ELSEVIER

Information and Software Technology 44 (2002) 351–359

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

An evaluation of the impact of component-based architectures on software reusability

Kevin McArthur[a], Hossein Saiedian[b,*], Mansour Zand[a]

[a]Department of Computer Science, University of Nebraska, Omaha, NE 68182, USA
[b]Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS 66045, USA

Received 10 April 2001; revised 5 November 2001; accepted 6 February 2002

Abstract

Component-based software development offers a promising solution to the production of complex distributed large-scale software systems. Development for reuse—the production of reusable components—and development with reuse—the production of systems with reusable components—provide the characteristics necessary to break the complexity of large-scale distributed systems. Two criteria for reuse in component-based architectures (CBAs) include inter-operability (focus on development for reuse) and integration (focus on development with reuse). Interoperability concerns how well components interact and integration defines how well components plug and play. The objective of this work is to evaluate the impact of three popular CBAs, namely, Enterprise Java Beans, Distributed interNetwork Architecture, and Object Management Architecture on reusability. A framework is introduced for a systematic and comprehensive analysis and evaluation of CBAs. The proposed framework is used to determine which of the above architectures more effectively addresses integration and interoperability. The results allow businesses to determine which CBA, of the above three, is ideal for reuse for a particular application. Further research opportunities in this area are discussed at the end. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Component-based software development; CORBA (Common Object Request Broker Architecture); Microsoft's distributed component object model; Remote method invocation; Software architecture; Software reusability

1. Introduction

Garlan and Shaw [8] define the architecture of a software system in terms of its computational components and interactions among those components. Architectural styles differ in the kinds of components they use and in the way those components interact with each other. The framework used in their book is to treat architecture of a specific system as a collection of computational components—or simply components—together with a description of the interactions among these components—the connectors. An architectural style defines a family of the above collections and interactions in terms of a pattern of structural organization. It specifies a consistent list of components and connector types, along with a set of constraints on how they can be glued together. There can also exist one or more semantic models that specify how to determine a system's overall properties from the properties of its parts.

Components are such things as clients and servers, databases, filters, and layers in hierarchical system. The interactions between these components can be as simple as a procedure call and shared variable access. However, the interactions can also be complex and semantically rich, like client/server protocols, database accessing protocols, asynchronous event multicast, and piped streams. While architectures have existed for some time now, the specifications of the components and their interactions are largely informal and typical. Shaw and Garlan mention the typical box and line type drawings with some informal descriptions of the meanings behind the symbols and the reasons for the specific choice of components and interactions.

Problem definition: The primary purpose of this paper is to define and present an analysis and evaluation of the impact of component-based architectures (CBAs) on software reusability. A comparison of CBAs can help determine which one is ideal for a reuse driven organization or for a particular application developed with reuse in mind. We have developed a framework for evaluating the CBAs. The framework has three elements:

- the overall system model—includes the CBA's facilities, the CBA's services, the application components, and the CBA's component manager, or connector,

* Corresponding author. Tel.: +1-913-897-8515; fax: +1-913-897-8490.
E-mail address: saiedian@eecs.ukans.edu (H. Saiedian).

0950-5849/02/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0950-5849(02)00020-4

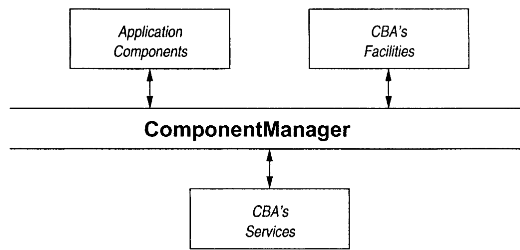

Fig. 1. The overall system model.

- the component—an isolated element, a member of an external distributed composite system, and it interacts with that system through a standard set of interfaces,
- the connectors—facilities provided to components to enable asynchronous, dynamic, anonymous component management and communication; an interface that provides proper connectivity between components and ensures the communication between components.

We apply the proposed framework to evaluate three primary CBAs, namely:

- SONI's (Sun + Oracle + Netscape + IBM) Enterprise Java Beans (EJB),
- Microsoft's Distributed interNetwork Architecture (DNA), and
- Object Management Group's (OMG) Object Management Architecture (OMA).

