# Creating Commercial Components

CORBA Component Model (CCM)

•

**Technical White Paper**

View Contents

**Date:** December 1, 2000

**Authors:** Andrew Pharoah, ComponentSource
Jon Siegel, Object Management Group
Chris Brooke, ComponentSource

www.componentsource.com

**Email:**

publishers@componentsource.com

**US Headquarters**

**ComponentSource**
3391 Town Point Drive,
Suite 350,
Kennesaw, GA 30144-7083
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 server applications and 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 Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA ) Component Model (CCM ) specification. In the following chapter we discuss the business benefits of using components and identify the functionality suitable for server-side component development in CCM. Following this we detail the CCM architecture and the environment in which these components can be used.

---

# 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)

Traditionally, server applications have been built using proprietary transaction processing monitor (TP Monitor) systems. This made it difficult to write portable, enterprise-class software. With the introduction of the CORBA Component Model, server-side, enterprise software applications may now be created as a collection of software components. These CCM-based applications may now be deployed on any CCM-compliant application server. Increasingly enterprise 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, such as *Credit Card Authenticating* components for E-Business applications. To find out what is in demand visit our Component Request Center: www.componentsource.com/business or look at the Case Studies of Authors who have already entered the 'open market' for components.

### *What is helping make this happen now?*

The open, vendor-neutral CCM specification from the Object Management Group (OMG) is a component model for building server-side, enterprise class applications. In addition to its CORBA foundation, the CCM shares a base architecture with Enterprise JavaBeans™ (EJB), extending this popular environment to programming languages beyond Java™. CCM allows component authors to focus on creating portable and reusable components rather than spending time on building complex proprietary application framework environments

that lock users into a particular technology. The CCM specification requires the application servers to provide a host of services that the CCM-based components may depend upon. Since the services are specified using CORBA technology interfaces expressed in OMG Interface Definition Language (IDL), the component implementation is not tied to any application server vendor's implementation of those services. The CCM specification also enables the application server vendors to provide a robust, scalable, secure and transactional environment to host the components. CCM components may, in principle be implemented in any programming language with an OMG-specified mapping from IDL. Currently, mappings from the CCM-specified IDL extensions have been defined for C++ and Java.

To find out about the CCM architecture and how to design, implement and deploy CCM components - read the remainder of this white paper.

---

# 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. Each component advertises one or more business interfaces. The users (or clients) of the component interact with it only through these interfaces. The clients are completely decoupled from the implementation of the component. The component implementation may be changed or upgraded without affecting the clients. One of the main characteristics of a component is that the business logic is separate from the data that a component manipulates. However, this does not apply to single parameter data that is passed to methods of the interface. For example, in CCM, a special kind of components, known as Entity components, are used to model persistent data. However the underlying data may come from almost any data source that the programmer has tied into the implementation - the client has only an object view of the data!

### 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 Architecture

*Where are components installed?* - The CORBA component container provides robust persistence, transactionality, security, and distributed event-handling to the components installed in it. This is fine for the server side of an application, but too heavyweight for all but the most robust of clients. Therefore, you should write CCM-based components for the *server side only*. The client side of your application may be modular and composed of CORBA objects, but it will *not* contain CCM components.

### Server-Side Components

Compared to the GUI elements that we're used to seeing as client-side components, server-side components are relatively new to the market. The server-side component runtime environment is termed a *container*. The container supports the components installed within it with critical services in two areas:

- First, the container provides key enterprise services: Persistence of an object's state; transactionality; security; and event handling. This makes CCM Components *easier to program*, because the services are provided as run-time rather then coding-time constructs, through high-level interfaces that access CCM-generated code.
- Second, the container manages server-side resources, primarily memory and CPU access, by activating or deactivating the code that executes component functionality as needed, according to patterns selected by the developer. This allows CCM applications to serve *Internet hit rates on enterprise numbers of instances* - that is, CCM applications *scale*.

As we'll show in this white paper, all of the boundaries in the system - both component-to-component and container-to-component - and the services that flow across these boundaries - are well-defined. Well-defined container services and interfaces enable developers to produce components that install neatly into the container, taking advantage of the services and making efficient use of resources such as CPU and memory. Well-defined functional interfaces enable components to work together, with different types from potentially different suppliers assembling into a coherent application. Together, these aspects of the architecture support a dynamic market of third-party components, created by specialists in their functional areas.

**A CCM Example -** The Shopping Cart CCM example presented in CORBA 3 Fundamentals and Programming (Siegel, Jon; John Wiley and Sons, NY, 2000) is an example of a set of CORBA components that execute cooperatively in a CCM server environment. This set of interoperating components encapsulates the functionality of an online shopping application, including representations of the customer and the shopping cart as used by e-businesses. In the past, most online stores had their own proprietary implementations of a shopping cart. A ShoppingCart CCM implementation enables them use a well-tested, well-designed component in a "plug-and-play" fashion - they simply need to integrate and configure the CCM into their applications. (The code in the book is a teaching example only; it would need a lot of work to become the kind of robust, tested component that would do well in an open market!)

