

Analyzing Commercial Component Models

Wayne DePrince Jr. & Christine Hofmeister

Computer Science and Engineering Department

Lehigh University

19 Memorial Drive West, Bethlehem, PA 18015 USA

Abstract: Our goal in this paper was to clarify what commercial component models provide to support component-based systems. One of the key motivations was the evident confusion about what a component is, even within a single component model. In this paper we first make a distinction between the functionality of a component model and the mechanisms it uses to support that functionality. We define three kinds of invocation mechanisms for component model services: explicit, implied, and declarative. We describe the elements of a component, show how they support the different invocation mechanisms, and show the role of packaging standards for well-defined components. The component models we examined fall into three distinct groups, with JavaBeans at the weak end, CORBA 2.3, COM/DCOM, and COM+ in the middle, and EJB and .NET at the strong end. The strong component models have configurable execution semantics for operation invocation, declarative service usage, and, most importantly, a well-defined component concept. They also have the properties of the middle group of component models, which use the implied service usage mechanism to support transparent remote communication, and allow an application to use multiple implementations of a component at a time. COM+ falls between the strong and middle groups.

Key words: Software architecture, middleware, component model, component, ADL, CORBA, JavaBeans, EJB, COM, DCOM, COM+, .NET, ActiveX, IDL.

1. INTRODUCTION

Commercial component models have evolved over time, and continue to evolve. Initially (before they were called component models) they provided middleware to simplify the programming burden when communicating across processes and processors [27]. With the advent of name servers, they provided a mechanism for selecting the component implementation at runtime rather than at link time. They began to require that a component follow certain conventions for its instance management and registration with the name server. They also began to support interface specifications written in an IDL (Interface Definition Language). These specifications describe only the interface(s) over which the components communicate—they were

not intended for describing an entire system. Because of these roots, most component models can be used for applications that are not strictly component-based, meaning they consist of some components and some traditional code.

A newer trend is for the component models to provide additional services tailored for a particular application domain, such as enterprise applications. Originally the developer invoked services such as security, transactions, and persistence directly in the implementation code, but now some component models allow the use of these services to be specified declaratively, with the invocation details provided by the component model. Integrated Development Environments (IDEs) are now available for most component models; these tools allow the application to be assembled by configuring and connecting previously-built components.

Given this continual evolution of component models, architects are faced with the difficult task of choosing an appropriate technology for their system. The key issues are: functionality; mechanisms for achieving this functionality; environment (languages and platforms supported, interoperability with existing software); application needs; and organization needs. For the mechanisms, we distinguish between the basic mechanism, meaning the basic approach taken, and the language-specific details needed for each component model.

One basic kind of literature related to component-based systems is information about how to use a particular component model. Because of the first kind of literature, the functionality and environment supported by each component model are generally well understood, with a few exceptions that we will discuss later. This literature also explains the language-specific mechanisms used by the component model. [4,5,6,7,11,12,17,29,30]

A second category of literature provides information about how to choose among component models, generally through pairwise comparisons. These articles cover most aspects of the functionality and environment, but compare mechanisms only at the detail level. This makes comparison difficult because the similarities of the mechanisms are lost in the language-level details [2,3,15,16,21,22].

The most difficult information for architects to acquire is how well a given component model will satisfy the application and organization needs. These comparison articles usually try to provide some information about application needs, but this is clearly at best domain-specific, not application-specific. None of the literature has attempted to help the architect assess the technology according to their organizational needs.

Our contribution to this problem is to clarify the kinds of information needed by the architect, to reveal some functional differences we believe are

not well understood, and to compare the basic mechanisms used by various component models.

The lack of understanding of the basic mechanisms probably contributes to the disagreement about which variants to compare [21]. The unstated goal of the comparison articles is to compare component models that use the same basic mechanism. For example, to compare automatic persistence and transaction support, the comparison should be among COM+, .NET, EJB, and CORBA 3.0; or between COM/MTS and CORBA 2.x.

This lack of understanding is also probably the cause of authors misrepresenting a mechanism as the presence or absence of a service. For example, event support is often claimed to be a fundamental difference among the component models. The reality is that in all cases event communication is achieved by combining standard procedure-call communication in a particular pattern, with certain naming conventions. Some component models then have an IDE that recognizes the event conventions, and in turn can generate them. This is a difference in basic mechanism, not in functionality.