This paper will provide the classification necessary to determine how well CBAs impact reuse. By applying this classification, we will determine how well a CBA facilitates reuse, and determine which CBA (OMA, EJB, or DNA) is ideal for reuse in a certain organization or for a particular application.

2. Component-based software development (CBSD)

The complexity of the software industry substantially increased since the nineteen-fifties, and the software being developed has grown very large and complex making it difficult to develop software with reuse in mind. CBSD allows for code to be used and reused across multiple platforms, in various languages and in different locations across networks. Such development approach offers a consistent infrastructure but requires a standard interface for components. The infrastructures for CBSD need three main elements: uniform design notation, standard interface, and repositories. The uniform design notation ensures consistent architectural diagram to describe a component's functions and properties. This is critical to designing collaboration between components and ease communication between developers.

A standard interface to components would allow any application in any language on any platform access its features. This would be achieved by an application by binding to the component model or interface definition language (IDL). Some interfaces now in existence include Microsoft's Distributed Component Object Model (DCOM) [6] and Sun Microsystems' JavaBeans. The OMA is an extensive component standard that allows multiple specifications of various aspects of the technology, including component interfaces. This will allow IDLs and DCOMs to be interoperable. Various standard interfaces offer CBSD as a more reasonable strategy in tackling the increasingly complex task of software development.

2.1. Component-based architecture

The framework for CBAs chosen consists of three major parts: overall system model, components, and connectors. The framework used includes these parts to provide for the classification of CBAs. The framework provides a high level of abstraction to ease the understanding of complex distributed CBAs.

The framework of the current description of CBAs must be expressed at a high level of abstraction; enough to provide a distilled overview of complex distributed component-based systems currently in existence.

Our framework needs to be updated as technology advances at an intense rate. A high level of abstraction enables the illustrations to be updated while enabling the details to remain the same. In addition, as the details of the pieces change, the main illustration may not need to change.

This work uses three well known parts of software architecture necessary to adequately provide a high level of abstraction of a CBA. The common framework used to view the CBAs includes the overall system model, the component, and the connectors. Many researchers have used similar parts in the illustration of various software architectures (e.g. see Refs. [1,5,8] to name a few). The following three sub-sections describe how each of these parts are used in this work.

2.2. System model

The system model used should provide a high level of understandability to enable a reader to quickly gain insight into each of the CBAs. It portrays a high level of abstraction to provide an overall view of the CBAs. It must be graphically depicted, explicitly defined, and semantically correct. The model provides a high level of understandability.

The overall system model provides a high level of abstraction like the one used later in the portrayal of EJB, DNA, and OMA; however, certain pieces of the CBAs are investigated further in the comparison section based on specific criteria. Many complex details of the CBAs will remain hidden in the components and connectors, since we are concerned primarily with reuse. However, we scrutinize certain pieces to flush out the technical detail needed to determine which CBAs impact reuse better. Other researchers can use the same models to analyze the CBAs

even further, scrutinizing only the pieces that impact their research.

The illustration in Fig. 1 provides a look at the overall system model. The illustration includes the CBA's facilities, the CBA's services, the application components, and the CBA's component manager, or connector. A CBA provides component facilities and component services to increase the simplicity of a component-based heterogeneous distributed environment. The application components concern specialized functionality and may be a wrapped legacy or newly developed components. The component manger is depicted as the connector for the CBA. The connector provides essential ingredients to enable communication between components.

CBAs usually offer components of their own such as facilities and services. Services include common functionality used among many applications, such as a naming service. Facilities include application level components for horizontal (across several domains) and vertical (within one domain) reuse.

2.3. Components

A component is an isolated element and a member of an external distributed composite system. A component interacts with that system through a standard through a standard set of interfaces. This definition closely resembles the one formed at the Workshop on Component-Oriented Programming 1996 European Conference on Object-Oriented Programming (ECOOP) [9]. The major difference is the requirement of explicit context dependencies.