## CORBA Technology

CORBA technology has emerged as a very popular environment for building E-business applications. The CORBAservices define a rich set of classes that augment those provided by the container to extend the distributed environment out into the enterprise. CORBA foundations for the CCM include OMG Interface Definition Language (OMG IDL), now an ISO standard; strong typing for both objects and parameters, integrated with the type systems of Java and C++; seamless exception handling across network boundaries; and support for multi-threading.

### Implementation Language Issues

CORBA is a multi-language environment. The CCM standard specifies all the services provided to server-side side components in OMG IDL interfaces. Although mappings from OMG IDL have been defined for eight programming languages, the CCM extensions have been mapped to only two thus far: Java, and C++. So, CORBA components can be programmed in either of these two

languages. CCM clients, on the other hand, can be programmed in C, C++, Java, Ada, COBOL, Smalltalk, Lisp, PL/1, or the scripting languages Python and IDLscript. Clients in any of these languages can invoke operations directly on a CCM server.

### Integration with Enterprise JavaBeans

The CCM specification defines two levels of component: basic, and extended. Basic CCM components have exactly the capabilities of Release 1.1 Enterprise JavaBeans (EJBs), while Extended components add a number of capabilities including distributed event handling, multiple interfaces and navigation, segmented persistence, and more. The basic component environment takes advantage of the EJB parallel, and the requirement that EJBs interoperate using the IIOP protocol, to define an environment where EJBs and CCM Components can be assembled to form integrated applications.

---

# CCM™ Components

## Introduction

The CCM is a specification of a server-side component model for building and deploying enterprise-class applications. The enterprise application developer may build his/her application as a set of interconnected enterprise components and deploy it in a CCM-compliant run-time environment. This environment is structured as a number of containers supporting the different component types (service, session, process, and entity, as we'll explain shortly), each providing enterprise-level services to the components contained within it. The standard also specifies a set of interfaces that developer-written components must implement in order for them to be deployed in an CCM-compliant application server. That is, the CCM server promises a set of services and, in return, expects the components to implement certain interfaces so the server may manage these components. The CCM standard enables the enterprise developer to focus on the actual business logic of the application encoded in the components, leaving the CCM server responsible for the enterprise services it provides: Transaction Management and Concurrency, Persistence, Security Management, Event Handling, Identity (for Entity components), Distribution, and Resource Management. CCM based applications are transactional, secure, robust, scalable, and portable. To understand the problem space CCM addresses, let us consider the motivations for using this technology.

## Why use CCM Components?

To understand the need for CCM, it is useful to understand the relative merits of TP Monitors and server-side systems. Here we present the strengths and weakness of both these architectures.

### Transaction Processing Monitors

Traditionally enterprise-class systems were implemented using systems generally known as Transaction Processing Monitors or TP Monitors. Large scale enterprise applications such as banking, insurance and airline reservation systems are built using TP monitors. Some of the popular TP Monitors are IBM's CICS® and BEA Tuxedo®. TP Monitors were a natural choice for enterprise applications because they handled all the database transactions efficiently and in a manner where the enterprise developer did not have to explicitly write code to manage transactions.

TP Monitors are designed to handle large workloads and manage concurrent access to enterprise application resources. TP Monitors also handle the security management, database access and the network connectivity for the enterprise applications. In other words, TP Monitors provide these services so that an application programmer may focus on implementing the business logic of the application.

In a way, you may think of the TP monitors as an Operating System for business applications. When you use a normal Operating System, you expect a host for services such as virtual memory management, file system management etc. from the system, you do not code for those services in your normal applications. Similarly, enterprise applications may expect to find services pertaining to Transaction, Security, Concurrency, Resource Management etc. from the TP

Monitors.

Given that TP Monitors do so much for an enterprise programmer, why not use them? Why worry about CCM? For all their strengths, typical TP monitors suffer from two major drawbacks. First, most TP monitors do not have a component model. The services are offered, typically, as functions which leads to monolithic applications as opposed to component based applications. It is very hard to replace one service implementation with another. For example, for an e-commerce application, you might want to have the flexibility to replace the credit card processing object with a superior implementation. Lack of an object model prevents you from doing that easily. It also makes it hard for you to implement 'objects' that reside on the server but are dedicated to specific clients and execute programs, on the server, on behalf of their 'owner' clients. A typical example of this kind of application is a Shopping cart or a Mobile Smart Agent.

The second, and more serious, problem with typical TP Monitors is the lack of portability of enterprise applications implemented using them. Enterprise applications implemented on TP Monitors are usually tied to a proprietary API and model. It is usually a large effort to port an enterprise application from TP Monitor system to some other vendor's TP Monitor. The problem arises because there is no standard for TP Monitors. Each TP Monitor may implement all the necessary services that an application might require and use, but since each vendor exposes the services in a proprietary way, the enterprise application becomes non-portable.

