2001

EJBrew : enabling application composition for Enterprise JavaBeans

Ragavan Ramanathan
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Ramanathan, Ragavan, "EJBrew : enabling application composition for Enterprise JavaBeans" (2001).
Master's Theses. 2155.
DOI: <https://doi.org/10.31979/etd.v2fh-gpm3>
https://scholarworks.sjsu.edu/etd_theses/2155

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

***EJBrew* – Enabling Application
Composition for Enterprise JavaBeans**

A Thesis

Presented to

The Faculty of the College of Engineering

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Ragavan Ramanathan

May 2001

UMI Number: 1403988

Copyright 2001 by
Ramanathan, Ragavan

All rights reserved.

UMI[®]

UMI Microform 1403988

Copyright 2001 by Bell & Howell Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© 2001

Ragavan Ramanathan

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF ENGINEERING

Dr. Ali M. Zargar, Chairman

Professor Dan Harkey

Professor Dmitry Dimov

APPROVED FOR THE UNIVERSITY

ABSTRACT

***EJBrew –* Enabling Application Composition for Enterprise JavaBeans**

By Ragavan Ramanathan

Since the introduction of Sun Microsystems' Java-based server-side components called Enterprise JavaBeans (EJBs) in 1998, server-side application development is becoming increasingly simpler as various EJB based components are assembled. With an increased demand and rapidly growing market for EJB-based components, many vendors are introducing specialized EJB components to handle various types of business functions.

With larger numbers of EJB components, the complexity of their effective use is also increasing. Application assemblers wishing to use a mix of vendor-delivered and in-house developed EJBs need effective methods and tools to perform their tasks quickly and easily. With the growth of Internet-based web applications, time-to-market is critical. Corporations must put together a web-front in just weeks.

Current methodologies available to connect or assemble EJBs have limited capabilities and some shortcomings. This thesis focuses on investigating currently available methodologies, comparing their advantages and disadvantages, researching new methodologies, and recommending a specific methodology.

Acknowledgment

I would like to thank my advisor, Professor Dan Harkey, for his continuous guidance, encouragement, and support throughout this thesis work, and for being supportive throughout my graduate work.

I would like to thank Dr. Ali Zargar and Professor Dmitry Dimov for serving on my committee. I would also like to thank Professor Vicki Parrish for reviewing my thesis.

Last, but not least, I would like to thank my wife Inthi and baby Amrita for being patient, and for providing constant support during the years of my graduate work.

Table of Contents

Chapter 1.....	1
Introduction.....	1
Problem Statement.....	3
Organization of Thesis.....	4
Chapter 2.....	5
Component-Based Software Development.....	5
Introduction.....	5
What is a Component?.....	7
Why Components?.....	7
Business Requirements of Components.....	10
Component-Based Software Development Process.....	11
Component Models and Frameworks.....	13
Component Model.....	14
Component Framework.....	14
Chapter 3.....	16
Component Models and Frameworks.....	16
JavaBeans.....	18
Component Object Model.....	19
Enterprise JavaBeans.....	21
CORBA Component Model.....	23
Comparison Table.....	25
Chapter 4.....	26
Enterprise JavaBeans.....	26
Introduction.....	26
Overview.....	27
EJB Taxonomy.....	28
EJB Architecture.....	30
Session Beans.....	33
Entity Beans.....	34
Message-Driven Beans.....	36
EJB Development Process and Roles.....	36
Chapter 5.....	42
CORBA Component Model.....	42
Introduction.....	42
Overview.....	44

Abstract Model.....	47
Interfaces.....	49
Events.....	51
Navigation.....	52
Attributes.....	52
Application Servers.....	53
CCM Containers	53
Container Framework	54
CCM Clients	55
CCM Components	56
Persistence.....	58
Interfaces Between the Application Server and CCM Containers.....	58
Component Configuration	59
Component Declaration and Implementation	61
Component Packaging	62
Component Descriptor.....	62
Component Assembly.....	63
Component Deployment and Installation	64
CCM and EJB	64
Chapter 6.....	68
Requirement: Enabling Application Composition for EJBs.....	68
J2EE Application Architecture	68
Software Development Using IDEs.....	69
EJB Application Composition Using IDEs.....	71
Building New or Identifying Available Components	72
Selecting Necessary Components	73
Instantiating Components	73
Configuring Component Instances	74
Connecting Component Instances.....	74
Specifying Connection Parameters	76
Application Code Generation	76
Creating Deployment Packages.....	78
Development-Time and Runtime Requirements	78
Summary of the Requirements for EJB Application Composition.....	79
Chapter 7.....	81
The Solution: <i>EJBrew</i>	81
Introduction.....	81
Solution.....	82
Specification of CORBA Componentized EJB	84
The CcmEjb.Home Interface	87
The CcmEjb.Bean Interface and Sub Interfaces	87
The CcmEjb.Remote Interface.....	89
UML Profile for EJB and CcmEjb UML.....	92

UML profile for EJB.....	92
CcmEjb UML.....	94
CcmEjb Interface Implementation.....	95
CcmEjb Naming Patterns.....	96
Definition of a CcmEjb-Composed Application on Package	97
Chapter 8.....	98
Future Work	98
Introduction.....	98
Currency of the Standards and IDE Tools	98
CIDL-Based CcmEjb Generation	99
Integration with J2EE IDEs	99
Meeting Run-time Component Integration/Configuration Requirements	99
Chapter 9.....	101
Conclusion	101
References	104

List of Figures

Figure 1. EJB Taxonomy.....	28
Figure 2. EJB Architecture	31
Figure 3. EJB Component. Contents of an EJB JAR File.	32
Figure 4. CCM Abstract Model	47
Figure 5. Sample CORBA Component Definition in CIDL.....	51
Figure 6. CCM Container Framework	55
Figure 7. Interoperation in a Mixed Environment	66
Figure 8. J2EE Application Architecture	69
Figure 9. EJB and IDE Interaction.....	75
Figure 10. Proposed EJBrew Solution.	84

List of Tables

Table 1. Comparison of Component Models.....	25
Table 2. Features of Java IDEs	71

CHAPTER 1

Introduction

Component software is a term used to describe a programming technique in which software is written in a compartmentalized form to encourage its reuse. Instead of a single main program, the application is broken down into components that operate together. Each component exists as a "sub-program" that is designed to provide a specific set of functionality. Well-known software components are found in Graphical User Interface (GUI) applications. Examples of GUI software components include a spreadsheet grid, a data-graphing window, and a rotating marquee display. Over the last few years, the use of software components on the back-end, or server-side is becoming common. Some examples of server-side software components are an e-commerce shopping cart, credit-card authorization, catalog, and customer. These components are not stand-alone applications; they are "pieces" of applications that can be tied together through a controlling program.

Component architectures are among the latest developments in the effort to engineer portable, reusable software. These architectures typically consist of application programming interfaces (APIs) and utilities that facilitate the creation of reusable objects and the composition of these objects into larger software systems. Conventionally, the

goal of component architectures has been the efficient creation of custom end-user software by assembling and configuring generic, “off-the-shelf” components.

The most commonly used component architectures are based on CORBA, Java, and COM. Sun Microsystems introduced two Java programming language based component architectures: JavaBeans for the client-side component programming, and Enterprise JavaBeans (EJB) for server-side programming. The JavaBeans component model is heavily used in Graphical User Interface (GUI) client applications. EJBs are becoming more popular in enterprise-level server applications and web application servers. Traditionally, client and server applications were programmed using some function libraries. Function libraries provided a form of code reuse. Effective code reuse is one of the most critical issues in software engineering.

With the introduction of component architectures and object-oriented programming languages such as C++ and Java, it is not only possible to just reuse functions, but also it is possible to reuse software components that are modeled after business functions and entities. The new software engineering paradigm is shifting away from programming, testing, and debugging individual modules to buying common-off-the-shelf (COTS) software components that are developed, tested, and debugged by vendors who are specialized in a specific business area.

The explosive growth of Internet based e-commerce and Java programming language's success in web applications has made EJB components very successful. According to *PricewaterhouseCooper* (Patterson, 2000), the market for off-the-shelf software components will be worth \$1 billion by 2002, and the related services will be worth \$3.2 billion (Patterson, 2000). Research conducted by Sun suggests that 77% of developers using Java technology want access to off-the-shelf EJB components (Heiss, 2000). In the growing market for server-side components EJB will be the primary player.

Problem Statement

The EJB 1.0 specification was released to the public on March 28, 1998 (Sun Micro Systems, n.d.). About forty companies have announced support for EJB based products or application servers. Integrated development environment (IDE) is an advanced form of programming tool that is generally available to programmers, but there are very few IDEs that support EJB development. Even the IDEs that do support EJB development do not support EJB application composition that would involve connecting or wiring one or more EJBs to form an application unit that can be deployed in a server.

This thesis analyzes currently available component models and chooses a suitable model for server-side application composition. The proposal is to utilize the *Extended CORBA Component* definition and apply it to EJB. The choice is based on the fact that EJB has become more widely used, and CORBA has more complete definition of a

server-side component suitable for application composition. This thesis is not about developing EJB classes; there are many capable tools on the market already. It is about using existing EJB classes and deployments and creating new applications by wiring different EJBs and associated presentation classes. The concept and the associated process and methodology for enabling application composition for EJBs are called *EJBrew*.

Organization of Thesis

This thesis is organized into several chapters to provide background information on component technologies and to document suggested ideas, methodology, and a proposed solution.

Chapter 2 provides a general overview of software components and component-based software development (CBSD). Chapter 3 reviews some of the most commonly known component standards such as JavaBeans, COM, CORBA, and EJB. Chapter 4 reviews the EJB standard and discusses EJB based software development. Detailed background information on the CORBA Component Model is provided in Chapter 5.

Chapter 6 documents the requirements identified for enabling EJB for application composition. The proposed solution to meet the requirements is documented in Chapter 7. Suggestions on improving the proposed solution and recommended related work can be found in Chapter 8. Finally, Chapter 9 contains the conclusion.

CHAPTER 2

Component-Based Software Development

Introduction

Software development projects have a very high-rate of failure. Failures are caused by not meeting the requirements, exceeding the allotted budget, not meeting project deadlines, and too many defects being found. According to a survey conducted by KPMG (Klynveld Peat Marwick Goerdeler) in 1995 in the U.S.A. 31% of Information Technology (IT) projects were cancelled before completion (Project Management Resources, n.d.). *The Wall Street Journal*. "42% of corporate information-technology projects were abandoned before completion....U.S. companies spend about \$250 billion annually on computer technology" (Kapur, n.d.).

Compared to other industries such as manufacturing and civil construction, software development is relatively young. Even though the use of computers began little more than fifty years ago, widespread use of computers began in the early eighties with the introduction of the IBM and Apple personal computers. Along with the growth of computer use, many computer programming languages, software development methodologies, and development tools also evolved. In order to ease the process of software development, many generations of programming languages and different development processes were introduced.