Explicit context dependencies define the needs of components, or what assumptions the components make of the system environment. Szyperski [9] claims maximizing reuse minimizes use due to the 'explosion of context dependencies'. The degree of evolution (high turnover) of components and their surrounding worlds (computing and networking platforms) is likely to continue, and with each additional context dependency, the ability to reuse that component declines.

For example, a dynamic link library (DLL) in the Microsoft environment contains a group of functions providing imaging services. The DLL originally supported a 16-bit operating system; however, recent enhancements upgraded the client application to a 32-bit implementation. Unfortunately, the 32-bit application to 16-bit DLL connection was not seamless. This additional requirement should have been explicitly specified. Both the internals of the application and the DLL remained the same. The 16-bit DLL outlived its usefulness and needs to be retired. A component should not outlive its usefulness; and therefore the context dependencies should not explode.

Szyperski [9] offers insight into the relation of leanness, or limited context dependencies, and robustness, or limited fat, and concludes the ideal component should not be too large or too small. Reuse studies also have explained the relation to size of a component and its reuse ability (reuse potential for small components versus reuse benefit for large components). The ideal size of a component needs to be of medium scale.

A component needs to be isolated from the external composite system, and it must follow strict standards of interface to facilitate construction of an application. It must provide a standard interface, or set of interfaces, to communicate with the external system. Some components may require other components to exist in order to complete their task. Some CBAs allow components to list other components needed to surface any assumptions. As noted by Garlan et al. [1], it is not a good idea to assume that certain components exist.

A component's interface provides communication to the external world, and enables encapsulation so the component may hide its inner details. A consistent interface enables a component to work well with others. For example, IDL provide a component with a consistent interface so that other components may understand how to communicate with it without knowing the internal workings. A standard interface eases the integration of components into a system, and simplified maintenance when a new version of the component is installed into the system. If an interface changes, then every component that communicates with it would need to change.

A stateless component means the component exists only during its usefulness, and the component's existence in a system must be distinct and not allow for multiple copies of the identical components. A persistent state component means the component persists after creation and use, not expiring when its task complete.

2.4. Connectors

Connector interfaces concern facilities provided to components to enable asynchronous, dynamic, and anonymous communication. A connector's interface provides proper connectivity between components and ensures the communication between components. Connectors facilitate the inter-operability, legacy reuse, and re-engineering of components in a CBA.

Connection-oriented programming focuses on dynamic connection of components rather than statically chained function calls. The connectors are factored out as separate services such as an event or messaging services. The independence of connectors provides the needed connection to facilitate component reuse in CBAs.

Two fundamental ingredients for connectors include naming and locating services for components. Naming services concern the how components are named. The names are for use by the connectors and have no meaning for component developers. The names entail unique identifiers normally global in nature, and possibly of groups of components. All CBAs should provide a registry or repository of the names to help locate components.

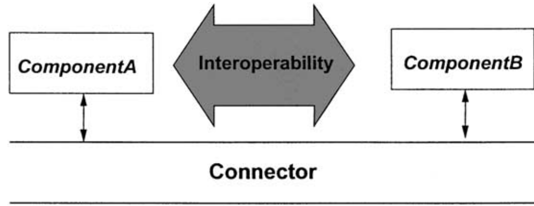

Fig. 2. Interoperability.

3. Reusability criteria for component-based architectures

Two essential characteristics of CBAs that impact software reuse include interoperability and integration. The ability for interoperability and integration of components across a heterogeneous platform provide for maximum reuse of moderately sized components. As the following figures depict, interoperability concerns communications across a CBA (for example, between component A and component B in Fig. 2). Integration concerns deployment in CBAs (for example, component C in Fig. 3 during either installing or uninstalling or plug/pull and play). Interoperability concerns the ability for a component to operate with other components:

(1) existing on multiple platforms
(2) written in various languages
(3) without the need for modification

Integration concerns the ability for a component to combine with other components:

(1) in a dynamic manner (during run time)
(2) in a seamless way without risks
(3) without concern for different versions

Table 1 illustrates important factors that affect interoperability; and integration. These factors are common knowledge and are widely referenced in the literature.