### Component Transaction Monitors

Enterprise JavaBeans Components essentially combine the strengths of traditional TP Monitors and CORBA. In other words, using CCM, you get all the benefits of TP Monitors and the portability and component model of CORBA. CCM servers belong to a class of systems commonly known as ***Component Transaction Monitors*** or **CTM**. Using a CCM-compliant server, an developer may build enterprise-class applications rapidly, focusing purely on the business and application logic. All the infrastructure services are now the responsibility of the server and are provided automatically to the application. The developer can configure these services declaratively - the configuration is specified using XML. The enterprise application is implemented as a set of CCM components, with well defined business interfaces, that are deployed on an CCM server. The developer is no longer tied to any one implementation of the application server and may simply deploy her application on any CCM-compliant application server, such as *iPlanet*, *WebLogic, WebSphere* or *iPortal*, without even recompiling the application! The CCM server generates the appropriate objects to provide the enterprise services and ties them with the developer-implemented components during the application deployment.

In summary, you would use the CCM component architecture if you want to build portable, component-based, scalable, secure, transactional and robust enterprise applications rapidly. You would also use components if you want to implement only the business logic and want the application server to handle all the system services.

## 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.

### Components, Assemblies, and Applications

Although you could build an accounting system as a single CCM component with a single component reference, this type of application would not take advantage of the features that make the CCM scale to enterprise and the Internet loads. Instead, good CCM applications consist of some number of component types that work together to provide the total application functionality. In the deployed application (and in the component product offered for sale), each component type is represented by a factory (which we will see is referred to as a component home) which creates instances of its type at run-time as they are needed. For example, an e-Commerce application could consist of a customer component type, a shopping-cart component type, and a checkout component type, which work together to execute the entire shopping trip from customer

registration (which only happens once), through shopping, to checkout and shipping. When an e-commerce site buys this application, they don't get any actual customer, shopping cart, or checkout components - they actually get factories for these three component types. As it runs, the application will create a customer component instance for each new customer that logs in, and a new shopping cart component instance every time a customer starts shopping, and a checkout component instance every time a customer checks out. Customer component instances, which will be entity type components (which we'll explain later in this paper), will last forever since companies love their customers and want them to come back again and again, although the component infrastructure will de-activate and re-activate the instances as needed to conserve memory. Shopping cart instances will live as long as a shopping trip takes; when a customer checks out, his shopping cart is destroyed and its resources reclaimed. Checkout components are created, used once, destroyed and their resources reclaimed. These three patterns (and one more, that we'll present later) let our server use resource in a very parsimonious way, providing users with the ultimate in scalability.

The CCM infrastructure manages resources for the component instances. Because instances may be de-activated and re-activated to allow more of them to fit in available memory, it is important to design them to be a reasonable size. When an e-commerce site using your components grows to have tens of millions of customers, and millions of shopping carts, this will be key to keeping things running on a reasonable amount of hardware!

The combination of components that, working together, provides the functionality of an application is termed an assembly. We'll discuss assemblies further after we've introduced components.

## 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 provide a precise solution rather than one that provides features over and above a basic requirement. For example, a business component that provides addressing services could include various functions such as address duplication, 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 duplication component that incorporated extended functionality such as off-line batch duplication 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, for example.

Confining component scope will help ensure that 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. In addition, efficiency gains may be realized by splitting functionality off into different component types. For example, "use-once" functionality should be split off of service or session component types. 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 factor will allow authors to publish highly focused discrete components and provide customers a wider choice of more efficient implementations.

## Prototype the 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. You may prevent this problem by writing the help file before implementation. This will help you detail a functional specification and pinpoint any areas that could be consolidated or improved upon.

# Architecture of CCM Component Applications

As we've already mentioned, CCM Components are typically combined into an assembly of multiple component types that work together, and this assembly is deployed on CCM-compliant application servers. At runtime, CCM component instances execute within special constructs termed containers. The container is responsible for providing system-level enterprise services to the enterprise components that it manages.

## Basic CCM Concepts

CCM components reside on the server. Their functionality is exposed to their clients and callers through interfaces defined in OMG IDL. The callers may be in the same process space as the server objects or they could be in another process and even another machine. A CCM application will use both:

- The (typically) initial call from a client that initiates a transaction on the server is virtually always a remote call, over the network.
- The execution of the transaction typically involves a number of component types working together, calling each other to execute different parts of the work. These are typically local calls, staying within the same process.

In CORBA, invocation of objects and components is location transparent: That is, a client makes its invocation exactly the same way regardless of whether the target is local or remote. (Only the value assigned to the object reference changes.) This is a tremendous advantage: First of all, it unifies programming since invocations by the remote desktop client are programmed identically with those that involve one component type on the server calling another. And second, it enables the same server-side component implementations to run in either a single-process server, or in one that is split among a number of linked machines to provide load-balancing and fault-tolerance.

OMG IDL compiles into a client stub and server skeleton. On the client side, the stub provides an interface proxy that is called by the client. The stub and Object Request Broker (ORB) work together to marshal input arguments and route the invocation out to the network. When it reaches its destination, the server ORB routes the invocation through the skeleton (which functions, in part, as a server-side proxy), unmarshals the arguments, and delivers them to code that executes the function of the target object. Return values, or exceptions if any were triggered, return via the reverse path.