An important related body of work is Architecture Description Languages (ADLs). Both users of component models and users of ADLs speak of building component-based systems with their respective technology. But as is often noted [e.g. in 18], the IDLs of component models lack the ability to describe:

- required interfaces (they support only provided interfaces),
- a system configuration (element types, instances, and connections)
- and first class connectors.

In this paper we first summarize the key features of component models from the three families, and clarify the distinction between the services available and the mechanisms for invoking those services (Section 2). The next two sections examine the elements of a component and the mechanisms for invoking services in each of the component models. For this analysis we use multiple architecture views: a module, execution, and code architecture view [14]. Section 5 discusses the scope and granularity of a component in the various models, and how packaging standards contribute to a well-defined component. Section 6 discusses related work, and Section 7 concludes the paper.

2. COMPONENT MODEL FEATURES

The component models we examine in detail in this paper are: JavaBeans, CORBA 2.3, COM/DCOM, COM+, .NET, and EJB [4,5,6,11,17,12]. We mention CORBA 3.0 in a few places, but do not give a

full comparison since it was only recently accepted as a specification and implementations are not yet available. [7,24]. ActiveX is not considered separately since it is simply an additional set of conventions imposed on COM components [30]. The .NET platform supports applications that are a mix of different elements: COM+ components, simple classes, and a new kind of .NET component [29]. When we refer to the .NET component model we mean the parts of the platform that support .NET components. Similarly, the J2EE platform supports applications composed of EJB components, JavaBeans, and Java classes, so we consider these separately.

The next two tables summarize some key features of these six component models. The first part of *Table 1* highlights some of the important functional features of the component models.

Table 1. Functional Features and Some Basic Mechanisms of Component Models

	JavaBeans	CORBA 2.3	COM/DCOM, COM+	EJB	.NET
Functionality					
Remote communication	use RMI services	transparent	transparent	transparent	transparent
Name Service	none	public name	public name	public name	public name
Execution semantics of invoked comp. operations	executes in caller's thread	one thread per invocation; executes in comp. context	two choices (like JavaBeans or like CORBA)	dictated by bean type (entity, session, ...)	various choices
Other services	see <i>Table 2</i>				
Mechanism					
Interface specification	implicit or explicit interface	language-independent IDL	language-independent IDL	explicit Java interface	explicit language interface
Granularity of component	class (typically)	various	DLL/EXE (typically)	EJB-jar file	Assembly DLL file
Component packaging standards	none	none	none	EJB-jar file	Assembly DLL file

The Name Service, which JavaBeans lacks, provides a convenient way to find the component implementation associated with a particular public name. But its greater significance is that it allows an application to use multiple different implementations of a component at one time.

The execution semantics of an invoked component operation describes the context in which that operation executes: it could execute in the caller's thread or the callee's (component's) context, where there could be one thread reserved for each caller, or new thread could be used for each

operation invocation. This issue of execution semantics is usually discussed only when a component model provides choices, but it is rarely mentioned when the semantics is dictated by the component model. This can lead to incorrect assumptions about execution behavior, causing unexpected problems at runtime.

The second part of the table describes some basic mechanisms of the component models. In all models except JavaBeans a component must have an explicit interface. As we will discuss in Section 5, a component interface is important for determining the scope of a component. The component granularity and packaging standards will also be discussed in Section 5. The lack of a standard for packaging (preparing and delivering) creates ambiguity about the granularity of a component: in some cases multiple granularities are possible, and in other cases it simply causes confusion.

Table 2 summarizes the services provided by each component model and the basic mechanism for invoking them. In JavaBeans all services must be invoked explicitly in the source code (Explicit service usage).

Table 2. Component Model Services and their Invocation Mechanisms

SERVICE	JavaBeans	CORBA 2.3, COM/DCOM	EJB, COM+, .NET
Database Services	Explicit	Explicit	Declarative
Security	Explicit	Explicit	Declarative
Remote Comm. – proc call	Explicit	<i>Implied</i>	<i>Implied</i>
Remote Comm. – msg	Explicit	<i>Implied</i>	<i>Implied</i>
Remote Comm. – event	Explicit	Explicit	<i>Implied</i>
Exceptions	Explicit	Explicit	Explicit
Activation	N/A	Explicit	Explicit (COM+) Declarative (EJB, .NET)
Registration	N/A	Explicit	Declarative
Lookup	N/A	Explicit	Explicit (EJB, COM+) Declarative (.NET)

The other five component models all use what we call Implied service usage for remote procedure call and message communication. To the caller of a component's operations, it looks like a local operation invocation; the use of the remote communication service is implied. The code needed to support this is provided automatically by the component model.