The elements of CBAs that impact software reusability through interoperability and integration include components, connectors, packaging and patterns. A component of a CBA can be defined as follows:

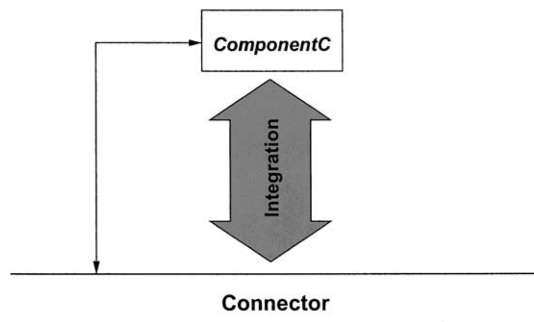

Fig. 3. Integration.

Table 1
Factors that affect interoperability and integration

Interoperability and integration factors	
Interoperability factors	Integration factors
Interface	Portability
Location transparency	Adaptability
Synchronization	Understandability
Threads	Confidence
External dependencies	Extendibility
Performance	Scalability
Availability	Environment constraints

(1) an independent unit of composition
(2) a member of a distributed composite system
(3) a contract with a standard and consistent set of interfaces

A connector is normally defined as controlled and or dynamic connection between components. A package can be defined as the method of control or containment that a component uses. Finally, a pattern can be defined as a non-specific problem solution when certain conditions are met.

Certain of a CBA's components, connectors, packages, and patterns that can assist software reusability reuse are shown in Table 2. All or perhaps most of these factors are also common knowledge and widely accepted factors.

4. The impact of popular CBAs on software reusability

This section provides a comparison of three popular CBAs, DNA, EJB, and OMA, by scrutinizing one property of a CBA, reuse, in a detailed and technical manner.

Most criteria for components discussed earlier impacted both integration and interoperability. The discussion described that all criteria impact the integration, though. Integration is critical for the success of a CBA and may be the driving force behind the increasing popularity of CBAs. The following further analyses CBAs based on technical details concerning components.

All of the architectures provide a facility for component use/reuse. The facilities include repositories and registries. All three CBAs provide a registry or repository of the names to help locate components. The DNA uses the existing Microsoft Repository to facilitate reuse. This product provides for the automated cataloging of components. OMG's ORB implementation repository provides the registration of objects with on ORB. The Java RMI registry enables component assembly in EJB.

COM object reuse entails containment and aggregations. Containment simply means one object contains another object (through reference). Aggregation in COM means the contained object's reference may be passed around to clients of the container.

Table 2
CBA features that assist software reusability

CBA feature for			
Component	Connector	Package	Pattern
Facilities for locating reusable components	Transparency	Containment	Heterogeneity
Contract through interface	Distribution	Vertical communication	Catalogs
Versioning	Communication support	Crosswise communication	Control constraints
Modifiability	Connector dependencies	Multi-protocol environment	Explicit structure
Efficiency	Component management	Pre- and post- conditions	

OMA and EJB object reuse involves inheritance and aggregations and additionally offers multiple inheritance and composition. Inheritance allows a new object to be derived from existing classes through object by reference. Aggregation and composition in OMA and EJB means a more specialized form of an association.

All three CBAs provide forms of late binding, encapsulation, and dynamic polymorphism (or sub-typing). Subtyping in Java and OMA provide the ability for a component to have a new interface through polymorphism. Java enables independent extension [9] to enable extended components to be combined dynamically like within a browser supporting Java.

Multiple interfaces to a component are enabled through inheritance in Java and OMA and simultaneous in COM. Java and OMA supports extension of the interface via inheritance. COM provides immutable interface and a new one must be used for every instance of a component. All three provide substitution with contracts. A COM component can have multiple interfaces. The interfaces include a set of methods and properties that other components may invoke query, or set. These interfaces are exposed to other components.

The COM approach to versioning is to treat each interface as published and unchangeable. Each version is considered a different component. This is an effective way to make the versioning sound [9]; however, every time the version of a component changes, other components must also change in order to use the new component. This hinders the ability to dynamically introduce a component into the system creating maintenance and domain reuse problems. For example, if a new version of a grammar checker is released, then all components that interface with it need to know to invoke the new components. IBM's SOM, which will possibly become a member of OMG's CORBA facilities, enables a release order for a component allowing a component to evolve rather than being replaced. Java's leaves versioning at the binary level compatibility. Java enables a new version of a component to integrate into a CBA without affecting a client; however, we must ensure the greatest independence of components or a simple global variable may cause a malfunction.