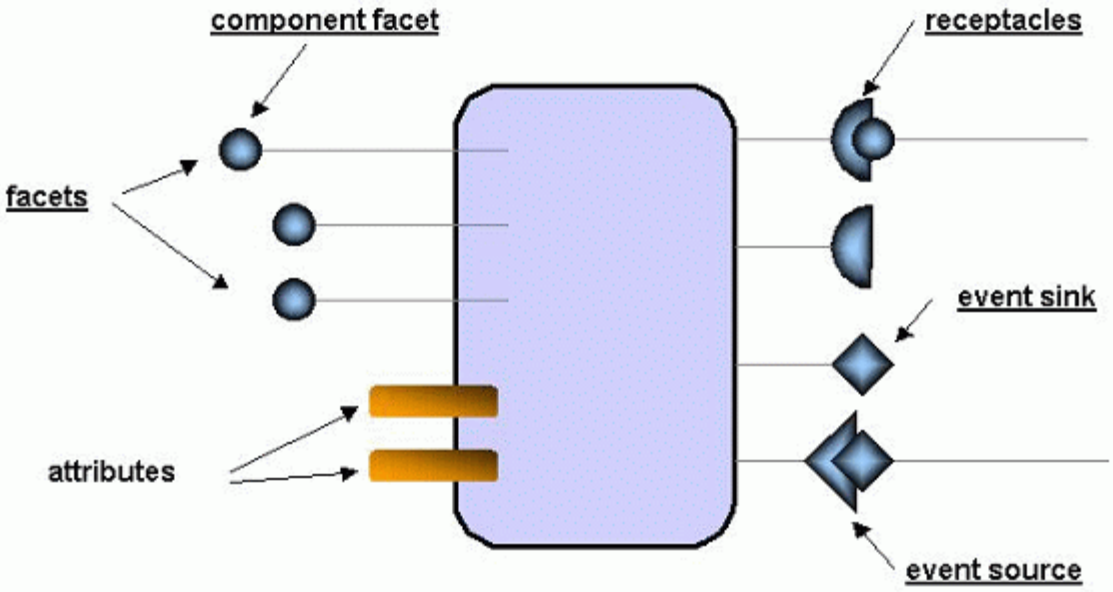


Figure 1: CCM Components' Architectural Features.

CCM Components have four architectural features, shown in figure 1:

- **Facets:** Extended components may bear multiple interfaces, termed *facets*. The CCM environment defines and implements navigation methods among the various facets.
- **Attributes:** Used for configuration, these *attributes* let you write a component in a flexible way. The component is then configured to act in a particular way by setting the value of its configuration attributes at install time. A call to **configuration_complete** tells the system when installation is complete, and the installed component is ready to accept calls.
- **Receptacles:** Client-side interfaces that the component uses to invoke operations on other component types, *receptacles* are supported by the CCM environment. You define which component type is to be called when you configure the *assembly* - that is, the combination of component types that work together as an application. At runtime, the CCM creates the target component and connects it to the receptacle.
- **Event sockets:** The CCM supports a set of named, distributed event channels. Components may be either source or sink for one or more channels.

## Container-Provided Services

As we've mentioned already, the container provides a number of enterprise services to the components installed within it. In addition, the container manages server-side resources - memory and persistent storage - allowing CCM applications to scale to enterprise numbers of objects, and Internet hit rates.