The third type of invocation mechanism is Declarative service usage. Here the developer declares a component to have certain properties, which then imply the use of supporting services when the component is used. Here also the supporting code is provided automatically by the component model.

Implied and declarative service usage have clear advantages over explicit service usage. Because most of the details are generated, they are easier to

use and less error-prone. They also have the advantage of forcing a separation of concerns: because it's generated, the code needed to accomplish the service invocation is necessarily separated from the rest of the application code.

Although the trend in component models is to move services to the declarative category, not all services are amenable to this mechanism. The use of services like remote communication must be embedded directly in the application code; they can't be specified as a property of the component or its operations. So for these kinds of services the declarative mechanism is not possible.

3. ELEMENTS OF A COMPONENT

The module architecture view describes the static structure of a system: the subsystems, layers, modules, interfaces, and their relationships. This is useful for examining the various elements that make up a component and the basic mechanisms used to support Implied and Declarative service usage.

Figure 1 shows the elements of a component and their relationships. It applies to all the component models except JavaBeans.

A component has an explicit interface used by others to access its services. We call this the component interface, and in *Figure 1* it is represented by *IMyC*. This interface is not an IDL specification; it is the programming language-specific interface (generated from the IDL for those component models that use an IDL).

Interface *IMyC* is implemented by the module *MyC*, but as shown in the figure, both of these are partitioned into two parts to distinguish the core services (*IMyC_svcs*, *MyC_svcs*) from certain services required by the component model (*IMyC_req*, *MyC_req*). In COM/DCOM/COM+ the required services are implemented by the developer, and in CORBA, EJB, and .NET they are provided by the component model.

It is often the case that parts of a component implementation are unique to it and parts are potentially used in multiple components. The unique parts are represented by *MyC*, which may use multiple *Other* modules to implement the component's services.

The remaining four modules in *Figure 1* are the infrastructure of the component: *MyC_skel*, *MyC_wrap*, *Activator*, and *Registrar*. Module *MyC_skel* supports the Implied service usage mechanism. It is discussed in more detail in the following section.

The component models with Declarative service usage (EJB, COM+, .NET) put support for declarative services in a wrapper or container, represented here by module *MyC_wrap*. The wrapper both implements and

uses interface IMyC: it intercepts calls to services (thus implementing IMyC), does some processing, then forwards the calls to MyC (thus using IMyC). It is also provided automatically by the component model.

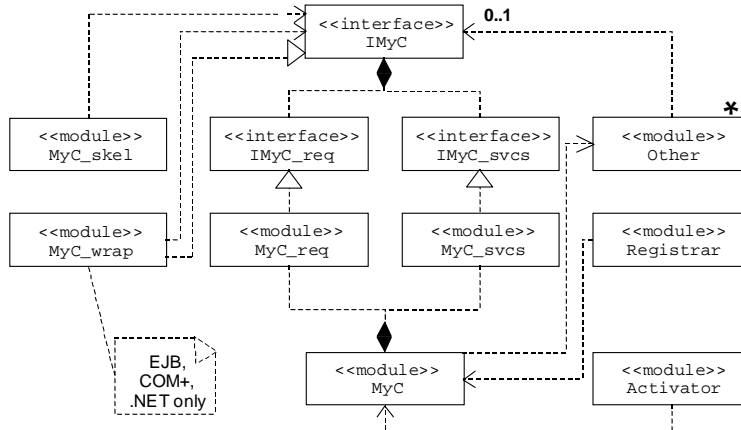


Figure 1. Module architecture view: elements of a component

The Activator is responsible for starting up the component instance. The Registrar is responsible for giving the Name Service a public name for this component implementation. For COM/DCOM/COM+ this could be an installation script, a registry file, or a configuration file. In EJB the Activator and Registrar are provided as part of MyC_wrap, and in CORBA the developer provides them.

JavaBeans does not have anything corresponding to IMyC_req and MyC_req, since it requires no services beyond what those necessary for any Java class. MyC_skel and MyC_wrap are not relevant because JavaBeans does not support Implied or Declarative service usage. It is not possible to give a public name to a JavaBean; it is always known by its class name.

4. IMPLIED SERVICE USAGE

The component models with Implied service usage use it for remote communication, so to explain this mechanism we add a remote client component, MyClient, that uses the operations of component MyC.