OMA allows for components to be varied in granularity, implemented by various groups, in different programming languages, and across different operating systems. COM components are not bound to particular programming languages and support several operating systems to include Unix flavors like Solaris. Java provides robust APIs to facilitate reuse with the Java language and across various operating systems.

DCOM is not available on platforms other than Microsoft's 95/98/NT. This obviously impacts the reusability of components across heterogeneous platforms in a negative manner. EJB successfully enables the reuse of components across many platforms; however, it is more difficult when we choose to implement some components in languages other than Java. CORBA is a standard that provides true integration and interoperability of components written in various languages for use on multiple heterogeneous platforms.

Notice the word 'glue' has not been used as part of the definition for a connector in CBAs. The purpose for the avoidance of this word is that 'glue' would not be a good description for connectors in CBAs. The reason for this is that CBAs operate with a high integration capacity. In other words, CBAs exist as a plug/pull and play environment. A component can be plugged into a CBA, and it may be pulled out of a CBA, preferably without negatively impacting a CBA. 'Glue' would symbolize a component being plugged (and cemented) into place, but never pulled.

4.1. Distribution of components

The distribution of components in DNA, EJB and OMA differ. For example, DNA's DCOM is based on distributed behavior, EJB's RMI and OMA's CORBA is based on distribute objects. Distributed behavior means the object resides on the server while processing a service. Distributed object means the object is distributed to the client for processing of a service. An object that is distributed across a network will be reused more often than a object residing on one server.

4.2. Location transparency

EJB provides for automatic and transparent location of components. Dynamic binding is provided through introspection and dynamic discovery through reflection. DCOM also provides a means for transparent location and

binding of objects. CORBA uses dynamic binding through method invocations and dynamic discovery via various degrees of late binding.

4.3. Connection and communication

Marshalling involves the use of stubs to transparently send information to and receive information from remote objects. OMG's IDL enables CORBA to provide generic marshalling and unmarshalling mechanisms. CORBA requires languages used to have a binding to the OMG IDL. The bindings allow communication between various languages and the OMG IDL. RMI also implements stubs and skeletons as a standard mechanism for communicating with remote objects. Microsoft uses MIDL to facilitate marshaling.

Connection pooling affects the scalability in various ways. DCOM uses connection pooling to increase scalability and protect operating system resources. DCOM scalability is limited to the size of a connection pool. EJB uses instance pooling, which increases the accessibility and performance of components and increases the scalability. ORBs use various methods of connection pooling to include thread pools [7].

4.4. Packaging

CORBA components, Java Beans, and COM components all provide interface standards. CORBA interface standards are well defined and provide language bindings to facilitate source code compatibility across ORB connectors. While CORBA provides an excellent standard interface, certain ORB specific objects used on the server reduces the portability of CORBA-based servers.

4.5. Component transfer format

All CBAs allow some form of a component transfer format. JavaBeans provides JAR files. COM relies on structured storage. OpenDoc's Bento is another format. The JAR files may contain multiple beans and include entries for the following: class files, bean prototype instances, help files, localization information, icons, and resource files needed. Standard packaging for runtime components includes MS ActiveX, JavaBeans and Enterprise JavaBeans. Standard packaging for component environments eases the deployment and management of CBAs to provide easy component integration. Containers have certain properties that may be set to alter the looks and behavior of a component. Microsoft's ActiveX containers define numerous interfaces to be used to handle properties. Java's beans reside inside of a container. A Java container can provide additional services like security, transaction behavior, concurrency, and persistence. A container's properties may be examined and altered. One negative remark on EJB is the freedom they give vendors in creation of containers. The vendors will always compete, and introduce proprietary lingo into products so customers are locked into their implementation. To ensure the portability of beans, proprietary features need to be avoided.