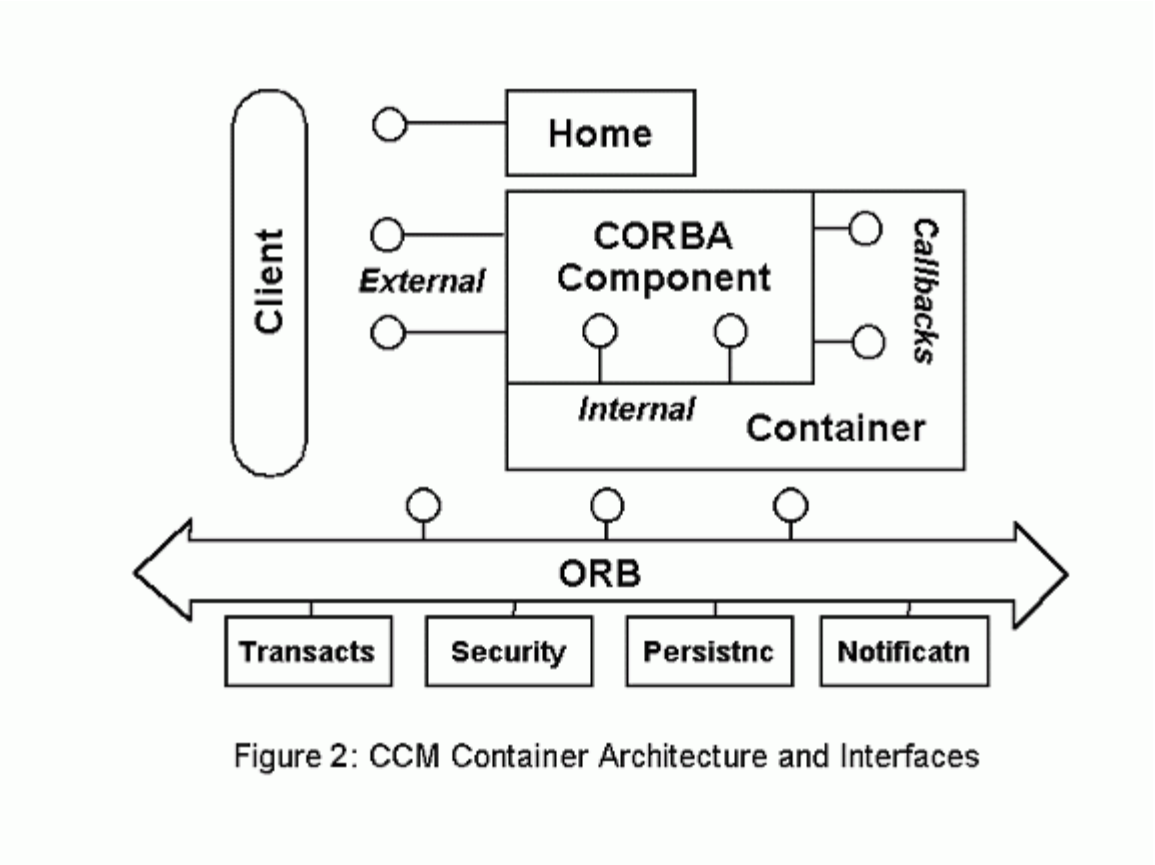


Figure 2: CCM Container Architecture and Interfaces

Figure 2 shows the container and its services:

- **The Component Home**, defined by the CCM and provided by the container, provides factory operations (create, destroy) for its type. For entity components (which we'll define shortly), the Home also maintains a directory which keeps track of the extent (the set of objects that it has created). For other component types, you can easily add code to the home that keeps track of their extent as well. This makes it easy to add operations to the home that operate on the extent.
- **Persistence**: The container provides access to the CORBA Persistent State Service which, through Persistent State Definition Language (PSDL), provides nearly transparent operations to store and recover the persistent state of a component instance. We'll describe this in more detail when we describe activation and de-activation of a servant, shortly.
- **Transactionality**: The container also provides access to a transaction processing system. You can either code control of transactions' begin and end yourself, or leave it to the system by just declaring "transactionality=required" in your deployment configuration file (another CCM feature that we'll get to soon).
- **Security**: The CCM also provides a secure environment which, as you might have guessed, may be controlled via the component configuration file.
- **Events**: Access to event channels is mediated by the container.
- **Callback Interfaces**: These interfaces, defined by the CCM standard, must be borne by the components you write, and implemented by you. They carry out functions necessary to allow the servants - that is, the code that performs the functions of the component - to be activated and deactivated by the container as it manages server resources.

## CCM Services

The CCM application server provides a host of services to enterprise components:

- Transaction Services
- Security Services
- Naming Service
- Persistence
- Resource Management

## Transaction Service

The CCM supports both container-managed transactions and self-managed transactions. Container-managed transactions, the simpler form to program, are declared in a component's deployment descriptor file and implemented entirely by the container. Self-managed transactions are programmed using the container's **UserTransaction** interface or direct calls to the CORBA Transaction Service.

Container-managed transactions may be fine-tuned by setting transaction policies at the component and operation level. Policies parallel those defined in the Enterprise JavaBean specification. Here is the set of CORBA policy attributes and their effects:

| Attribute | Description |
|---|---|
| **Not Supported** | When a caller invokes a method, the caller's transaction, if any, is suspended and is resumed after the method call. |
| **Required** | The component requires a current transaction in order to execute. If the caller supplies a transactional context, the called component executes within the caller's transactional context. If the caller is not in a transactional context, the container starts a new transaction at the beginning of execution and attempts to commit when the operation completes. |
| **Supports** | If the caller supplies a transactional context, the called component becomes part of the caller's transactional context. If the caller is not in a transactional context, the operation executes outside of the scope of any transaction. |
| **RequiresNew** | The component requires its own transaction in order to execute. If the caller supplies a transactional context, the caller's transaction is suspended and a new transaction is created for the duration of this method execution. If the caller does not supply a transactional context, the container creates a new transaction for the duration of this method execution. |
| **Mandatory** | The caller must be in a transactional context before invoking a method on the component. The called component becomes part of the caller's transactional context. If the caller does not supply a transactional context, the container throws a **TRANSACTION_REQUIRED** CORBA exception. |
| **Never** | If the caller supplies a transactional context, the container throws the **INVALID_TRANSACTION** CORBA exception. If the caller does not supply a transactional context, the container does not start a new transaction. |

## Security Service

Security policy is applied consistently to all categories of components. The container relies on CORBA security to consume the security policy declarations from the deployment descriptor and to check the active credentials for invoking operations. The security policy remains in effect until changed by a subsequent invocation on a different component having a different policy.

Access permissions are defined by the deployment descriptor associated with the component. The granularity of permissions must be aligned by the deployer with a set of rights recognized by the installed CORBA security mechanism since it will be used to check permissions at operation invocation time. Access permissions can be defined for any of the component's ports as well as the component's home interface.

## Event Service

CCM components use a simple subset of the CORBA notification service to emit and consume events. The subset can be characterized by the following attributes:

- Events are represented as valuetypes to the component implementor and the component client
- The event data structure is mapped to an any in the body of a structured event presented to and received from CORBA notification.
- The fixed portion of the structured event is added to the event data structure by the container on sending and removed from the event data structure when receiving
- Events have transaction and security policies associated with the component's event ports as defined in the deployment descriptor
- All channel management is implemented by the container, not the component.
- Filters are set administratively by the container, not the component.