The “PC revolution”, followed by the explosion of the Internet, and the desire of established companies to not be left behind in the burgeoning world of E-commerce have placed a premium on any technology that can improve the time-to-market for software. One of the solutions for rapid software development and delivery is reusing existing “software parts”. The software parts that are reusable are generally called components. These components are either developed internally or externally.

Component technologies are to programming what manufacturing parts are to mechanical and electronic engineering. The initial design is very costly and time consuming to produce, but once it has been tested successfully, it can be mass-produced for a fraction of its original cost and then used in a variety of assemblies beyond its original design point. Components promise the same kind of leverage for software, making possible a second industrial revolution in the 21st century. Components have become the de facto standard for GUI development. The next step is to make them the standard for server development. The payoff, by comparison, is immense.

Following the parts and assembly line based manufacturing concepts, component-based software development or engineering (CBSD/CBSE) was introduced to the information technology (IT) industry. It is an emerging development paradigm that promises to accelerate software development and to reduce costs by assembling systems from prefabricated software components. Designing, developing and maintaining

components for reuse is, however, a very complex process, which demands high requirements not only for component functionality and flexibility, but also for the development organization.

What is a Component?

In order to discuss component-based development, it is important to have a working definition. A software component is defined as (Cobb, 2000):

- A self-contained unit of software code, consisting of its own data and logic
- Independently developed, delivered and installed
- Having well-defined connections or interfaces exposed for communication (provided, and expected)
- Only allowing access to data and behavior through the interfaces it provides
- Designed for repeated use in developing applications
- Able to be used with or without customization
- Separating its interface from the implementation used to support it (plugability)

Why Components?

As described earlier, CBSE was introduced to improve the process of software development. The benefits of using CBSE are as follows:

Faster time to market: Through the use of off-the-shelf components, which are usually well documented, tested and proven, and developed by domain experts, a software development project can eliminate the time needed to design, develop, and test common business components.

Increased developer productivity: By reusing existing components, developers concentrate only on programming the pieces specific to their application domain. As components are already tested and proven, the programming bugs are reduced.

Reduced complexity: By focusing only on the pieces specific to application development, and by removing the need to develop components outside the developer's domain of expertise, the complexity of the overall application development is reduced.

Reuse of existing code: Reusing existing components from internal projects or from vendors.

Assembly like programming rather than development: When components are used in application development, the programming involves mostly assembling components together by connecting them using some "wiring", "gluing", or scripting methods.

Reduced skills requirements: Use of components eliminates the need for experts outside the domain of the application. With the shortage of skilled IT workers, this would be a tremendous benefit for any software development effort.

Consistent Look and Feel: Use of standard off-the-shelf, or in-house components provides a consistent look and feel for all the applications in an enterprise. This helps reduce time and cost for end-use training.

Easy to Upgrade, Extend, and Distribute: Component based software can be easily upgraded, extended, and distributed due to their modular design.

As with any solution, there are also pros and cons to CBSE. Some of the drawbacks of CBSE are:

Impact on System Performance: As component-based software will use modular design and a component framework, extra layers of communication are added to the software. This would normally reduce the performance of the overall system.

Cost: External components need to be purchased from vendors. Typically, along with the price of the components, there will be costs for

licensing, distribution, and support. Even though there is cost associated with the use of components, the overall cost would be less.

Dependency on Vendors: Use of external components from vendors means that an application developer has an external dependency on ongoing support, development, and guidance. There is a possibility that the vendor may not be around in a few years or may not support a particular component in the future.

The above list of benefits and drawbacks of CBSE shows the tremendous impact components can have on software development.

Business Requirements of Components

The previous discussion focused on the point-of-view of the application programmer who uses CBSE. The other side of the coin is the point-of-view of the component developer. A component developer must follow a public or a vendor-specific component model (defined and discussed later). Following is a list of some of the business requirements for a successful component model (Cobb, 2000):

Ease of Use: Must be able to be used easily for quick adoption by even non-professional programmers.

Availability of Tools: Easily accessible, low cost tools must be available to present and collect various components, and to assemble them into new applications.

Robust Application Production: The tools must produce applications that are robust enough to be deployed in mission-critical environments.

Support for Multiple Languages: As newer programming languages are introduced, older languages are not abandoned. The industry usually uses more than one language at the same time. This makes support for multiple languages essential.

Scripting Capabilities: Along with IDE-type development tools, it is important to provide scripting capabilities to make it possible for less-skilled workers to assemble applications. Scripting is essential for integration with other tools as well.

Component-Based Software Development Process

When components are used for the software development process, using CBSE, the development process is different from the classical process. The most common classical process is: gather requirements, perform analysis, design the system, implement, and test. The steps in the CBSE process would be: gather requirements, identify a suitable component model defined by the architecture, identify required components, find

matching components, develop any in-house components, compose application using components, and test. Tasks specific to component-based development are discussed below:

1. Component Qualification

An analysis must be performed to study interface characteristics of a candidate component. The component needs to meet the requirements that it has been selected for and to have interfaces compatible with the system architecture. If this is not the case, the component can either be adapted or discarded and an alternative component found.

2. Component Adaptation

This is the process of altering a component's interface to allow it to operate within a framework it was not originally designed for. Reliability of the modified component must be tested if the adaptation was non-trivial.

3. Component Composition

This is the process of selecting suitable component architecture and then organizing the selected components within the specified architecture. The methods of communication between components must also be defined.

4. Component Update

This task involves the revision or replacement of components to meet new requirements or to allow changes to other parts of the system. This task could be complicated if the system uses COTS components, as the source code for these components is either poorly documented or not made available. In such cases, the vendor of the components may need to become involved, at additional cost.

Component Models and Frameworks

In a component-based development environment there are two key concepts that are frequently misused due to confusion of terminology. They are *component model* and *component framework*. These two are related, but different concepts. In this section, these two concepts are defined and described.

After definition of the component model and component framework, a brief overview of some of the most common component models and frameworks are discussed in the next chapter. In later chapters, the Enterprise JavaBean Component Model and Framework, and CORBA Component Model are discussed more thoroughly.

Component Model

The standards and conventions imposed on developers of components are specified by a *component model*. Component models specify the design rules or patterns that must be obeyed by components. A component model defines the architecture of components and how they are manipulated and interacted with externally. The external interface and behavior definitions improve component composability by removing a variety of sources of interface mismatch. The design rules ensure that system-wide quality attributes are achieved and that components may be easily deployed into development and runtime environments. Another responsibility of a component model's architecture is determining how components are able to interact in a dynamic environment (D'Souza, & Wills, 1998).

Component Framework

Component models need services to manage and support their communications, resources, and lifecycle. These types of services are implemented by a *component framework*. Component framework uses its services to enforce a component model of the software. Based on the services provided by a component framework, it can be compared to an operating system for components. An operating system manages resources, and the lifecycle of a process, and a component framework does the same for components deployed within it. Some of the lifecycle operations are to start, suspend, resume, or

terminate component execution. Whereas general-purpose operating systems, such as Unix, support a rich array of interaction mechanisms, component frameworks are specialized to support only a limited range of component types and interactions among these types (D'Souza, & Wills, 1998).

CHAPTER 3

Component Models and Frameworks

In the current software market there are about four more commonly used component models and frameworks. They are Sun's JavaBeans and Enterprise JavaBeans (EJB), Microsoft's Component Object Model, and Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) Component Model (CCM). Although each of the four names suggests that they are component models, each specification also has an associated framework.

Sun's JavaBeans components are most useful for client-side software development. They are most commonly used for GUI-based visual component development. As the name suggests, JavaBeans is based on the Java programming language. Compared to the other models, the JavaBean specification is lightweight as it builds from the existing framework provided by the Java programming language. This component model is available on the platforms (operating system, machine architecture) where Java is available.

Sun's answer to the server-side component model is the Enterprise JavaBeans (EJB) specification. It is also based on the Java programming language. This specification is very elaborate as it defines many server-side framework services

necessary for the EJB component model. This component model is available on the platforms (operating system, machine architecture) where Java is available.

Microsoft's component model, COM, is very specific to their Windows operating system (OS). As Windows provides the component framework services, it is very difficult to have COM work on non-Windows operating systems like Unix. There has been some porting work performed by third-party software vendors, but this has had very little market impact. COM's other technical deficiencies have been overcome by its variations such as Distributed COM (DCOM), COM+, and Active X.

CCM is the component specification from the OMG (Object Management Group). The OMG is an open industry consortium. CCM is the component model that is based on the framework provided by CORBA. As CORBA and CCM are language-neutral specifications, they are available for the most commonly used Java and C++ programming languages and many platforms.

In this chapter four component models will be briefly described. In the following chapters EJB, and CCM will be discussed in more detail.

JavaBeans

Sun's JavaBeans component architecture is a set of APIs intended to support the creation of Java components that can be composed together into applications by end users. JavaBeans has taken the component software assembly paradigm to a new level. JavaBeans is an architecture and platform neutral API for creating and using dynamic Java components. The three most important features of the JavaBeans architecture are the ability to discover (and use) properties exposed by an object, events generated by an object, and methods exposed by an object. A Java class, called BeanInfo, which accompanies the JavaBean component class, supports these features. A property is a named attribute that may be queried and modified by invoking methods that conform to a particular naming pattern. Events are objects that are passed from one object to another to provide notification when something interesting has happened. An exposed method is simply a method that has been marked as visible to other objects. A builder tool, using the BeanInfo class, may invoke the exposed methods of an object in any way that the tool supports. Builder tools may, for example, allow invocation of an exposed method when an event is "fired" or generated by another object. To support builder tools, the JavaBeans package provides the ability to inspect (or "introspect," in JavaBeans terminology) arbitrary objects to determine the properties, events, and methods they expose (Hamilton, 1997).

Application developers will be able to use a variety of development tools to assemble custom applications from fully portable JavaBeans. JavaBeans enhance the Java platform by allowing richer, more dynamic interaction. JavaBeans allow developers to define independent components that can be used and re-used in a variety of combinations to compose new applications.

JavaBeans components are mostly GUI widgets, but they can be non-visual functions and services, applets and more full-scale applications as well. Different developers can build each of these components at separate times. JavaBeans components do not need to be part of the same application build. Instead, they communicate dynamically. JavaBeans are development components and are not deployment components (Roman, 1999). You typically do not deploy a JavaBean because a JavaBean is not a complete application; rather, JavaBeans help you construct larger software that is deployable.