OMA provides several common functions and properties in its base object inherited by all OMA IDLs. This allows the basis services of CORBA to change without affecting the compiled IDLs that are in place. This decreases the risk of breaking a component that relies on these basic services if the CORBA services are changed. DCOM handles the basic services during runtime, keeping them separate from MIDL. This could cause all components' interface to break when the DCOM basic services change, and if history repeats itself, this is guaranteed.

4.6. Legacy applications

All three provide facilities to wrap legacy applications for use. Microsoft released the COM Transaction Integrator (TI), which extends CICS and IMS transaction programs with COM components. TI consists of a set of development tools and runtime services that automatically 'wrap' IBM mainframe transaction and business logic as COM components that run in a Windows DNA environment. Other CBAs use wrappers in the form of IDLs to enable legacy applications to integrate with modern technology. OMA provides the standard OMG IDL interface for a seamless integration of legacy applications.

5. Overall comparisons

One difference between the three CBAs stems from the foundation from which they are built. COM differs from OMA and EJB in the manner for which it was developed. COM, like some other Microsoft technology, followed a different method of development than the other two. First, COM technology is implemented and put into operation, and then anywhere standards are needed they are created. OMA and EJB both create standards first, with OMA particularly strong in this area, and then create the technology.

5.1. Interface definition language

All three CBAs define a much-needed consistent interface (IDLs) for components. The major differences are the aspects of each of the technologies they respectively represent. EJB, for example, is based on programming technology and therefore the ability for JavaBeans written in Java to connect to components of another language additional steps must be taken. Therefore, the language requirement pressures a developer to implement components in Java to avoid the complexities of additional interface. DNA is based on COM and focuses on operating system technology. COM is reliant on technology involved with Microsoft and therefore increases the complexity when crossing over to other operating systems; though, COM implementations do exist on some Unix flavors like Solaris. OMA is set to succeed in

this arena due to its unbiased representation. OMA provides mostly specifications created from a large pool of professional organizations and large corporations, and allow independent vendors to implement the specifications creating constructive competition.

The IDLs of each also differ in how they are implemented. The EJB IDL is based on the OMA IDL. The main difference between OMA IDL and DNA MIDL aside from specifications is that OMA IDL offers multiple inheritance of interfaces. MIDL implements multiple interfaces to a component, based on each method may have its own interface. Multiple inheritance enables OMA components to support the specialization, or refinement of classes into subclasses, and composition. This enables composition and increases reuse. Additionally, CORBA interfaces inherit the base class CORBA object which performs the common tasks, like object registration, object reference generation, skeleton instantiation, etc., after construction. DCOM handles these tasks by server programs or dynamically by DCOM run-time system. Additionally unlike CORBA IDL, MIDL defines no common set of data types. Anyone can define interfaces and data types in MIDL accessible from C++, but not from Visual Basic or Java. Finally, the OMA IDL is based on a distributed object environment; the DNA IDL is based primarily on DCE's RPC IDL, created for the 'C' procedural environment. Moreover, the MIDL implementation is not even DCE complaint, further hindering the distribution, reuse, and ultimately interoperability of components. The OMA IDL is ingrained into the architecture, whereas the DCOM architecture does not put as much importance on the IDL and does not even require one.

Inheritance facilities the creation of a hierarchy of related interfaces that share code and create a common interface. A common and consistent interface is critical to the ability to reuse components. Multiple inheritance may be used to design a framework to capture conceptual elements of a component to be used for system building and reuse. Also, an IDL that reuses inherited common tasks and data types greatly reduces the repetitious and extra amount of work required implementing component IDLs (See Appendix A). Since DCOM does not support multiple inheritance of interfaces, DCOM objects cannot easily support multiple interfaces. DCOM provides inheritance through containment and aggregation limiting the level of abstraction for components. With all this in mind, its safe to say the CORBA IDL is by far the better IDL.