Because events can be emitted and consumed by clients as well as component implementations, operations for emitting and consuming events are generated from the specifications in component IDL. The container maps these operations to the CORBA notification service.

## Naming Service

Clients that use the finder method to locate and execute an operation on a CCM component may look it up in the CORBA Naming Service or Trader Service if it is running on the network, or use the component home's **find_by_primary_key** method to locate an entity component. (With the factory method, of course, there is nothing to look up!)

## Persistence Services

The CCM supports persistence using CORBA's Persistent State Service (PSS). Modes of PSS operation support what EJB programmers would recognize as container-managed persistence and self-managed persistence. To use container-managed persistence, the programmer must define his component's state using OMG PSDL (Persistent State Definition Language, a superset of OMG IDL). In this mode, the saving and restoration of an object's state over a deactivation/activation cycle is transparent to the programmer. Under self-managed persistence, the programmer must save state explicitly when his component is called by the container prior to deactivation, and restore it explicitly when called during activation prior to method execution.

## Managing Server-Side Resources

Architecturally, a typical CCM application consists of a set of component factories that create component instances as required, at runtime. For example, an e-commerce application could consist of customer components, shopping cart components, and checkout components. There is a customer component instance for each customer in our database. There is a shopping cart component instance for each customer who happens to be shopping at a particular instant. And, checkout component instances are created when a customer presses the "check out" button. They come into existence, perform the functions necessary for checkout (charging the customer's credit card, notifying the shipping department, creating a bill of lading, etc.), and disappear as soon as they're done.

CORBA entity and process components (which we'll present in more detail in the next section) are persistent - that is, their lifetime spans multiple calls; they may even be called after a server crash. Because there is no client API for component activation/deactivation in the CCM, all the client can do when it wants to invoke an operation on a component instance is do it. This keeps the architecture clean, and the client code simple. (In this discussion, it's important to differentiate between component creation/destruction, which happens only once for each instance and is definitely visible to the client, and activation/deactivation, which may happen repeatedly but is only visible to the server.)

So, if we have several million customers in our database, we wouldn't want every customer component instance and shopping cart instance to be active in memory all the time - this would be very wasteful of resource. So, the CCM runtime manages the instances, activating an instance when an invocation comes in, and deactivating it (and freeing its resources) when it's done. In the CCM (and CORBA in general, through the POA), activation/deactivation is a server-side function, invisible to the client. From the client's point of view, its customer component is always running. The client has no API to activate a servant, as we just pointed out. The client has only to invoke an operation defined in the component's interface for it to execute and the response to come back. The CCM runtime will activate the component instance automatically, if necessary, before passing the invocation to it. So, even though the customer component is not always running, it is always available.

This sophisticated management capability optimizes resource use at runtime. With this capability, CCM servers can be used by any e-business, no matter how large, and no matter how heavily loaded their servers become. Think of how this expands the market potential for your CCM components.

The pattern of resource allocation that we just described - activation-per-invocation - is one of four patterns supported by the CCM, each with its own name and pattern. Here are descriptions of all four patterns:

## Types of CCM Components

The CCM divides components into four categories:

- Service Components
- Session Components
- Process Components
- Entity Components

Of these four, Service and Session components have transient component references - that is, their references are rendered invalid should a server process terminate and be re-started. Even though UPS power and redundant hardware makes server process termination a rare event, it is still poor programming practice to store a transient reference in a permanent store such as a naming or trader service, so Service and Session type components are used only for brief, self-contained operations or functions. In contrast, Process and Entity components, whose references remain valid - even across server restarts - until the instance is explicitly deleted by a client authorized to do so by the security policies in effect at the site, are useful for long-lived records and functions and may usefully represent, for example, customers or bank accounts. If you need to store a component reference in a Naming or Trader service for lookup later, be sure that its component type is either Process or Entity.

**Service Components** have the briefest possible lifetime: a single call. They are useful for functions that are self-contained, such as checking out the contents of a shopping cart (that is, removing the items from inventory, billing the customer's credit card, and generating a bill of lading for the shipping department), or committing a particular transaction type. Because they are cheap to create and destroy, and consume resources only when active, they represent the most efficient component form of these four. By coding as much functionality as you can as service components, you will maximize the load that your component-based server will be able to handle. In particular, "use-once" functionality initiated by longer-lived Process or Entity components should be off-loaded to Service components, instead of coded into the Process or Entity component where it takes up space waiting to be called.

**Session Components** may be called more than once, but do not persist through a server outage: when a server goes down and is brought up again, all of its session components are gone. Even though outages will be rare for a server with redundant hardware and battery power backup, a well-designed application will not use session components for anything except transitory functions such as iterators.

**Process Components** represent, as their name implies, a process with a beginning and an end,

such as applying for a mortgage or bank account. During the process, the component persists reliably, even over a server outage, maintaining its state from one invocation to the next. However, when the process completes, the product is something else - the mortgage or bank account, in our examples. So, the process component creates the product account component and vanishes, freeing up its resources.