Component Object Model

Microsoft's Component Object Model (COM), originally released in 1994, was the first component architecture to be used on a significant scale outside vendor laboratories. It is an evolution of Distributed Computing Environment (DCE) Remote Procedure Call (RPC). COM is now the foundation for all Microsoft's software -

including its Office suite, Windows and NT operating systems, and development tools such as Visual C++ and Visual Basic.

COM defines a binary standard for the definition of component interfaces. For each combination of operating system and hardware that Microsoft supports, COM defines a standard memory layout for a COM interface. Part of this standard is a specific set of supported data types that can be passed as arguments of operations in component interfaces. Microsoft implements COM on each of its operating system platforms (Sessions, 1997).

A COM component exposes a set of interfaces in a predefined way, and can be written in any programming language. COM components can be developed using the most commonly used languages such as Visual Basic, C, or Visual C++. Along with Microsoft's business strategy, language independence is one of the main reasons for COM's popularity among the developer community. COM developers can now choose from a wide range of commodity development tools, and the production of COM components with these tools is now a seamless process.

COM was originally positioned as a client-side component architecture that underpinned small, GUI-centric components; at the time this made a lot of sense because Microsoft had no server platform. But through its simplicity, and precisely because of all the things it does not specify, COM is suitable for any environment and any type of

component. It has emerged as a "lowest-common-denominator" software layer that can underpin Microsoft runtime services in any context.

The Windows operating system comes with a set of services that are COM components and provide services to other COM components. These services complete the framework necessary for COM:

- Microsoft Transaction Server (MTS) hosts COM components and manages distributed transactions.
- Microsoft Message Queue (MSMQ) provides message-based communications to COM components, including asynchronous invocations of COM calls.
- Internet Information Server (IIS) uses Active Server Pages (ASP) to translate HTTP requests into calls to COM components.

A new version of COM, COM+, improves the container architecture in which the components live with more notification and deployment-related functionality. A weak spot remains - the lack of persistence management that is still left to the developer. The distributed model of COM is known as DCOM.

Enterprise JavaBeans

Enterprise JavaBeans (EJB) is one of several technologies that make up the Java 2 Enterprise Edition (J2EE) specification. The J2EE specification is intended to define a

common multi-system, vendor-neutral platform for building applications using the Java programming language. The Enterprise JavaBeans component model logically extends the JavaBeans component model to support server components. Server components are reusable, prepackaged pieces of application functionality that are designed to run in an application server. They can be combined with other components to create customized application systems. Server components are similar to development components, but they are generally larger grained and more complete than development components. Enterprise JavaBeans components (enterprise beans) cannot be manipulated by a visual Java IDE in the same way that JavaBeans components can; instead, they can be assembled and customized at deployment time using tools provided by an EJB-compliant Java application server (Matena, & Hapner, 1999).

The Enterprise JavaBeans architecture provides an integrated application framework that dramatically simplifies the process of developing enterprise-class application systems. An EJB server automatically manages a number of complex middleware services on behalf of the application components. EJB component-builders can concentrate on writing business logic rather than complex middleware. The results are that applications get developed more quickly and the code is of better quality. The EJB model supports a number of implicit services, including lifecycle, state management, security, transactions, and persistence (Thomas, 1998). Capabilities like transaction processing support or persistence can be added without writing code. Beans are specifically packaged and deployed for execution on a server.

With the exception of EJB being a Java-only specification, it has identical goals to the CORBA Component Model. EJB is also the Java language mapping for CORBA components (Cobb, 2000). A brief overview of Enterprise JavaBean is provided here. A more detailed discussion is provided in Chapter 4.

CORBA Component Model

The CORBA component model (CCM) is a server-side component model for building and deploying CORBA applications. It is a programming language-neutral superset of the EJB specification. It was submitted by a group of application server vendors who provided CORBA-based products. Like EJB, it takes most common design patterns and standardizes their usage, enabling a large amount of code to be generated and system functions to be implemented by the container provider, rather than the application.

Since CCM has no dependency on a particular programming language, its packaging technology needs to be more comprehensive than EJB. It accomplishes this by building on the software packaging architecture being developed by the W3C to provide a complete vocabulary for describing binary executables. Like EJB, it focuses on the services needed for server applications (transaction, security, persistence, and events) and leaves most of the implementation details to the container, not the application.

An important feature of CCM is its specification of a CORBA-based container for hosting EJBs. This is essential to providing two-way interoperability between EJBs in Java and CCMs in any CORBA programming language. As part of CCM, CORBA IDL was extended to express runtime aspects from which tools can generate necessary helper components. The extended IDL is called Component IDL (CIDL). CORBA components can present multiple interfaces and can be partitioned so that only the part needed for handling a request needs to be loaded for increased efficiency.

A brief overview of the CORBA Component Model is provided here. A more detailed discussion follows in Chapter 5.

Comparison Table

The following table summarizes and compares the four component models:

Table 1

Comparison of Component Models

	COM	CCM	EJB	JavaBeans
Services	Transactions, Security, Event notification, load balancing, data caching	Persistence, Transactions, Security, Event notification	Persistence, Transactions, Security, Directory, thread pooling, message queuing	Events
Language Neutral	Yes	Yes	No	No
Cross-platform	No	Yes	Yes	Yes
Framework	OS Dependent Services	CORBA, OS Independent	EJB Server/Container OS Independent	Java, OS Independent
Server/Client Side	Both	Server	Server	Client

CHAPTER 4

Enterprise JavaBeans

Introduction

Enterprise JavaBeans (EJB) is one of many technologies that are part of the Java 2 Enterprise Edition (J2EE). EJB architecture is component architecture for the development and deployment of Java component-based distributed business applications. Applications written using the EJB architecture are scalable, transactional, secure and thread-safe. These applications may be written once and then deployed on any server platform that supports the EJB specification (Matena, & Hapner, 1999).

The EJB initiative was announced at the JavaOne '97 Conference and a draft was posted in December 1997. The EJB 1.0 formal specification was released on March 21, 1998. More than two years later, on October 25, 2000, version 2.0 of the specification final draft was released. From the feedback to the original version, many improvements and new features have been added to version 2.0. EJB is a programming specification, which is implemented as a product by an EJB vendor, which allows a Java application programmer to focus on the application rather than the infrastructure issues for server programs. EJB defines a container model and guarantees it to be the same regardless of the container product vendor. With the exception of being a Java-only specification, EJB

has identical goals to the CORBA component model (CCM). The CCM specification has ensured that EJB is the Java language mapping for CORBA components (Cobb, 2000).

Overview

An EJB component typically contains business logic that operates on the enterprise's data. The EJB instances are created and managed at runtime by a Container. The Container in which the EJB is deployed mediates client access. An EJB can be customized at deployment time by editing its environment entries. An EJB can be included in an assembled application without requiring source code changes or recompilation of the EJB. Different kinds of service information, such as transaction and security attributes, are separate from the enterprise bean class. This allows the service information to be managed by tools during application assembly and deployment. An enterprise Bean can be included in an assembled application without requiring source code changes or recompilation of the enterprise Bean.

The Bean Provider defines a client view of an enterprise Bean. The Bean developer can manually define the client view, or application development tools can generate it automatically. The client view is unaffected by the container and server in which the Bean is deployed. This ensures that both the Beans and their clients can be deployed in multiple-execution environments without changes or recompilation.

EJB Taxonomy

The EJB specification defines different types of enterprise beans for various application scenarios. The diagram below shows the classification EJBs – it is not a class hierarchy diagram.

Figure 1. EJB Taxonomy.

The EJB specification supports a flexible component model that defines most commonly used server-side components. The EJB component model supports three types of enterprise beans: Session, Entity, and Message Driven. The session EJB can be of

two types: stateless and stateful. A stateless session bean represents a stateless service. Stateless session beans are identical to a transaction processing (TP) monitor transaction in systems like BEA's Tuxedo or IBM's CICS. A stateful session bean represents a conversational session with a particular client. A message driven bean represents a stateless service whose invocation is asynchronous, driven by the arrival of Java Messaging Service (JMS) messages. Such session beans automatically maintain their conversational state across multiple client-invoked methods.

An entity bean represents a business object that can be shared among multiple clients (DeMichiel, Yalçinalp, & Krishnan, 2000). They are identified by a primary key and typically represent persistent data stored in a database. Their state can be automatically managed by the container (container-managed persistence) or managed by the bean itself (bean-managed persistence). A single application may use beans of both categories. Detailed descriptions of each type of EJB are provided in later sections of this chapter.

EJB Architecture

The diagram below, Figure 6, Figure 2. EJB Architecture shows the pictorial view of EJB architecture. The main components are an EJB server, container, security service, transaction service, persistence service, and naming service. An EJB server, which is provided by a vendor, can host one more containers. A container is where EJBs are deployed. "Container" is a logical term that defines the part of the EJB server that hosts instances of enterprise beans. The container is responsible for managing transactions, security, concurrency, persistence, and resources that the enterprise bean uses at run time (Matena, & Hapner, 1999).

Figure 2. EJB Architecture

An EJB component consists of four important parts: a home interface, a remote interface, a bean class, and an XML deployment descriptor. A logical view of an EJB component with the four parts is depicted in Figure 3. The home interface declares factory methods for life cycle management that are used by Java clients to create new enterprise beans, locate existing enterprise beans, and destroy beans. The remote interface declares the business methods of the enterprise bean, which are used by Java clients at run time. The EJB implementation class encapsulates the application logic of

the bean, and implements the business methods defined in the remote interface. A remote Java client can only access the methods exposed in the remote interface; however, a Java class on the server-side can access the other “non-remoted” methods in the EJB implementation class.

Figure 3. EJB Component. Contents of an EJB JAR File.

The deployment descriptor is an XML configuration file that describes the EJB, and its run-time attributes to the EJB server when it is deployed. It is the responsibility of the bean developer to create the four elements of the EJB and package them into a JAR file for deployment. Based on these four parts and the deployment descriptor, the EJB server vendor tools can generate the classes needed for the container, in which the EJB will be deployed, and the server to interact. The container generates proxy classes that

implement the EJBHome and EJBObject (remote) classes. Typically, these additional generated classes are also added to the JAR file for deployment. The container is the logical term that defines the part of the EJB server that hosts the instances of EJBs. The EJB server holds the container and the implementations of the home and the bean objects. It uses deployment descriptors to manage policies for transactions, security, and persistence. There is one EJBHome for every EJB deployed and one EJBObject for every remote reference to an EJB.

Session Beans

A session EJB is a component where each instance is mapped to a specific client connection. This means that a session bean is never shared among different clients at the same time. Session EJBs represent non-persistent, transaction-aware, server-side components. They are used to model processes, services, and client-server sessions. There are two types of session EJBs: stateful, and stateless.