5.2. Connectors

The three connectors of DNA, EJB, and OMA are DCOM, RMI, and ORB, respectively. Distributed components require a distributed computing and therefore the connector which handles the distributed computing best will likely succeed in most benchmark tests. The worst of the three for reusability is DCOM due to its lack of integration on multiple, heterogeneous platforms, like Unix flavors. RMI on the other hand enables the integration of JavaBeans on various heterogeneous platforms. However, the ability for interoperability among components implemented in another programming language is impeded by the level of difficulty in interfacing between a non-Java component and an EJB component. OMA is set to achieve the best interoperability and integration using ORBs that are not platform dependent and are programming language neutral.

OMA's CORBA is the creation of many companies attempting to standardize distributed object computing. CORBA manages all aspects of component interoperability ensuring a distributed system that reuses components from multiple sources. CORBA enables the discovery of components on the network greatly increasing the chance of component reuse. Many details of a component in CORBA remain hidden to clients, like operating system and programming language. The value of information hiding for a component's is recognized as extremely important to reuse. Only the well-specified interface is of importance to a client.

5.3. Locating components

Two services enable a CORBA component to locate another component. One is the Naming Service, which allows a component to be found by registered name. The second service is the Trade Service that allows a component to be found by registered service characteristics. The second method would have a higher volume of reuse since a client would not be bound to one name. If we think of it like the telephone book, a person may always call the same barber from the white pages if the name is known. However, if we do not know the barber's name, we can use the yellow pages to look up barber services, and just choose any one.

DCOM does not provide a naming or trading service at run time because of its component's lack of a unique object identification. Additionally, if a component cannot be connected to a given server without a method to identify the server, the component may only request a service from an arbitrary server supporting the requested services.

A component is not bound by an ORB to a client or server role. The component can receive requests and service the requests, and it can request a service from another component. An OMA component can be a client and a server.

OMA's IIOP provides a common communication backbone needed to allow different ORBs to interoperate. This is crucial for reuse of components and patterns when they may be implemented for various ORBs.

5.4. Packaging

The packaging in OMA, EJB, and DNA each has tradeoffs that make each other differ. OMG provides a solid IDL. EJB's JAR file leads in component transfer formats. DNA leads the market in control components. EJB containers may negatively force the use of only one vendor. DNA offers a

promising solution to legacy applications with its TI. DNA seems to offer the best options, and EJB coming in at second and OMA finishing a strong third due to its rigorous standards.

Although DNA holds the desktop arena for facilities within the Microsoft NT operating system, the OMA provides both vertical and horizontal facilities on the server end located on various boxes or hardware/operating system combinations. OMA therefore exceeds in the distributed arena. In addition, if Microsoft DNA is chosen, then Microsoft supported tools such as Microsoft Visual Basic, C++, Java must also be chosen. EJB continues to provide increased development facilities, but due to its programming language constraint, it is difficult to reuse domain components created in other languages or possibly for say CORBA. Because of the heterogeneous environment allowed by OMA and its non-vendor specific cause, it is set to succeed. In addition, CORBA is based on the object model and well-specified, which specification is well-documented to be a design step in successful development of computer systems. However, we should never underestimate the powerful Microsoft Corporation, which has steamed ahead full-power into the component market with its ActiveX controls and simple Visual Basic programming tool/language. Microsoft has a strong installed foundation into the corporate world and has a large pool of resources.

6. Discussion

CBAs provide the facility for reuse of components. Characteristics of components, connectors, packaging, and patterns help determine how well a CBA impacts reuse. A highly inter-operable CBA significantly increases the potential for reuse. Additionally, the ability to integrate components and connectors critically improves the facility necessary with reuse. This paper provides a framework that can be used as a basis for evaluating different CBAs on the basis of the three major parts: overall system model, components, and connectors. Related work continues on software architectures. Software Engineering Institute (SEI) with many writings from various SEI representatives, including Mary Shaw and David Garlan's book [8]. Related work on evaluation of different CBAs also exists, and some work focus on certain properties of e.g. CBA [7] and a number of articles in the Proceedings of the Software Architecture and Component Architecture Workshop (1999). The existing works in software architecture evaluation and component software evaluation does not a consistent and formal way to distill CBA. Additionally, there has not been a detailed and technical evaluation of how well CBAs impact reuse through both integration and interoperability using four key elements of a CBA: components, connectors, patterns, and packaging. A need exists for a consistent framework to effectively distill a CBA to a level that is understandable and actual evaluations and comparison of existing CBAs are invaluable. Also, the comparison of CBAs based on how well they impact reuse is crucial due to reuse being one of the driving forces behind component-based software world. This paper offers a consistent framework for distilling CBAs and the application of the framework on three popular CBAs to provide overall evaluations and a comparison. The work offered after the distillation of the three CBAs provides a more detailed and technical comparison of the three CBAs based on how well they impact reuse. This framework expedites the understanding of complex distributed CBAs and provides a needed architectural foundation to base further scrutinization. The common framework used to view the CBAs consists of the overall system model, the component, and the connectors.