**Entity Components** represent the truly persistent items in your application such as your customers, or their mortgages and accounts. Typically (although not always), they will represent data in your database, so their implementation will talk to your database through the CORBA Persistent State Service on the back end (as we showed in Figure 3), and serve these data to your application through the component interface front end. To help you keep track of these important representations, the CCM lets you assign a key to every instance of an entity component, and retrieve instances via their keys.

The CCM server assigns a resource allocation pattern to a component based on its category. Robustness increases as you go down the list, but so does resource usage, so try to stay as close to the top as you can without sacrificing reliability. Keep these categories in mind even at the beginning of your application design stage: If you can keep your entity components small by splitting single-use functionality off of into a session component, you in turn increase the load that your server can handle. Always consider the pros and cons of performance and resource usage when using entity components.

## Creating CCM Components

To define a component in the CCM, the component developer has to define the:

1. Component Interface
2. A Portion of the Home Interface
3. Component Implementation
4. Configuration File
5. Deployment descriptor

### Component Interface

Written in both OMG IDL and CIDL (Component Implementation Definition Language), this interface exposes the business functionality of the enterprise component to the clients that call it. Keep in mind that the "client" in a component-based application may be another component within the same server (or another!), and is not necessarily the remote desktop application. The interface represents the syntax portion of the contract between client and component: it lists the functionality this component provides, but not how the component provides it. From the client programmer's perspective, the interface lists all the business methods of the enterprise component that calling applications may invoke when using this component. Components for the open market are defined as 'Black Box' - that is, all functionality is encapsulated and no implementation code is available to the user. The CCM compiler generates skeletons for the component from this interface declaration.

### Home Interface

This interface exposes the life cycle methods of the component. The CCM automatically generates methods to *create* and *remove* components on the server (but not to activate or deactivate them, since this is transparent to the client). Because only the default constructor is created automatically, you will have to add any others (i.e. differently parameterized constructors) that you need. Because the component home keeps track of its extent, this is the place to declare any operations you might want to define on the set of objects of its type.

***Factory and Finder Methods:*** To use a service or session component, a client will use the factory method to literally create (not activate!) a new instance. The factory will return a component instance reference to the client; the reference is good for a single use of a service component, or repeated calls over a limited time on a session component. Although a client may use the factory method to create a new instance of a process or entity component, most

accesses to process or entity component types will be to existing instances. To access Entity component instances, the client will use the finder method, using the **find_by_primary_key** operation defined on the component's home by the CCM. Process components do not support the find_by_primary_key operation even though their references are persistent, requiring a client to store their references itself for future use. References may be stored in a client cookie, or a Naming or Trader service.

## CCM Implementation

This class contains the real implementation of the CCM component. The CCM implementation class has to implement:

1. All the methods specified in the remote interface
2. Operations defined in the home interface, except for the default constructor and destructor
3. Callback methods specified by the CCM specification so that the CCM container may manage and interact with the enterprise component by invoking these methods.

## Deployment Descriptors

Every CCM needs a deployment description file. The outline of this file is generated automatically by the CIDL compiler, but you will have to fill in details because the CIDL compiler does not have all the information it needs. The deployment description is specified in XML, but will almost surely be generated by a tool (which will probably be provided by your CCM vendor, although you will have to write component-specific information into a file that it will use). Statements in the descriptor file determine the type of container (service, session, process, or entity) the component requires, and its policies regarding transactionality, security, threading, and other container-provided services.

## Property File

This file details component and home attribute settings. As we've mentioned, attributes retain install-time configuration information for a component and, perhaps, its home. The CCM specification prescribes the format and use of the Property File. Properties set during assembly may be overridden at install time.

## Assembly Descriptor

CCM Components assemble into applications. It is also possible to produce a partial assembly - that is, a grouping of a number of component types that work together but do not, by themselves, constitute a complete application. You may decide to add value to your product by marketing a partial assembly of CCM Components. If you do, you will have to produce an assembly descriptor file.

## 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 sent back to the client that called the method. For that reason, you must ensure that the information the client receives is meaningful. A client 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.

CORBA exception handling gives you a lot of help with this. If you declare type-specific exceptions in your IDL, your client may wrap component invocations in a try/catch loop. If you throw the exception in the component, the client will catch it (and its payload) after the invocation returns. This is a very natural way for your client programmer to access unavoidable errors. However, you should return as few errors as possible to the client.

Here are the main techniques for handling errors in CCM.

**Handling Errors Internally** - Handling errors within an CCM 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. To avoid this situation, intercept the error, assess its severity and take corrective action, either by resuming to a specific line of code or by throwing an appropriate exception back to the invoking client.

**Passing Errors Back to the Client** - To return an error back to the calling client, you only need to throw one of the exceptions that you defined in your component's IDL, or one of the CORBA standard exceptions. CORBA will transport the exception and its payload back to the client, where it will be raised in the try/catch loop that wraps the call. CCM clients must be prepared to deal with these IDL-defined exceptions, but it's up to you to create the right ones, and make them easy for client programmers to understand and recover from. It's OK (and usually unavoidable) to pass back errors resulting from bad, inconsistent, or out-of-range parameter values or bad operation sequences, since these errors can only be fixed by the calling client. In general, it's a bad idea to pass back exceptions that the client can not fix.