A stateful EJB is dedicated to the Java client that created it. Its class defines instance variables that can hold session data from one method invocation to the next, which makes methods interdependent. This design allows the bean to act as an extension of the client, maintaining session data on the EJB server, and executing tasks on behalf of the client. Stateful session beans are used to model business concepts that represent agents or roles that perform tasks on a client's behalf. Examples of such beans may

include a bank teller that processes monetary transactions, a shopping cart used in e-commerce, or a travel agent that makes flight, hotel, and car-rental reservations.

Unlike stateful session EJBs, the stateless session EJBs are shared by many Java clients. A single stateless EJB instance could be shared between many clients. As stateless session EJB instances are shared between many clients between method calls, a client cannot reliably store its data in the instance variables. The stateless session EJB method calls are like API calls in a function library; session data cannot be maintained between method calls, and method calls are not interdependent. Each invocation is effectively stateless and is analogous to static methods in Java classes. Sharing of stateless session EJB instances makes it more scalable than stateful session EJBs with the complexity of added client-side session data management. Stateless session EJBs are used to model stateless services which perform some function based on the input data, for example, credit card processing, financial calculations, and queries.

Entity Beans

An entity EJB is the object representation of an entity stored in an underlying data source. This data source is usually a database, but can also be any other application. Entity EJBs are persistent, transactional, server-side components. The entity EJBs are typically used for modeling business domain objects, which have a persistent

representation in a database or an enterprise application. The type of data modeled by an entity EJB could be an employee record, bank account, and sales or purchase orders.

The protocol for transferring the state of the bean between the underlying data source and the bean's instance variables is referred to as "object persistence". The lifetime of such an object is not limited to the time the server is running since it is only a representation of data that is stored in a persistent data source. An entity EJB definition can be compared to a relational database table, and the instances of that EJB can be compared to the unique rows of that table. As a row in a table would have a primary key to identify it, entity EJBs also have a key to uniquely identify them.

The persistence mechanisms can be implemented in two ways: one is called bean-managed persistence (BMP), and the other is called container-managed persistence (CMP). In the BMP mechanism, the EJB itself has to handle the persistent state management. In the CMP method, the state management is delegated to the container. In this case, the container will use the mapping data in the deployment descriptor to perform automatic state management. Based on the deployment descriptor, the necessary classes are generated for data source access.

Message-Driven Beans

Message-driven beans are stateless, server-side components that are used to receive and process messages that Java clients send to them via Java Message Service (JMS). The message-driven bean was newly added to the EJB 2.0 specification due to the growth of JMS-based applications (DeMichiel, Yalçinalp, & Krishnan, 2000). They are used to model processes and routers that operate on inbound enterprise messages. Developers might use message-driven beans to integrate an EJB system with a legacy system or to enable business-to-business interactions. While a message-driven bean consists of a bean class and an XML deployment descriptor, it does not have a remote or home interface. Message-driven beans are not distributed components and do not have EJBObject and EJBHome references. They are meant as the server-side responders of asynchronous messages delivered from JMS. There are specific, or direct, clients for message-based beans. The clients are JMS clients that send messages to a specific JMS queue or topic on which a message-driven bean is “listening.”

EJB Development Process and Roles

In addition to specifying the architecture for EJB, the EJB specification also defines the process and roles in the EJB application development and deployment lifecycle. As the goal of *EJBrew* is to provide a solution for EJB application development by composition, it is important to get an understanding of the different roles

related to the process of implementing an application within the EJB framework. There are seven roles, which may all be adopted by different people. The specification ensures the interoperability of the involved parties.

- **Enterprise Bean Provider:** The bean provider is in charge of writing the beans, which contain the business logic. Each bean includes its class files, the specific remote and home interfaces and its deployment descriptor. The output from the EJB provider is a JAR file containing one or more EJB implementation class files, associated home and remote interfaces, and the deployment descriptors. As the EJB provider is implementing business logic, he or she is typically an application domain expert. The EJB provider does not need to handle the system-level programming of security, transactions, concurrency, and distribution as they are handled and managed by the container.
- **Application Assembler:** The role of the application assembler is to compose code, using beans contained in an ejb-jar file. This code can be other EJBs or any other Java application, applets, JSP pages, or servlets. The output from an application assembler is one or more ejb-jar files containing larger deployable application units. The application assembler inserts the assembly instructions into the deployment descriptor.

The EJB specification focuses on having the application assembly step prior to deployment. However, the EJB architecture does not prevent post-deployment application assembly (DeMichiel et al. 2000), which is one of the primary goals of the *EJBrew* solution.

Just as EJB Producer is a domain expert, the Application Assembler is also a domain expert; but, the domain covered by each role is quite different. In the EJB Producer's case the domain is at an individual bean-level, and the beans could be used in many industries. In the case of the Application Assembler, the domain is typically a specific industry, and the assembled EJB could only be useful to that industry.

The Application Assembler does not need to know the internal implementation of the EJBs being used for the application. The Application Assembler works with the client-view of the EJB and needs to know only the home and remote interfaces of the EJB.

- **Deployer:** The responsibility of the deployer is to install and configure a bean, which is packaged in an ejb-jar file provided by a Bean Provider or an Application Assembler, in a specific operational environment. The operational environment for an EJB is a vendor-specific EJB Server and Container; to do so, the deployer can manipulate the deployment descriptor related to the bean. The

deployer neither needs to know the details of the business logic implemented by the bean nor the details of the container and server implementation.

The Deployer is an expert in the configuration and management of the operational environment that includes the EJB server, container, operating system, databases, and other enterprise system resources. The steps in deployment are as follows:

- Resolve external resources references of the EJB in the deployment descriptor.
 - Specify user and security information and configuration in the deployment descriptor.
 - Using the deployment tools of the specific EJB server, generate additional classes and interfaces needed for the container. The container for run-time management of the EJB uses these classes.
 - Actually install the EJB and additional classes, and interface into the EJB Container.
- **EJB Server Provider:** The Server Provider is an expert in distributed computation, transaction management, and low-level system processes. The server provider is responsible for the implementation of the EJB Server runtime. As the current EJB specification does not architect an interface between the server and the container, they are very tightly coupled, and the EJB server vendor and

container provider are the same. A typical EJB Server Provider is an operating system vendor, database vendor, or middleware vendor.

- **EJB Container Provider:** The EJB Container Provider's primary responsibilities are to provide deployment tools and runtime support for the deployed EJBs. The services provided by the container are transaction and security management, network distribution of clients, scalable management of resources, and other operations and management services. As the EJB specification does not define the architecture between the EJB container and server, the vendors decide the distinction between them.
- **Persistence Manager Provider:** When an entity EJB uses container-managed persistence (CMP), the state management services are handled by the container. For this service the container uses a persistence manager that is either provided by the EJB server/container provider or by another provider. In the first case the EJB server/container provider is also playing the role of a Persistence Manager Provider. In the case where the Persistence Manager Provider is another vendor, that vendor and the EJB server/container vendor must agree on the interfaces between them as the current EJB specification does not architect such an interface. The Persistence Manager Provider is responsible for providing a service that does the following:

- Tools to generate code that moves data between the entity EJB and a database or an enterprise application.
 - Manages persistent state-of-the-entity EJB and dependent objects.
 - Maintains referential integrity of the relationships among the entity EJBs and dependent objects.
 - Is responsible for generating code for and executing finder and select methods based on EJB Query Language (EQL).
-
- **System Administrator:** The System Administrator has to take care of configuration and administration of the enterprise's computing and networking infrastructure, which includes the EJB Server and Container. The System Administrator is also responsible for overseeing the well-being of the deployed enterprise beans applications at runtime using operations and management tools provided by the EJB server/container provider.

CHAPTER 5

CORBA Component Model

Introduction

Common Object Request Broker Architecture (CORBA) is a standard architecture for distributed object systems. It allows a distributed, heterogeneous collection of objects to interoperate. Being heterogeneous is probably the most important reason for CORBA's adoption by the industry: whereas, EJB is a Java-only specification, CORBA has mappings in C++, Java, and COBOL as well as other languages. The Object Management Group (OMG) is responsible for defining CORBA. The OMG comprises over 700 companies and organizations, including almost all the major vendors and developers of distributed object technology, including platform, database, and application vendors as well as software tool and corporate developers.

The most current version of the CORBA specification is 3.0, and it is currently being finalized. With CORBA 3.0, the OMG introduced the CORBA Component Model (CCM).

The CCM specification was submitted by a group of vendors led by IBM, BEA, Oracle, Iona and Inprise. Ovum's analysis of those vendors' own application-server

technology shows that CCM has its roots in some work that originally found its way into IBM's ComponentBroker product (Ward-Dutton, 2000).

The CCM specification is a component model for building server-side, enterprise class applications. Even though the CCM is based on the CORBA framework, it shares the base architecture with EJB. Based on this, CCM helps EJB to extend beyond the Java programming language. The CORBA CCM is a cross-language superset of Sun's EJB specification. The CCM specification dictates how EJBs should interact with CORBA components, and how a CORBA-based container hosts EJBs (Cobb, 2000).

Like EJB, 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 the 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.

Overview

The CORBA Component architecture consists of several interlocking conceptual pieces that enable a complete distributed enterprise server computing architecture. These include an Abstract Component Model, a Packaging and Deployment Model, a Container Model, a mapping to EJB and an Integration Model for Persistence and Transactions.

A typical CCM Architecture consists of:

- CCM Containers
- CORBA components that run in these containers
- Portable Object Adapter (POA)
- Object Request Broker (ORB)
- CORBA object services like CORBA Transactions, CORBA Security, CORBA Persistence, CORBA Events, etc.

In a typical development and deployment scenario, there will be an Application server provider who creates and sells an Application server along with CCM containers that will run on these servers. Then there will be the CCM providers-people responsible for developing CORBA component, and the Application assemblers-people that use pre-built CORBA components to build their applications.

By scoping the design space to these few patterns, much of the tedious work can be shouldered by code generation tools. This is analogous to the work that the current IDL compiler does to generate all the marshalling and remote invocation code in IDL stubs and skeletons. The difference is that now instead of generating stubs and skeletons, it will be possible to generate most of a server within a CORBA Components framework, into which a developer can “drop” his business logic, or component.

The Object Management Architecture (OMA) framework for supporting the definition, code generation, packaging, assembly, and deployment of these CORBA components is collectively called the CORBA Component Model (CCM). It represents a major extension and addition to the OMA and CORBA.

The CORBA CCM specifies a number of different container aspects that support, between them, four different varieties of server-based components: Service, Session, Process, and Entity.

Components are described using an extension to CORBA IDL called CIDL, which allows developers to describe not only a component's interfaces, but also the “policy” that will dictate how the component will be managed at runtime. Tools to generate all the “helper” components that manage the interactions between application developers' components and a CORBA CCM container use these descriptions.