This paper provides a comparison of three popular CBAs, DNA, EJB, and OMA, by scrutinizing one property of a CBA, reuse, in a detailed and technical manner. Two essential aspects of reuse in CBAs covered include interoperability and integration. Though, DNA, EJB, and OMA at a high abstract level appear equivalent, a look at a more detailed and technical level distinguishes important properties that impact reuse. By evaluating the properties of each CBA, we provide decision-makers a better look at the different options at a high level and a technical level. The evaluation provides a tool to enable an organization to select the most appropriate CBA for their organization or for a particular application.

7. Future directions

Research could be conducted on other properties of the CBAs like threading, performance, reliability, and security. For example, how CBAs impact multi-threaded components. Schmidt [7] provides an article comparing architectures for multithreaded Object Request Brokers (ORBs). A comparison of how different CBAs, like EJB and DNA, facilitate integration and interoperability in a multi-threaded environment would provide us with additional pertinent information for deciding which CBA is appropriate in threaded environments.

Further research into CBAs and reuse could be conducted to determine the level of reuse [3] a CBA would facilitate. For example, the vertical CORBA facilities assist in the development of a domain-specific reuse organization. What other facilities exist in CBAs and how do they benefit the development of a domain-specific reuse organization?

Extremely important in the analysis of any software technology is the implementation in a lab setting. In order to provide the best comparison of the CBAs, it would be necessary to set up a CBA lab. We could then choose one general domain application and implement it on each of the CBAs, possibly even across the CBAs. We could then analyze not only the ability for reuse, but also the feasibility of the

product created. Critical issues such as security, efficiency, and dependability of components could be flushed out.

One area that needs thorough examination is the existing CBA middleware technology. Middleware will most likely play a huge role in CBAs. For instance, CBA bridges seek to provide improved interoperability between CBAs, like CORBA and COM. It has been suggested to use COM on Windows 98/98/00 machines for a graphical user interface (GUI) front end and use CORBA to enable the integration of legacy applications on the back end servers like mainframe and Unix flavors. CBA bridges on Windows NT platforms combine CORBA and COM, for example, so CORBA objects appear as COM objects to front-end applications. The combination of the various CBAs may be necessary to integrate large, distributed, heterogeneous systems. Finally, the ultimate 'component system architecture' [2,9] should be invented. Software architecture of a large distributed application environment is crucial to its success, and the need for multiple views [4] versus the simple abstraction of components and connectors.

References

[1] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch or why it's hard to build systems out of existing parts, Proceedings of 17th International Conference on Software Engineering, April 1995.

[2] R.L. Glass, Reuse: what's wrong with this picture? IEEE Software 15 (2) (1998) 57–59.

[3] I. Jacobson, M. Griss, P. Jonsson, Software Reuse, Addison-Wesley, Reading, MA, 1997.

[4] P.B. Kruchten, The 4 + 1 view model of architecture, IEEE Software 12 (6) (1995) 42–50.

[5] D. Perry, A. Wolfe, Foundations for the study of software architecture, ACM Software Engineering Notes 17 (4) (1992).

[6] W. Rubin, M. Brain, Understanding DCOM, Prentice-Hall, Englewood Cliffs, NJ, 1999.

[7] D.C. Schmidt, Evaluating architectures for multithreaded object request brokers, Communications of the ACM 41 (10) (1998) 54–60.

[8] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, Englewood Cliffs, NJ, 1996.

[9] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, Reading, MA, 1998.