MyClient uses the operations specified in IMyC, but before calling them it must locate an implementation of component MyC. In Figure 2 this functionality is in module Lookup. It uses MyCPublicName and the component model services (Name Service) to find MyC.

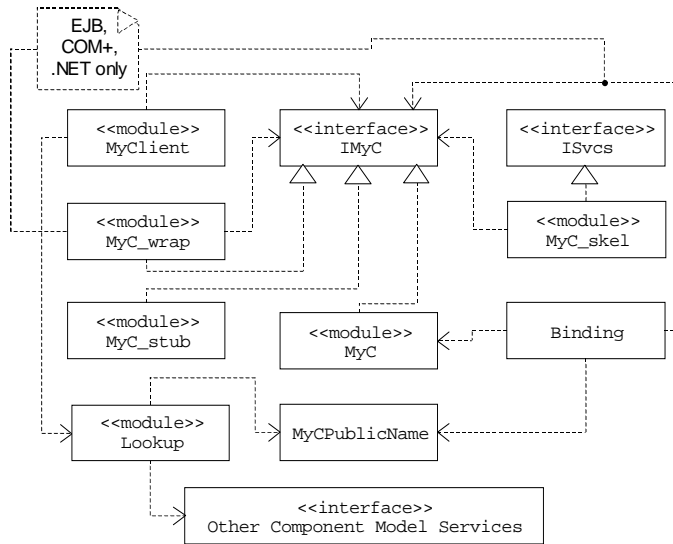


Figure 2. Module architecture view of infrastructure for Implied service usage

Figure 2 shows an additional realization of IMyC: MyC_stub. This realization is generated by the component model and linked in with MyClient, so it is executed whenever MyClient uses an operation in IMyC. MyC_stub routes the request to the component, where MyC_skel receives the request via its generic interface ISvcs. MyC_skel in turn invokes the appropriate service in IMyC, and now the implementation MyC executes (perhaps after passing through MyC_wrap).

Thus the client infrastructure includes modules Lookup and MyC_stub (for Implied service usage). The component infrastructure is composed of modules Activator, Registrar, MyC_skel (for Implied service usage), and MyC_wrap (for Declarative service usage).

The execution architecture view explains how modules are mapped to processes, tasks, threads, etc.; how these communicate at runtime; and how they are mapped to physical resources. Because of limited space here we describe the execution architecture rather than showing a diagram.

For the modules described in Figure 1 and Figure 2 there are many possible mappings to execution elements, so we give only one example. The example uses CORBA, although except for JavaBeans the other component models would be very similar. Modules MyClient, MyC_stub, and Lookup are linked together and run in a process on one host. Lookup and MyC_stub communicate with the Component Model Services, which are accessed as a shared library.

It communicates via IIOP with a similar shared library on the host where the component is running. The elements of the component are linked into an executable file and a shared library. Only MyC_skel, the Activator, and the Registrar access the Component Model Services.

5. COMPONENT SCOPE AND GRANULARITY

Section 3 described the elements of a component:

- module MyC which realizes interface IMyC
- infrastructure modules Activator, Registrar, MyC_skel, MyC_wrap
- additional helper modules Other

This section examines the scope and granularity of a component in these component models. For scope the question is which of the above elements are packaged as part of the component. For granularity the question is whether a component is a single class, a single package, or a set of packages.

To address the question of scope, first we note the distinction between a component interface and a local interface. The two different kinds of interfaces lead to two different kinds of dependencies. One is the local, programming-language-level dependency that arises when one class depends on another class or interface. These can be resolved statically by a linker. This is the only kind of dependency there is in JavaBeans.

The other kind of dependency, a component dependency, is across a component interface like IMyC. A component dependency is resolved at runtime using the Name Service. We can use this interface distinction to determine the Other modules: they are all modules on which MyC has a local dependency.

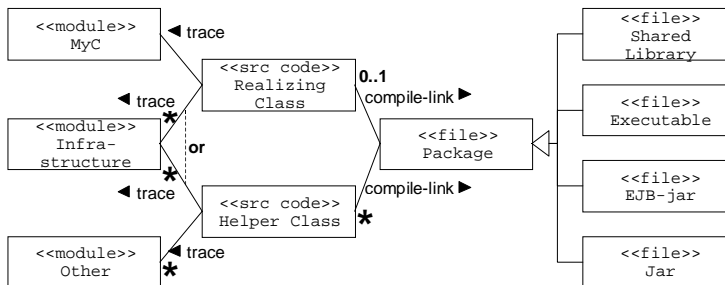


Figure 3. Code architecture view: mapping modules to implementation classes and files

Next we look at how the modules can be mapped to implementation classes and packages, as summarized in *Figure 3*. In this diagram, which is part of the code architecture view, multiplicities that are unmarked are

assumed to be 1. The package file is the artifact used to deliver the component.