**Raising Errors from Error Handlers** - The majority of methods you write will contain error handler routines. Where an error handler receives an unexpected error then returning a generic 'unexpected error' exception will not help the client find a solution. If you can't do better than this, the least you must do is return the name of the method that failed and the parameters that were passed to it. The user could then pass this information back to the component author for investigation.

**Handling Errors from Another Component** - If your component invokes a third party CCM, it's good practice to handle all errors (known or unknown) that the secondary component may generate. Developers using your component will have no knowledge of the dependencies your component has unless you document them. If logical dependencies require, you may document these errors and pass them back to the client; otherwise, handle them yourself.

### Threading

The EJB architecture assumes responsibility for managing concurrency. Do not try to explicitly manage threads or thread synchronization as this may interfere with the EJB server's thread management. Also, the EJB server is free to use multiple JVMs and your explicit thread management may not work correctly.

## Roles in CCM Development and Deployment

The CCM specification defines a number of roles in the application development and deployment process. They are:

- Component Developers
- Application Assemblers
- Application Deployers
- Server Providers
- Container Providers
- Administrators

### Component Developer

This role is played by the component developer who specifies the Remote Interface, Home Interface, Component implementation class, and deployment description. The component developer writes files in OMG IDL, OMG PSDL (Persistent State Definition Language), and OMG CIDL (Component Implementation Definition Language). Compilers for the OMG IDL and CIDL

are provided by the vendor of the component runtime, while the compiler for the PDSL comes from the vendor of the Persistent State Service. Files output by these compilations, all in either Java or C++, include stubs and skeletons (Figure 1), plus generated code that implements or triggers resource and persistence management. Code executing business rules is inserted into these files by the component developer, who then compiles the language code. In a subsequent step, using a specialized tool supplied by the component container vendor, the developer packages his implementation along with a configuration file and properties file.

### Application Assembler

There are two places where assembly may be done:

First, you may decide that it makes sense to package and sell functionality implemented as more than one component, working together. In this case, you would assemble these components together and offer the package for sale as a unit. Buyers might deploy and use this package just as you offered it, or have the option of assembling it with additional components (that they wrote themselves, perhaps, or compatible ones that they bought separately either from you or an independent company).

Second, buyers may purchase components separately and assemble them just prior to deployment, combining them either (as in the case we just described) components that they wrote themselves, or compatible components that they purchased separately.

### Application Deployer

An Application Deployer is typically an IT manager who deploys, after modifying the deployment description file, the application on a CCM-compliant application server. Even though many run-time configuration choices were made when the component was packaged, others remain to be set, and many that were set may be overridden.

### Server Provider

This role is played by vendors who implement the application servers based on the CCM specification.

### Container Provider

This role is played by the writers of containers that hold the CCM components. The components themselves reside within a server. This role is typically played by the Server Provider, although some groups are discovering that it is feasible to generate a CCM infrastructure on top of an ORB that they did not write themselves.

### Administrator

This role is played by the CCM server administrator who would be responsible for managing the database connections, Naming and Trading Services, and performance monitoring.

---

# Documenting Commercial EJB 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 may be '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 at: http://www.componentsource.com

### 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. 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 free (though proprietary) viewer that must be downloaded first. PDF files may be generated from any Postscript file, or directly from many word processors. To write a PDF file you will need to download the Adobe® PDF Writer, or use one of a number of other tools Adobe provides for this purpose.

### 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 Data Source 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 have 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 has 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 perceived quality of the final retail product. Where possible, design your demonstration in-line with an accepted standard. This helps build a perception of quality and trust with customers - remember demonstrations can make or break a sale.

## 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 of 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 made available 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.

## 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 understand how the component operates.

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 perceived 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 developers who need to rapidly assemble a component-based solution.

## 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.

**Product 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.

**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 focused on providing a future solution.

### f) Prerequisites

Prerequisites provide the customer with details on required software, product size, required memory, service packs where appropriate, and publicly available drivers. It is worth including the minimum and recommended size when defining memory and hard disk allocation.

---

## Component Testing

*How do I test a component?* - Thorough testing is paramount to the success of a component being accepted 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. 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. This may include the development language for design time testing and the application server for deployment testing.

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 efficient solution.

**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.

---

# Conclusion

Build components and enter the component market now!

Customer 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

**ComponentSource**

As a commercial developer, you can create professional quality software components that provide features like filtering, reverb, dynamics processing, and sample-based looping. You can also create simple or elaborate MIDI-based music synthesizers, as well as more technical audio units such as time and pitch shifters and data format converters. As part of Core Audio and being integral to OS X, audio units offer a development approach for audio plug-ins that excels in terms of performance, robustness, and ease of deployment. Web Components consist of three separate technologies that are used together: Custom Elements. Quite simply, these are fully-valid HTML elements with custom templates, behaviors and tag names (e.g. ) made with a set of JavaScript APIs.Â  Referring to any of these as Web Components is technically accurate because the term itself is a bit overloaded. As a result, each of the technologies can be used independently or combined with any of the others. In other words, they are not mutually exclusive.