The CCM adds to IDL the ability to define CORBA components and introduces CIDL to describe implementation details sufficient to allow the entire server-side framework to be generated, assembled, and deployed.

In the CORBA CCM, components can present multiple interfaces: process and entity components can also be "partitioned," with each component partition's persistence being managed independently by the container. This feature is aimed at enhancing efficiency: When a partitioned component is deployed and a client requests a component operation, only the component partition associated with the requested operation is brought into memory.

- It defines interfaces necessary to support navigation among the multiple interfaces supported by a CORBA component.
- It defines a mechanism for tailoring CORBA components prior to deployment using both metadata defined by the component model and runtime properties that can be tailored using a design tool.
- It introduces a deployment model to CORBA using XML to describe the run-time properties of a CORBA component.

Abstract Model

The figure below, Figure 4, depicts all the features of a CORBA component. The CCM clients access the CCM application function via *facets*.

Figure 4. CCM Abstract Model

CCM defines two types of components: basic and extended. Basic components essentially provide a simple mechanism to "componentize" ordinary CORBA objects without adding a significant amount to the programming model. The basic component is

equivalent to a EJB 1.1 specification enterprise bean. Extended components provide a much richer set of functionality than the existing CORBA model as outlined below.

Components support a variety of surface features. New features that support ways of “connecting” to components are called ports and include:

- **Facets:** Distinct named interfaces and navigation between them provided to clients. Exposes navigation among the application facets as well as operations on existing objects that have been “componentized.” Facets expose application functions that can also be connected to other components at deployment time.
- **Receptacles:** Named connection points that describe the ability to use a reference. Other Components declare their external dependencies that are resolved at deployment time using receptacles.
- **Event Sources:** Named connection points that emit events of a specified type to one or more interested consumers or to an event channel. Publishers are exclusive providers; emitters share event channels with other event sources.
- **Event Sinks:** Named connection points into which events of a specified type may be pushed. Consumers of events generated by other components or notification channel users.

Other features of the model include:

- **Primary Keys:** Values exposed to clients to help identify particular components.

- **Component Attributes and Configuration:** Named values exposed through accessors and mutators, primarily used for component configuration
- **Standard Home:** Interfaces that provide standard factory and finder operations

In CCM, a component has a set of features which applications can exploit in building business applications. These include features for composition (facets and receptacles), loose coupling using events (publishers, emitters, and consumers), and configuration (attributes). Every component is managed by a home that provides lifecycle and other services on behalf of the component.

Interfaces

CCM component features are exposed via several categories of interfaces. In this section these interfaces are discussed. A component exposes its functional application behavior to its clients using facets. Facets are defined by Supported and Provided interfaces. A connection point called a receptacle is used by a component to receive object references upon which it can invoke operations. The receptacles of a component are defined by “Used” interface.

Supported interface allows existing objects to be turned into a component. Supported interface provides EJB equivalence for basic components. Basic components

are declared using the supports clause on the component declaration. The supports clause refers to a single interface or a set of interfaces related by inheritance.

Provided interfaces are the primary way a new component exposes its business operations. Provided interfaces are “ports” that *uses* declarations can be connected to at deployment time. Clients can navigate to a specified interface at run time using navigation operations generated by the “provides” declarations. Clients can use provided and supported interfaces as the “Navigational” interface to find out if a component provides certain operations.

Uses declarations define the dependencies of a component for an interface provided by another component or a standard CORBA object. A single component can use the same interface multiple times. This is identified using the multiple keywords. Uses declaration result in the construction of a receptacles object into which the “provides” declaration is connected. An example component definition is shown below. in Figure 5:

```

interface Sell, Buy;
//Define an equivalent, supported interfaces
component Stock_Exchange supports Sell, Buy {

    provides Stock_Quote;    // Facet
    consumes Buy_Offers;     // Event Sinks
    consumes Sell_Offers;
    publishes Price_Change; // Event Source
    uses SEC; // Receptacle
// ..... Other definitions
};

```

Figure 5. Sample CORBA Component Definition in CIDL.

Events

Components can generate and receive events. Two forms can generate events: publishes and emits. A publishes declaration establishes an exclusive path between the supplying component and an arbitrary receiver. An “emits” declaration uses an existing channel shared by multiple suppliers (not necessarily components) to broadcast events.

Components can also receive events generated by other components or a notification service user. This is specified using the “consumes” clause. Component events are represented as CORBA value types. They are transmitted as structured events enabling full interoperability with other users of CORBA notification.

Navigation

The CCM specifies navigational features for CCM clients to locate CCM components. Since components may have multiple interfaces, the CCM model provides multiple mechanisms to navigate from any facet to any other facet. Navigation is possible from any facet to component facet with `get_component()`, or from component reference to any facet via generated facet-specific operations, or via generic navigation operations.

Attributes

CCM components contain attributes for configuration purposes. CCM recommends use of attributes for configuration. It does not recommend the use of attributes for business operations. It does not enforce the use of attributes for component configuration and not for business operations.

Configuration is considered to be key for effective re-use of components. Component configurability makes a component more adaptable to different application needs. The CCM specifies a *Configurator*, which captures a collection of attribute settings in a components home and applies them to all instances of the component created by that home.

Application Servers

These provide the system services like a raw execution environment, multiprocessing, load balancing, device access, provide naming and transaction services and make containers visible.

CCM Containers

These act as the interface between a CORBA component and the clients. A CCM client never accesses a CORBA component directly. Any component access is done through container-generated methods that in turn invoke the component's methods. There are basically two basic types of containers. They are transient containers that may contain transient, non-persistent components whose states are not saved at all, and persistent containers that contain persistent components whose states are saved between invocations.

The container architecture defines seven different container categories: four that correspond to the four component categories, two for the EJB container API types, and an empty container to support user-defined frameworks:

- The Service category manages the service category for access to stateless components.

- The Session category manages the session component for stateful CORBA components with transient state.
- The Process category manages stateful process components that encapsulate data access in the server.
- The Entity category manages stateful entity components that share data access responsibility between the client and server.
- The EJBSession category manages EJB session beans.
- The EJBEntity category manages EJB entity beans.
- The Empty category provides no automatic management, but makes the standard CORBA 3.0 interfaces available to the component implementation.

A container, written in Java, can be an EJB container by supporting one of the EJB container API types.

Container Framework

CCM's Container Framework is a framework for server applications. It is built on the CORBA's Portable Object Adaptor (POA) architecture. This container framework provides component clients, supports interfaces for systems services such as events, persistence, security and transactions. The figure below, Figure 6, depicts the CCM container framework and related CORBA infrastructure.

Figure 6. CCM Container Framework

CCM Clients

These make use of the CCM components for their operations. They find the CCM container that contains the component through COSNaming and create, or obtain a reference to the component's container through the component's Home Interface. They then make use of the CCM Container to invoke the component methods.

CCM Components

CCM components can expose multiple interfaces. While defining the IDL for each component, the interfaces that it provides and the interfaces that it uses have to be specified. Similarly, the events that it emits (or publishes) and the events that it consumes have to be defined in the IDL. The interfaces that it provides are called facets. The ability to use an interface is facilitated by declaring it as a receptacle. Event emission and publication declarations are used to define an event source. Event consumption declarations are used to describe event sinks. Facets, receptacles, event sources and event sinks are collectively called Ports.

CORBA components have four categories of components. Three are exactly like their EJB equivalents. The fourth, the *process* component, has the same server characteristics as an entity, but does not expose its primary key to the client (i.e. a keyless external type). The process category has its genesis in the TP Framework of BEA's WebLogic Enterprise, which supports this behavior, albeit with its own programming API rather than the CCM standard. This makes it useful for modeling business processes rather than business entities and discourages programmers from re-inventing 2-tier programming using component models.

There are four types of CCM components. They are Service components, Session components, Process components and Entity components.

- **Services**, which are stateless components that have behavior, but no identity, are best suited to implementing simple command execution components (for example, the "wrapping" of a legacy transaction).
- **Sessions**, which are transient components that have behavior, but only need to take account of state for short periods of time, and do not need to have persistent identifiers, are best suited to implementing things like shopping baskets within e-commerce applications.
- **Processes**, are like sessions, but with persistent identities and the ability to store their state persistently.
- **Entities**, have behavior, persistent state and a persistent entity that is accessible to application developers via a primary key (just like a row in a database table).

Processes' and entities' behavior can involve the execution of transactions - and for both of these component types, transaction execution and persistence can be delegated to the container or explicitly managed by programmers.

Persistence

Persistence in Entity and Process components is of two types. They are Container-managed persistence, and Component-managed persistence.

In Container-managed persistence, the CCM container is responsible for saving the component's state. Since it is container-managed, the implementation is independent of the data source. The container automatically handles persistence.

In Component-managed persistence, the Entity component is directly responsible for saving its own state. The container does not need to generate any database calls. Hence the implementation is less adaptable than the previous one as the persistence needs to be hard-coded into the component.

Interfaces Between the Application Server and CCM Containers

The CCM specification defines a standard way to implement the container API through CORBA object services. Therefore, the interfaces between the Server and the container are well specified. In fact, since EJB is a subset of CCM, vendors can now implement the EJB container API through CORBA object services.

Component Configuration

The configuration architecture provides mechanisms to distinguish between design time and run time. A component's configuration is established primarily through its attributes. An attribute configuration is defined to be a description of a set of invocations on a component's attribute set methods, with specified values as parameters.

The CORBA component model provides mechanisms to support the concept of component configurability. Experience has proven that building re-usable components involves making difficult trade-offs between providing well-defined, reasonably-scoped functionality, and providing enough flexibility and generality to be useful (or re-useful) across a variety of possible applications. Packaging assumptions of the component architecture preclude customizing a component's behavior by directly altering its implementation or (in most cases) by deriving specialized sub-types. Instead, the model focuses on extension and customization through delegation (e.g., via dependencies expressed with uses declarations) and configuration. Our assumption is that generalized components will typically provide a set of optional behaviors or modalities that can be selected and adjusted for a specific application.

The configuration framework is designed to provide the following capabilities:

- The ability to define attributes on the component type that are used to establish a component instance's configuration. Component attributes are intended to be used during a component instance's initialization to establish its fundamental behavioral properties. Although the component model does not constrain the visibility or use of attributes defined on the component, it is generally assumed that they will not be of interest to the same clients that will use the component after it is configured. Rather, it is intended for use by component factories or by deployment tools in the process of instantiating an assembly of components.
- The ability to define a configuration in an environment other than the deployment environment (e.g., an assembly tool), and store that configuration in a component package or assembly package to be used subsequently in deployment.
- The ability to define such a configuration without having to instantiate the component type itself.
- The ability to associate a pre-defined configuration with a component factory, such that component instances created by that factory will be initialized with the associated configuration.
- Support for visual, interactive configuration tools to define configurations; specifically, the framework allows component implementers to provide a configuration manager associated with the component implementation. The configuration manager interface provides descriptive information to interactive

users, constrains configuration options, and performs validity checks on proposed configurations.