The implementation class for module MyC is called the Realizing Class because this class realizes IMyC. The Infrastructure modules could each be mapped to a separate Helper Class, several could be grouped into one Helper Class, or they could be inserted into the Realizing Class. Similarly the Other modules could be mapped into one or more Helper Classes.

The Realizing and Helper classes can be compiled/linked into one or more package files. So a package file can contain at most one Realizing Class, but multiple Helper Classes.

Now we look at the typical scope and granularity of a component in each of the component models. At one extreme is JavaBeans. Here a component interface is no different from a local interface, so there is no difference between a realizing class and helper class. There is no standard mechanism for determining the scope of a component, and the usual approach is to consider each class as a separate component. Thus in JavaBeans a component is typically a single class, the “realizing class” (*Table 3*).

Table 3. Scope and Granularity of a Component

	Deployable unit	Realizing package	Realizing class
Granularity	set of package files	single package file	single class
Typical for these component models	EJB, .NET	COM/ DCOM, COM+, CORBA	JavaBeans
Contains the Realizing Class	always	always	always
Contains Helper Classes	always	possibly	never
Contains Infrastructure def.	always	possibly	never

At the other extreme are EJB and .NET. In both of these there is a packaging standard that specifies a deployable unit: EJB uses EJB-jar files, and .NET uses the Assembly DLL. The granularity of the component is one or more package files, but the set of them must contain the realizing class and all helper classes. The infrastructure is generally defined declaratively, and it is also part of the deployable unit, although sometimes this information is added at deployment time.

The other component models are between the two extremes. For COM/DCOM and COM+ a component is a single package file, the one containing the realizing class. We call this a realizing package. There is no standard dictating how the helper classes and infrastructure should be delivered. These can be in the realizing package or in other packages. If they are in separate packages, they are not considered to be part of the component, even though the component has local dependencies on them.

For CORBA the typical scope and granularity of a component is less clear. It is often treated like a COM component, but given the complete lack of guidance, any of the choices is possible.

Given the goal of delivering a component complete with everything it needs to execute, the deployable unit from *Table 3* makes the most sense. This is not an option for JavaBeans, but CORBA, COM/DCOM, and COM+ could by convention make the deployable unit the standard component scope and granularity. EJB and .NET have already done this by providing standard mechanisms for a deployable unit.

6. RELATED WORK

The first set of related work compares component models with each other. Comparisons of two or three component models are given in [1],[3],[10],[21],[22]. However, these comparisons evaluate the component models and middlewares by comparing programming features, applications supported, and implementation mechanisms. We compared basic mechanisms and also looked at a greater number of component models.

Other comparisons of component models emphasize as we do the commonalities between them. Kobryn uses UML to compare features of the enterprise component models EJB and COM+, and also to evaluate the ability of UML to model component models [15].

Krieger and Adler in [16] present a unified description of JavaBeans and DCOM. Although we have been emphasizing how different JavaBeans is from the other component models, their unified description works because they focus on the facilities presented to the developer via an IDE.

In [1], the authors identify trends in component based software engineering and define common aspects of a component model. However, their analysis is primarily concerned with component composition and packaging in frameworks, while we focused understanding how components differ across component models.

The second group of related work compares component models and ADLs. The authors in [19] argue the need for ideas from both software architecture research and commercial component models to adequately address system-wide aspects of complex distributed software systems. Cugola et. al. focus on event-based architectures, comparing the level of support found in various middlewares, languages, and ADLs [8]. Stuurman in [26] makes a mapping from ADLs to JavaBeans and vice versa, matching structural information between the two and mapping the visual IDE of JavaBeans to a type of ADL configuration language.

Other papers do not make direct comparisons of component models and ADLs, but explore their utility and deficiencies by using one to implement or describe the other. Magee et. al [18] use CORBA as the underlying infrastructure for the ADL Darwin. They succeed in mapping Darwin's features into CORBA (by generating IDL and implementation details), in spite of the noted limitations of CORBA.

