using a Public/Private Key encryption. The name is generated using your private key and the public key is published with the component. This strong name is how the component is distinguished from other components. The use of strong naming in assemblies allows for multiple versions of the same component to exist side-by-side simultaneously. It also eliminates the need to create Globally Unique Identifiers (GUIDs), and to register those GUIDs, along with component information, in the system registry. Because .NET components are self-describing and do not need to be registered, they can be deployed by simply copying the assemblies to the target machine (known as XCopy Deployment).

d) Error Handling

The .NET Framework provides Structured Exception handling. Standard Exceptions are provided by the runtime and should be used in favor of creating new ones. 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 an exception 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) Interoperability with Threading Models

Designing .NET components for compatibility with existing COM components is an important consideration. 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 used by 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.

The .NET Framework does not use Apartments. All managed objects must use shared resources in a thread-safe manner themselves. However, .NET does have the ability to interoperate with existing COM objects that use Apartment Model Threading. A managed thread can create and enter either an STA or an MTA. You control the type of apartment created by setting the ApartmentState property of the thread.

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.

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. Components can also be created as Web Services. Web Services can be used by a number of clients. These clients can be web-based applications or even other Web Services.

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 .NET 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 .NET 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.NET, Visual C++.NET, C#, etc will ensure more developers are aware of compatibility with their chosen environment and that you are 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 C# 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 Resources 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.

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) 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 six areas detailed below.

Operating System for Deployment - This section covers the different operating systems that your component can run on. Since the .NET Common Language Runtime may eventually be ported to many Operating Systems, this is important to specify from a support perspective. Although theoretically, any .NET component will run on any OS that has the CLR, you may not have that OS in your organization to assist in troubleshooting should a customer experience technical difficulties.

Architecture of Product - The architecture of a component defines the type of system the component is compatible with. For instance, the .NET platform may be used to create components for differing architectures such as 64-bit or Compaq Alpha.

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 - Microsoft .NET components fall into four categories: ASP.NET Server Control, Web Service, .NET Windows Forms Control, and .NET Component or Component Library. This section also allows you to specify if your .NET component consists of 100% managed code. Your component will be receiving exposure in different filtered catalogs aimed at specific audiences.

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. System requirements, languages, etc. are just some of the options that may require inclusion.

Deploying Commercial .NET 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 .NET component into a system requires little more than copying files into directories on disk. However, .NET components can still benefit from integrated installation procedures. Indeed, developers have come to expect a SETUP.EXE that installs the application or component onto their system.

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.

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.

- **Expiration Date -** 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 - The security context that a .NET component operates under depends upon the type of component it is. For example, an ASP.NET Server Control/Web Control is accesses over the Internet by the browser. As such, its security context is dependent on the browser s security zones settings. For Windows Forms Controls and .NET Components or Component Libraries, security is verified through a stepped process:

- **Load Assembly -** The assembly is loaded/downloaded from the intranet/internet server.

- **Gather Evidence -** Evidence is completely extensible. Any object can be a piece of evidence. Evidence only impacts granting of permission if there is a code group membership condition that cares about it.

- **Verify IL -** This is an optional step that verifies the MSIL of the assembly.

- **Load Policy -** The policy determines what the code is permitted to do. It is the policy that grants permission to the assembly.

- **Grant Permission Based on the Policy**
- **Execute code**

c) Component Verification

All .NET components are digitally signed using their strong names. This identifies the component as coming from you. Digital signatures from a certificate authority (CA) are no longer required. Organizations can create their own public/private keys to create the strong names. Depending on the security context, once the component is verified, it will have the following permissions by default, based on where it is deployed:

Local Machine - By default, .NET grants the FullTrust PermissionSet. The component has full access to all machine capabilities. It can execute unmanaged code, and can be installed by simply copying the file.

Local Computer Zone - Full Trust unrestricted.

Intranet Zone - Local Intranet Can read environment variables (limited). Can access the User Interface and isolated storage. Web access to the same site/file read to the same UNC directory.

Intranet Zone - Internet Safe access to User interface and isolated storage. Web access to the same site.

Restricted Zone - None - No authorizations. Component can't run.

MS Strong Name (FX) - Full trust - Unrestricted.

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? The .NET Framework includes licensing functionality. There is a LicenseManager class built into the runtime that supports:

- **Free**
- **One time fee**
- **Per CPU**
- **Per Instance**
- **Monthly Subscription**

LicenseProviders provide the validation logic for the component. Classes are licensed by specifying a LicenseProvider, which developers are able to create themselves. Additional licensing can also be provided by an external source. 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!

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

The Microsoft .NET Framework is currently in BETA release. As .NET matures and the technology changes, this white paper will be updated to reflect relevant information.

Revision History:

First Published: July 9, 2001
Revised: December 14, 2001 - Revisions: Added information on J# Language

ComponentSource