Component Declaration and Implementation

A CORBA component is specified using (new extended IDL) CIDL. These extensions to IDL comprise the client view of components. The output is client-side stubs and metadata descriptors and IR entries.

It allows for platform and language-independent specification of features such as integration with transaction, persistence, events, etc. The component is implemented by “filling in” the business logic into the generated skeletons and servants. Code is written in a concrete programming language, possibly using a CORBA-component aware IDE, or just a basic text editor and compiler. The output is an elaboration of the compiled code produced in the declaration phase.

The Component Implementation Framework (CIF) defines the programming model for constructing component implementations. Component Implementation Definition Language (CIDL) is used for connecting the various pieces of the component implementations together so that they can cooperate with the container.

Component Packaging

A component implementation ultimately takes the form of a component package, an archive file containing one or more executable realizations of the component implementation and metadata describing the component type. Component packages have two potential uses. A component package may be loaded into an application assembly tool and used to construct an assembly of components that becomes a package itself. Alternatively, a component package may be deployed directly into an execution container, allowing instances of the component type to be constructed and used by application clients.

CCM uses XML descriptors for specifying information about packaging and deployment just like EJB. However, additionally, CCM has an assembly descriptor, which contains metadata about how two or more CCM components are wired together.

A component archive file contains the component implementation and component descriptor. It is created and the component implementation is added to the archive.

Component Descriptor

The component descriptor specifies additional descriptive information specific to CORBA components. It describes the category, structure, and properties of the

component. It identifies the component category and defines the policies to be applied to the component at runtime.

Component Assembly

The component is customized, connected to other components, and partitioned producing a component assembly archive file. An assembly is a description of a collection of prototypical components (that may have been customized) along with a description of their relationships. Using a (visual) design tool that presents a set of available components, components may be:

- Customized using property editors
- Connected to one another
- Partitioned by specifying partitioning sets of components that are to be deployed together within a single process or host

A Component assembly is a group of interconnected components and component homes represented by an assembly package. The output is a component assembly archive file that contains a set of component archives and a descriptor describing the assembly.

Component Deployment and Installation

In the field, a deployment/installation tool is used to deploy and install components and assemblies to particular machines on the network. The tool can deploy single components or complex component assemblies. It is used to install components and homes and to instantiate the initial connection topologies, on the deployment machines.

Components and assembly packages are deployed (installed) on target hosts on a network using a deployment tool or application. Components within an assembly may be installed on multiple machines.

CCM and EJB

Basic CORBA components are functionally equivalent to EJB 1.1 components. An EJB deployed into a CORBA component container is a CORBA component. The Java language mapping of the CCM APIs is the EJB 1.1 APIs. A CORBA client sees an EJB as a CORBA component. An EJB client sees a CORBA component as an EJB.

Basic CORBA components were introduced into the CCM to achieve full interoperability between EJBs and CORBA clients and CCMs and EJB clients. Among other things, this permits basic Java CORBA components to be written to the EJB 1.1

API, making an EJB a full-fledged CORBA component (when it is deployed on an EJB container implemented on CORBA). Java clients using the EJB client-programming model can view CORBA components as if they were EJBs and CORBA clients can view EJBs as if they were CORBA components. The CORBA component specification defines the transforms necessary to make this work seamlessly.

The basic component model defined by this specification is equivalent to the EJB 1.1 Component Architecture. As a result of this equivalence, an enterprise Bean is a CORBA component. The extended model contains additional functions beyond the EJB 1.1 specification including events and tighter integration with the CORBA object model.

The specification defines a mechanism for a basic CORBA component to be accessed by an EJB client and a CORBA client to access an enterprise Bean as if it were a CORBA component. This permits two-way interoperability between components and allows the construction of applications made up of both enterprise Beans and CORBA components.

All features of the EJB component architecture are accommodated in the basic CORBA component model, either directly or by the EJB to CORBA inter-working mappings, permitting two-way interoperability between EJB and CORBA components.

Figure 7. Interoperation in a Mixed Environment

Components of one kind, either EJB or CCM, are made accessible to clients of the other by combining the generation of the appropriate bindings (stubs) at development time and by the appropriate translation of method invocations by the bridge at runtime. Hence from the perspective of the CCM client, the EJB appears to be a CORBA component, and from the perspective of the EJB client, the CORBA component appears to be an EJB. Because there is a very close match between the two models, unlike interworking between other models, the bridge is “thin” and hence, should be quite efficient.

The CCM specification defines compliance points to support seamless interworking between the CCM and EJB 1.1 implementations. It defines 2 levels of the CCM: a Basic level that is a subset of the full CCM, which corresponds in functionality

to that supported by EJB 1.1, and an Extended level that encompasses the entire functionality.

CHAPTER 6

Requirement: Enabling Application Composition for EJBs

J2EE Application Architecture

In order to understand an enterprise Java application consisting of EJBs, it is necessary to understand the application architecture specified by the Java 2 Platform Enterprise Edition Specification (Sun Microsystems Inc., 2000, May 23). Sun's recent release of the Java 2 Enterprise Edition reference implementation, or J2EE RI, provides an implementation of all the J2EE functionality, including technologies such as Enterprise JavaBeans (EJB), CORBA, XML, Servlets, and JavaServer Pages (JSP). Along with the libraries and examples in the reference implementation package comes a new white paper from Sun describing its vision of how Java applications should be designed and built. This white paper is a collection of papers that document best practices architectures for J2EE-based enterprise applications. The set of documents is called *J2EE Blueprint* (Sun Microsystems Inc., 2000, March 20). The high-level architecture of an end-to-end J2EE application is shown in the following diagram:

Figure 8. J2EE Application Architecture

The Blueprint suggests that EJBs be used as the gateway to the enterprise systems, and that JSPs or Servlets connect EJBs to the web-based clients.

Software Development Using IDEs

In modern software development, the use of an Integrated Development Environment (IDE) is very common. Typically, once a programming language or framework is introduced, availability of supporting software development tools helps in the growth of that language. The most common tools are IDE, debuggers, profilers, and modelers.

The number of features provided in an IDE grows over time with each version or release of the IDE. For example, initially Java IDEs such as Symantec Visual Café provided an integrated text editor with a compiler/parser, debugger, and file management. Over time, this tool (currently known as WebGain Studio) has grown into a full-blown IDE with many features such as:

- Support for Java Enterprise Development (J2EE: JDBC, Servlets, EJB, JSP)
- Code Completion
- Live Editing
- Context-sensitive Help
- Distributed Debugging
- UML Modeling

Many IDEs exist for EJB development; each has its strengths and weaknesses. Following is a summary of capabilities of popular tools, from the perspective of J2EE application composition:

Table 2

Features of Java IDEs

Features	WebGain Studio	Visual Age	JBuilder	COOL:JOE	Rational Rose
Create & Import EJB	Yes	Yes	Yes	Yes	Yes
Deploy EJB	Yes	Yes	Yes	Yes	No
Dynamic Discovery of EJBs	No	No	No	No	No
UML Model EJB	Yes	No	No	Yes	Yes
Support for JSPs	Yes	Yes	Yes	No	No
Support for Servlets & JDBC	Yes	Yes	Yes	Yes	No
Application Server Support	Yes	Yes	Yes	Yes	No
“Wiring” Components	JavaBeans	JavaBeans	JavaBeans	JavaBeans	No
Distributed Debugging	Yes	Yes	Yes	No	No

EJB Application Composition Using IDEs

In this section, the process of developing EJB-based application using an ideal IDE is discussed. Although the discussion here is based on EJBs, it is applicable to any type of server-side component.

Building New or Identifying Available Components

Almost all of the current IDEs provide the feature to build new EJB components. New EJBs are usually built using some “Wizards” that can create the necessary class files and deployment descriptors for a new EJB. These wizards create skeleton files with markers to place the application business logic.

Very few, if any, IDEs provide features to list and identify existing EJBs by searching JAR files containing EJBs or by querying EJB deployed servers. This is a critical feature needed for any application composition using components, and CBD mostly involves using off-the-shelf components. A feature to add a list of existing components to an IDE is a very critical requirement.

As Java 2 Enterprise Edition (J2EE) specification application architecture suggests, EJBs are called by other J2EE technologies such as Servlets, JSPs and other EJBs. So, it is necessary for the IDE to list available Servlets, and JSPs or provide capabilities to create new Servlets or JSPs : current Java IDEs provide these capabilities. Although the focus of this thesis is on EJBs, the principles can be applied to Servlets and JSPs as well.

Most of the current IDEs currently provide a mechanism to add existing JavaBeans components to their IDEs. The same feature is required for EJB components.

Selecting Necessary Components

Once an IDE can display the available EJB components, it is necessary to be able to select the required EJBs for an application by querying, browsing, or by viewing their properties.

Instantiating Components

After required EJBs are selected, they need to be instantiated and placed on a working area such as a “Whiteboard.” As EJBs are server-side components, there is no need for them to be laid out in any particular location as it would be for GUI components.

Configuring Component Instances

Typical EJB components are built and shipped with some default values set for their properties. An IDE should provide property sheets that would allow a developer to customize EJBs using their configuration parameters.

Connecting Component Instances

Once components are placed on a “Whiteboard”, their programmatic interaction needs to be visually specified by connecting them with each other as necessary. In order to perform this, it is necessary to perform the following tasks:

1. Select an EJB instance (source) by right clicking by the mouse
2. Select another EJB instance (target) by right clicking by the mouse
3. At this time, the second EJB instance, the target, should contextually list the available operations.

Contextual presentation of the available operations of an EJB requires that the object representing the EJB in an IDE should be able to provide complete information about the EJB to the IDE’s user interaction functions such as mouse clicks and menu actions. The following diagram depicts this requirement:

Figure 9. EJB and IDE Interaction

In the above diagram, the EJB Remote Interface contains the business methods (operations) available from an Enterprise JavaBean class. The EJB Home Interface provides the lifecycle management methods for the EJB. These two interfaces are as specified by the EJB 2.0 specification. An EJB IDE View class is required to provide the IDE a single point-of-view of the EJB. The IDE Actions class presents the available operations to the user, upon a mouse-click or menu action, from the EJB IDE View class.

Currently, there are no standard specifications for the mapping of the operations from the EJB Remote Interface and the EJB Home Interface to the EJB IDE Class. To achieve EJB application composition using IDEs there is a need for such specification.

Specifying Connection Parameters

A connection (“wiring”) between two EJBs only indicates that there is a programmatic interaction between them. As a programmatic interaction, two EJBs is a method call from one EJB to another. A method usually takes zero or more parameters, and returns nothing, or some value (it could be a reference to an object). In order to complete the specification for the connection, it is necessary to determine the number of input parameters, their types and if there is a return value, and, if so, the type of the return value.