In [9] ADLs are used to describe two component models, JEDI and C2, although the term "middleware infrastructure" is used rather than "component model." They view a component model as defining or inducing an architectural style. Rather than comparing the features of JEDI and C2 directly with the features of ADLs, they instead use the ADLs to describe an architectural style, one that captures the features of JEDI or C2. This gave them insights into how readily the ADLs can describe these styles, but results in a less direct comparison of the component models.

Finally, in [23] the authors adapt component models to give them the ability to describe architectural styles. They use tools and wrappers to add support for explicit connectors and architectural styles.

7. CONCLUSION

We conclude by classifying the six component models from strong to weak (*Table 4*). In most cases a component model fits well in a single category, but there are a few exceptions that are listed as footnotes. Almost all of the exceptions are for COM+, which has characteristics of both a strong and average component model.

The first consideration is whether the scope and granularity of a component is well-defined. The strong component models have a packaging standard of a deployable unit, which contains

- the class realizing the component interface,
- all helper classes (those on which the realizing class has a direct or indirect local dependency)
- the infrastructure for activation, registration, implied service usage, and declarative service usage

All but the weak component models have a concept of a component interface, sometimes specified via an IDL, but always supporting a set of standard required services (IMyC_req). These also support explicit naming of a component using a public name that is different from the interface name. This allows applications to use multiple different implementations of a component at the same time, since each has a different public name.

In the strong component models, the execution semantics of invoked operations is well-specified and can be configured. It fixed in the other

component models, and is often not well understood. The weak component models execute the component's operations in the caller's thread, so there is no concurrency.

Table 4. Classification of Component Models

	Strong	Average	Weak
Component model	EJB, .NET	COM/DCOM, COM+, CORBA	JavaBeans
Well-defined component	Yes – packaging standard: deployable unit	No – convention: realizing package	No – convention: realizing class
Component interface	Yes – various mechanisms	Yes – various mechanisms	No – local interfaces only
Multiple implementations within an application	Yes – Name Service binds public name, interface, implementation ¹	Yes – Name Service binds public name and implementation	No – no concept of public name
Execution semantics of invoked operations	Configurable ²	fixed – invocations execute concurrently in component context	fixed – invocations execute in caller's context
Activation, Registration, Lookup	Yes – Declarative service usage ³	Yes – Explicit service usage ⁴	No – not applicable
Remote communication	Yes – via Implied service usage	Yes – via Implied service usage	Yes – via Explicit service usage
Declarative service usage for database services, security	Yes – use container concept (MyC_wrap) ⁵	No	No
¹ COM+ ² COM/DCOM, COM+ ³ COM+ Registration ⁴ EJB Lookup ⁵ COM+			

Finally, the component models differ in their mechanisms for invoking services. The strong component models support declarative service usage for activation, registration, lookup, database services, and security. All but the weak component models support implied service usage to achieve transparent remote communication. In the weak component models, all of these services must be invoked explicitly.

COM+ straddles the strong and average categories, but we ultimately classified it as an average component model because of the importance of having a well-defined component, which it lacks.

Clearly the direction of commercial component models is toward the strong end, with EJB and .NET components. However, both of these are embedded within platforms that allow or even encourage applications to combine components from the strong, average, and weak categories. This could be simply for backward compatibility, but this is not how the companies characterize it in their literature. It seems more likely to be a conscious decision to mix component models within an application.

It is interesting to note that almost none of the component models are addressing the key limitations identified by researchers in the architecture community. Only CORBA 3.0 is adding required interfaces. All focus on IDEs to facilitate system configuration, and none are moving toward first class connectors.

8. REFERENCES

1. Bachman, F., Bass, L., Buhman, C., Santiago, C., Long, F., Robert, J., Seacord, R., Wallnau, K. Volume II: Technical Concepts of Component-Based Software Engineering. CMU/SEI-2000-TR-208. May 2000.
2. Birngruber, D., Kurschl, W., Sametinger, J. Comparison of JavaBeans and ActiveX – A Case Study. In *STJA 99, Smalltalk und Java in Industrie und Ausbildung*, Erfurt, Germany, September 28-30, 1999.
3. Chung, P., Huang, Y., Yajnik, S., Liang, D., Shih, J., Wang, C., Wang, Y. DCOM and CORBA Side by Side, Step by Step, and Layer by Layer. <http://research.microsoft.com/~ymwang/papers/HTML/DCOMnCORBA/S.html>.
4. COM/DCOM 1.0 Spec. <http://www.microsoft.com/com/resources/specs.asp>.
5. COM+. <http://www.microsoft.com/com/tech/COMPlus.asp>.
6. CORBA 2.3 full spec. <ftp://ftp.omg.org/pub/docs/formal/98-12-01.pdf>.
7. CORBA 3.0 – CORBA Component Model. <http://www.omg.org/cgi-bin/doc?orbos/99-07-01.pdf>.
8. Cugola, G., Di Nitto, E., Fugetta, A. Exploiting an event-based infrastructure to develop complex distributed systems. In *20th International Conference on Software Engineering Proceedings (ICSE 98)*, Kyoto, Japan, April 1998, 261-270.
9. Di Nitto, E., Rosenblum, D. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In *Proceedings of the 21st ICSE (ICSE 99)*, Los Angeles CA, 1999, ACM Press, 13-22.
10. Emmerich, W. Software Engineering and Middleware: A Roadmap. In *The Future of Software Engineering – 22nd Int. Conf. on Software Engineering (ICSE2000)*, ACM Press, May 2000, 117-129.
11. Englander, R. *Developing JavaBeans*. O'Reilly, Sebastopol CA, 1997.
12. Enterprise JavaBeans 2.0 Specification - Final Release. <http://java.sun.com/products/ejb/docs.html#specs>.
13. Garlan, D., Monroe, R., Wile, D. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*, eds. Leavens, Sitaraman. Cambridge Univ. Press 2000, 47-68.
14. Hofmeister, C., Nord, R., Soni, D. *Applied Software Architecture*. Addison-Wesley, Reading MA, 2000.

15. Kobryn, K. Modeling Components and Frameworks with UML. In *Commun. ACM*, Vol. 43, No. 10, 31-38.
16. Krieger, D., Adler, R. The Emergence of Distributed Component Platforms. In *IEEE Computer Magazine*, Vol. 31, No. 3, Mar 98, 43-53.
17. JavaBeans 1.01 <http://java.sun.com/products/javabeans/docs/beans.101.pdf>.
18. Magee, J., Tseng, A., Kramer, J. Composing Distributed Objects in CORBA. In *Proceedings of 3rd International Symposium on Autonomous Decentralized Systems (ISADS 97)*, Berlin Germany, April 1997.
19. Oreizy, P., Medvidovic, N., Taylor, R., Rosenblum, D. Software Architecture and Component Technologies: Bridging the Gap. In *Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures*, Monterey, CA, January 1998.
20. Oreizy, P., Rosenblum, D., Taylor, R. On the Role of Connectors in Modeling and Implementing Software Architectures. In *Technical Report UCI-ICS-9804*, Department of Information and Computer Science, University of California, Irvine, Feb. 1998.
21. Raj G. A Detailed Comparison of CORBA, DCOM, and Java/RMI. <http://www.execpc.com/~gopalan/misc/compare.html>.
22. Raj G. A Detailed Comparison of Enterprise JavaBeans (EJB) & The Microsoft Transaction Server (MTS) Models. <http://members.tripod.com/gsraj/misc/ejbmts/ejbmtscomp.html>.
23. Rosenblum, D., Natarajan, R. Supporting Architectural Concerns in Component Interoperability Standards. To appear in *IEE Proc – Software Special Issue on Component-Based Software Engineering*, 2000.
24. Ruiz, Diego Sevilla. Current and expected CCM implementations. <http://ditec.um.es/~dsevilla/ccm/#impl> .
25. Shaw, M. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *Studies of Software Design, Proceedings of a 993 Workshop, Lecture Notes in Computer Science*, ed. D.A. Lamb. No. 1078, Springer-Verlag 1996.
26. Stuurman, S. Software Architecture and JavaBeans. In *Proceedings of WICSAI, the 1st First Working IFIP Conference on Software Architecture*, ed. P. Donohoe. Kluwer, Boston MA, 1999, 183-199.
27. Szyperski, Clemens. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Harlow England, 1999.
28. Wang, N., Schmidt, D., O’Ryan, C. Overview of the CORBA Component Model. In *Component-based Software Engineering: Putting the Pieces Together*, G. Heineman and W. Councill, eds. Addison-Wesley, Boston MA, 2001, 557-57.
29. .NET. <http://www.microsoft.com/net/> .
30. ActiveX. <http://www.microsoft.com/com/tech/activex.asp> .