For a user to complete the connection specification by specifying the input parameters and a place to store the return value, it is necessary for the IDE to be able to determine this information based on the method being called on the target EJB. Based on Figure 9, the requirement mandates that the IDE EJB View class be able to provide method signature (name, parameters, return type) to the IDE Action class.

Application Code Generation

Composition of an EJB application by “wiring” should store the inter-connection information in a persistent store. The simplest persistent store is a disk-based file; alternatively a database may also be used. When interconnection information is stored,

optionally the source code for the application may also be generated in real-time; on the other hand, the code could be generated once all the wiring has been completed.

In Java, all program source codes are part of some class. The “main” program is a static method of a Java class that initializes and invokes a program or a “service” method in a Servlet that is invoked by an event. Based on this, all the programmatic interactions with EJBs will be from another EJB, or a stand-alone Java class that is a client program, or a Servlet, or a JSP. As such, the code generation would only have to generate code within method bodies, or there may be a requirement to generate a Java class that could be used as a test driver.

Code generation requires that the IDE be able to parse the “wiring”/inter-connection information file and generate the code within the respective method bodies. For example, if object A’s method transferFunds calls object B’s withdrawFunds method, then the body of the method transferFunds need to be modified to have B.withdrawFunds(). Typical code generation would not create new class files; it would only modify the methods of existing classes.

However, in advanced use of the IDE for EJB application composition there may be a need for additional EJB creation to utilize existing EJBs. This would most likely be the case of Session EJBs that would interact with Entity EJBs.

Creating Deployment Packages

Current IDEs are capable of creating J2EE-based application deployment packages from the class files and deployment descriptors. EJB application composition using IDEs would not require any new features, as the application composition would only modify existing EJB classes or add new EJB classes.

When new EJB classes are added, new deployment descriptors need to be created as well. The current IDE tools would also handle this.

As EJB application composition uses additional informational classes like the EJB IDE View, it may be useful to include those classes with the deployment package to provide quick information regarding the available EJBs during runtime or to another development session that might initiate its application development from a deployment package.

Development-Time and Runtime Requirements

An application design phase requires more information of the involved components than during the runtime of that application. For example, in the JavaBean development BeanInfo classes are provided for additional development time information; these classes are not required during the runtime of the applications based on the

JavaBeans. In the case of JavaBeans-based commercial components, it is common for the vendor to ship two different packaging of the same set of components: one “thin” package with runtime only classes, and one “fat” package with runtime and development time classes.

Likewise, it would be desirable for the composed EJB application to have isolatable development-time information. As separation of run-time and development-time classes would require additional maintenance and tracking, it would be desirable to be able to generate the development-time information on-demand based on the deployed run-time information.

Summary of the Requirements for EJB Application Composition

Following is the summary of all the requirements gathered for enabling EJB for application composition using an IDE:

An IDE should

- Provide features to build new EJBs and other J2EE components (Servlets, JSP).
- Be able to discover existing deployed EJBs from Web application servers and deployed/un-deployed EJBs from packaged JAR files.
- Provide capabilities to provide information on the available EJBs, so that users can select the desired EJBs for their application.

- Provide “Whiteboard”- like features so that a user can place EJBs and interconnect them as desired.
- Enable EJBs to be visually connected by selecting source and target EJBs, and be able to label the connections uniquely.
- Be able to contextually provide information on available methods on the selected EJBs.
- Provide features to provide detailed information on the signature (return type, parameter types) of the selected method for a connection.
- Be able to generate necessary program code in Java for each connection method.
- Provide features to generate any new test driver classes as necessary.
- Be able to make changes to existing deployment descriptors, and/or be able to create new deployment descriptors.
- Provide run-time and development time information.

CHAPTER 7

The Solution: *EJBrew*

Introduction

The requirements specified in the chapter titled, Requirement: Enabling Application Composition for EJBs, must be met to develop EJB applications by composing them in an IDE.

The CORBA Component Model (CCM) specification has the features that meet the above requirements for a CORBA component. The CCM specification specifies Component Configuration and Assembly as part of the specification. Although, the Basic CORBA Component maps to the EJB specification, it is the Extended CORBA Component that would satisfy the requirement of enabling an EJB to be composed in an IDE.

Hence, the solution to meet the requirements is to apply the specification of Extended CORBA Component to EJB. This does not mean wrapping an EJB as a CORBA component, but camouflaging EJBs as CORBA components. This would require an EJB to implement interfaces necessary for it to be a CORBA Extended component along with the session and entity bean interfaces.

Solution

The approach is similar to making a regular/normal Java class into a JavaBean by just using design patterns for naming properties, methods, and events. Implementing the BeanInfo Interface of the JavaBean specification for a JavaBean (a separate class from the actual JavaBean implementation) provides additional information for IDE tools.

The solution for the functional requirements should be easy to use and should not impose too many burdens on the developers. These types of additional requirements are considered non-functional requirements, as they require the solution to be usable, and flexible. To meet both requirements, the suggested approach specifies marker interfaces for EJBHome, EJBRemote, and Entity and Session beans. The marker interfaces are all made part of the ***CcmEjb*** package and are called ***CcmEjb.Home***, ***CcmEjb.Remote***, and ***CcmEjb.Bean***.

The process of developing an EJB that can be used in an IDE for application composition has been named ***EJBrew***. This process makes a regular EJB into a ***CcmEjb***, which means that all the three basic classes and interfaces that make an EJB – EnterpriseBean implementation class, Remote and Home interfaces, are simply extended from their respective ***CcmEjb*** interfaces.

An EJB developed using the CcmEjb interfaces does not automatically enable an IDE to utilize the features of CcmEjb for EJB application composition. A CcmEjb interface-aware “introspection” engine, CcmEjbEngine, is required to introspect the EJBs and provide mediation between the CcmEjb-based EJB and the IDE. The CcmEjbEngine can be used as a standard plug-in to existing Java IDE tools. Current IDE tools provide some customization and extension mechanisms to allow third-party tools and modules to be plugged-in. As the extension mechanisms have not been standardized between IDEs, it is necessary to provide a CcmEjbEngine for each of the IDEs that would need the use of EJBrew.

This solution does not propose a completely new IDE that would meet the aforementioned requirements for EJB-based application composition. It only proposes a solution that would enhance the existing IDEs that already provide some capability to develop J2EE components such as EJBs, Servlets, and JSPs.

Figure 10. Proposed EJBrew Solution.

Specification of CORBA Componentized EJB

The EJBrew process requires that the two EJB implementation interfaces – Home, and Remote, and the implementation bean class – extending either the Session or Entity bean, have to indicate that they implement the appropriate CcmEjb interfaces.

For example, consider an existing session EJB with the following implementation classes and interfaces.

Bean Implementation Class: TraderBean.java

```
public class TraderBean implements SessionBean
{
    /* Standard Session EJB Implementation */
}
```

Remote Interface: Trader.java

```
public interface Trader extends EJBObject
{
    /* Standard Remote Interface Implementation */
}
```

Home Interface: TraderHome.java

```
public interface TraderHome extends EJBHome
{
    /* Standard Home Interface Implementation */
}
```

After the above EJB has gone through EJBrew, the implementation should be as follows:

Bean Implementation Class: TraderBean.java

```
public class TraderBean
    implements SessionBean, CcmEjb.Bean
{
    /* Standard Session EJB Implementation */
}
```

Remote Interface: Trader.java

```
public interface Trader extends EJBObject
    implements CcmEjb.Remote
{
    /* Standard Remote Interface Implementation */
}
```

Home Interface: TraderHome.java

```
public interface TraderHome extends EJBHome
    implements CcmEjb.Home
{
    /* Standard Home Interface Implementation */
}
```

Each of the CcmEjb interfaces is discussed in detail in the following sections.

The responsibilities of the developer with respect to the EJBrew process are also discussed.

The CcmEjb.Home Interface

The CcmEjb.Home interface defines its counter-part in the CCM specification, the *CCMHome* interface. Home interfaces provide factory like operations to create and destroy components.

The CcmEjb.Bean Interface and Sub Interfaces

EJB 1.1 specification defined two types of EJBs: entity and session. EJB 2.0 specification defined three types of EJBs: entity, session, and message-driven. For each type the EJB specification defines an interface: they are *EntityBean*, *SessionBean*, and *Message-DrivenBean*. All three interfaces are sub interfaces (derived) of the *EnterpriseBean* interface. The implementation class of each of these beans has to implement the respective interface based on the type of EJB.

The CCM defines equivalent interfaces for the *EnterpriseBean*, *SessionBean*, and *EntityBean*: they are *EnterpriseComponent*, *SessionComponent*, and *EntityComponent* respectively. As the CCM specification was mandated to support EJB 1.1 specification, there is no CCM equivalent for the EJB 2.0 specified *Message-DrivenBean*.

The CcmEjb also defines sub interfaces to correspond to each type of EJB. The additional methods and attributes defined in the CcmEjb sub interfaces define the corresponding CCM interfaces. The sub interfaces are CcmEjb.EntityBean, and CcmEjb.SessionBean. For the purpose of IDE-based EJB application composition, it is sufficient to only implement the base CcmEjb.Bean interface.

The CcmEjb.Bean is based on the CCM's EntityComponent interface, and it defines additional methods and attributes to expose an EJBs surface feature as "ports". This is discussed in detail in the CcmEjb.Remote interface discussion section.

The implementation class is where the bodies of all the methods of an EJB reside. Some of these methods are exposed by the remote interface to the clients. The methods that are not exposed remotely are usually the methods used internally by the EJB. As the CcmEjb.Remote interface defines some CcmEjb-specific methods to be exposed by the EJB, those methods need to be implemented by the EJB implementation class as well. The CcmEjb-specific methods that need to be implemented are specified by the CcmEjb.Bean interface. This implies that the EJB implementation class has to implement the CcmEjb.Bean interface along with the EJB type interface.

In order to reduce the burden of implementing additional methods for the CcmEjb.Bean interface, a helper implementation class is provided for the CcmEjb.Bean interface. This class is called CcmEjb.BeanImpl and the EJB implementation can extend

this class instead of implementing the CcmEjb.Bean interface. The helper implementation class needs to interact with the EJB implementation; this is done through object references.

The CcmEjb.Remote Interface

The remote interface is the client's view of an EJB. It represents all the remote operations available on an EJB. An EJB's remote interface is implemented by extending the EJBObject interface. The CCM equivalent of EJBObject is the CCMObject interface. The CcmEjb.Remote interface defines the CCMObject interface. In addition to the CCMObject interface, the CcmEjb.Remote interface also exposes EJBs surface features called by the CCM as "ports" of an extended CCM component. The ports of an extended CCM component are defined by the CCM specification. In brief, the "ports" of extended CCM components are:

- **Facets**, which are distinct named interfaces provided by the component for client interaction – CIDL keyword: provides
- **Receptacles**, which are named connection points that describe the component's ability to use a reference supplied by some external agent – CIDL keyword: uses
- **Event sources**, which are named connection points that emit events of a specified type to one or more interested event consumers, or to an event channel – CIDL keyword: emits, publishes
- **Event sinks**, which are named connection points into which events of a specified type may be pushed – CIDL keyword: consumes

- **Attributes**, which are named values, exposed through accessor and mutator operations. Attributes are primarily intended to be used for component configuration, although they may be used in a variety of other ways – CIDL keyword: attributes

Declarations for facets, receptacles, events sources, event sinks and attributes all map onto operations on the component's equivalent interface.

The Event source/sink ports may not be applicable to EJBs based on the EJB 1.1 specification as EJBs do not publish or receive events. But, EJB 2.0 specification added a new type of EJB called a MessageDrivenBean that can receive Java Messaging Specification based events.

The purpose of the CcmEjb.Remote interface is to present the details of the "ports" of an EJB. In order to do this the CcmEjb.Remote interface's methods must be able to look at the EJB's remote interface and possibly the implementations and determine all the different types of ports and gather their details. The CcmEjbEngine will provide a utility method to determine the "ports" for the CcmEjb.Remote interface.

Following three mechanisms are suggested for gathering details regarding the "ports" of an EJB. More than one mechanism is suggested in order to meet different

types of development scenarios. Some of the development scenarios covered by these mechanisms are:

- Development of new EJB by implementing the CcmEjb interface.
- Development of standard EJBs using UML-based modeling tools with extensions UML with CcmEjb tags (CcmEjb UML).
- Addition of a CcmEjb UML model to an existing EJB.
- Creating new EJBs, or modifying existing EJBs to conform to CcmEjb specified naming patterns for methods, events, and attributes.

In all four of the above development scenarios, the end results would produce an EJB that would provide its “ports” information in one of the following ways:

- From the accompanying CcmEjb UML model, or
- From the implementation of CcmEjb interface, or
- From the CcmEjb naming patterns

This specification does not restrict the developer to a single mechanism for the “ports” specification, but using more than one would be redundant. The CcmEjbEngine prioritizes the “ports” lookup/introspection in the following order: CcmEjb UML model, CcmEjb interface, and CcmEjb naming patterns. Each of these three mechanisms is defined in later sections.

UML Profile for EJB and CcmEjb UML

The Unified Modeling Language (UML) offers extensibility capabilities. Using the UML extensions mechanisms, the Java Community Process, a process by which new Java-related standards are requested and created, is currently working on a standard called *UML Profile for EJB* (Greenfield, 2000).

UML profile for EJB

The UML Profile for EJB defines a set of extensions to UML that can be used to model software implemented with EJB in UML. These extensions will let enterprise tool and framework vendors provide EJB modeling capabilities using UML, as well as forward and reverse engineering between UML models and EJB implementations.

In addition, the specification defines a mechanism for using UML models stored in an EJB-JAR to describe the contents of the EJB-JAR. This will let enterprise tool and framework vendors use UML models stored in EJB-JARs for automation and reflection.

The profile is based on EJB 1.1, and UML 1.3 standards. The current EJB standard is 2.0 and the UML standard 1.4 is currently being defined. The profile used UML to represent structure and semantics of EJBs. UML provides standard extension mechanisms for defining new model elements. Current UML-based modeling tools use

their own extensions to define UML for EJB modeling. The profile standardizes the UML extension use to interoperate tools between vendors.

Tools need to associate UML models describing EJBs with the EJB implementations in an EJB-JAR in order to use the models for automation and reflection. The EJB-JAR file is created with embedded UML information. Then that information can be used in tool building instead of using expensive introspection/reflection mechanisms.

This specification defines a set of standard UML extensions defining new model elements to represent EJBs. The following items express this definition:

UML Profile for EJB.

- Describes the relationship of logical and physical EJB constructs to UML model elements using the standard UML extension mechanisms.
- Describes the relationship between the EJB deployment descriptor and UML model elements using the standard UML extension mechanisms.
- Describes the forward engineering transformation from UML model elements using the standard UML extension mechanisms to EJB implementation artifacts.

- Describes the reverse engineering transformation from EJB implementation artifacts to UML model elements using the standard UML extension mechanisms.

In addition, this specification defines a mechanism for associating UML models stored in an EJB-JAR with the implementations of the EJBs they describe. The following items express this definition:

UML Descriptor for EJB.

Defines an XML DTD for a file placed within the EJB-JAR that identifies UML models stored in the same EJB-JAR and their relationship to EJBs in the EJB-JAR.

CcmEjb UML

The CcmEjb proposes to add tag names for the UML Profile for EJB so that the “ports” of an EJB can be marked as they are being modeled in UML tool with UML Profile for EJB support. The proposed tag name is *CcmEjb_UMLTag*, and the tag values are: *CcmEjbFacet*, *CcmEjbReceptacle*, *CcmEjbPublisher*, *CcmEjbConsumer*, and *CcmEjbAttribute*. These tags will be added to the definitions of methods and attributes of an EJB from within the UML modeling tool. Then CcmEjbEngine can inspect the UML model definition embedded in the EJB-JAR file and determine the methods and attributes that would meet the CcmEjb.Remote and CcmEjb.Bean interface requirements.

CcmEjb Interface Implementation

Instead of using CcmEjb UML to indicate the “ports” of an EJB, a developer can equivalently do the same by just choosing to implement the CcmEjb interfaces along with the standard EJB interfaces, or by simply extending the helper implementation classes for the CcmEjb interfaces.

The basic CcmEjb interfaces are CcmEjb.Bean, CcmEjb.Remote, and CcmEjb.Home. The optional sub interfaces are CcmEjb.EntityBean, and CcmEjb.SessionBean. A helper implementation class for each of the interfaces is provided, and it is named with an extension Impl to the corresponding interface name. For example, the helper implementation class for CcmEjb.Bean is CcmEjb.BeanImpl.

The CcmEjbEngine is capable of creating the minimal CcmEjb UML model by introspecting and applying reflection on the EJB that was implemented using the CcmEjb interfaces.

CcmEjb Naming Patterns

A developer not wanting to use the CcmEjb UML or to implement the CcmEjb interfaces can use the CcmEjb Naming Pattern to achieve the same results. This naming pattern is similar to the “design pattern” used by the JavaBeans specification.

In this method, all the EJB methods and attributes that are related to the “ports” of an EJB follow the CcmEjb Naming Patterns recommended naming scheme. The naming pattern general rule is that all the methods and attributes of an EJB related to “ports” are to have a prefix of CcmEjbXXX, where XXX is the keyword for type of the “port”. The type keywords as identified earlier are: CcmEjbFacet, CcmEjbReceptacle, CcmEjbPublisher, CcmEjbConsumer, and CcmEjbAttribute. The prefix values are the same values as the CcmEjb UML tag values.

When an EJB implementation JAR file does not have the CcmEjb UML model embedded, or the EJB was not implemented without implementing or extending CcmEjb interfaces or classes, then the *CcmEjbEngine* will use introspection and reflection along with the CcmEjb Naming Pattern rules to identify the “ports” of the EJB. From these results, the CcmEjbEngine is capable of creating the minimal CcmEjb UML model.

Definition of a CcmEjb-Composed Application on Package

The final form of a standard EJB is the EJB-JAR file that contains the Java class files, standard and application server specific deployment descriptor. In addition to these files, the EJBrew'able - EJB that is enabled for Application Composition in an IDE, should have additional metadata like information regarding its attributes and operations ("ports").

As the CcmEjb UML model contains the necessary information regarding the EJBrew'able EJB, that model in XML format will be added to the EJB-JAR file packaging. An XML DTD is being defined as part of the UML Profile for EJB standard; this definition will be utilized to define the CcmEjb UML definition added to the EJB-JAR file.

By having the CcmEjb UML model embedded in the EJB-JAR file, the application composition-enabled EJB would have its "ports" information along with it during the run-time as well as design-time when the EJB-JAR file is used within the IDE as the source of the EJB.

CHAPTER 8

Future Work

Introduction

In this chapter brief suggestions are made on the type of research, design, and implementation required for making the EJBrew process better – more complete, seamless, easy to use, and quick to respond within an IDE.

Currency of the Standards and IDE Tools

The identified problems, and shortcomings in the use of the current J2EE development tools (IDEs) for EJB based applications are based on the current state of the EJB specification, and the available tools. The proposed solution, EJBrew, is based primarily on the CCM specification that is part of the CORBA 3.0 that is yet to be officially released. Secondly the solution relies on the UML Profile for EJB specification that is currently being developed through the Java Community Process. As these standards are evolving, and more tools are also being announced, the solution must be implemented rapidly and must be designed to be flexible to support the evolving standards.

CIDL-Based CcmEjb Generation

In order to reduce the number of steps involved in the EJBrew process, and make the process seamless, the possibility of a tool to generate CcmEjb based EJB implementations directly from CIDL based definition file must be researched. This may require the need to extend the CIDL so that we can directly generate EJBs that are based on CcmEjb.

Integration with J2EE IDEs

The CcmEjbEngine module needs to be tightly integrated with J2EE IDEs and modelers Visual Café/WebGain, Rational Rose, JBuilder, and VisualAge. This task requires that the CcmEjbEngine be designed for flexibility and extensibility. Typically tools provide some sort of extension mechanisms through vendor-specific scripting. The integration would require vendor-specific integration modules that will work as mediators between the IDE and CcmEjbEngine.

Meeting Run-time Component Integration/Configuration Requirements

The design of the EJBrew process was primarily for design-time use. The solution suggested that the EJB-JAR packaging be embedded with the CcmEjb UML

model as well. Based on this embedded UML model's availability to J2EE application servers, an application structure can be dynamically modified and configured or be integrated with another application at run-time.

Run-time availability of the CcmEjb UML model depends on how the J2EE application servers load the EJBs, and if there are APIs that would enable an application programmer to load additional associated resources for the EJB.

CHAPTER 9

Conclusion

A proposal for a process and artifacts required for enabling Enterprise JavaBeans (EJB) for application composition using integrated development environment (IDE) has been introduced.

The scope of this thesis is to provide a flexible solution to the shortcomings in the current J2EE development tools that support EJB-based development. The primary shortcomings are:

- IDE tools have excellent support for stand-alone EJB (single component) development and deployment.
- IDE tools do not support application composition or “wiring” of EJBs together, or with other J2EE components such as JSPs or Servlets.
- IDE tools do not provide features to list available EJBs from files or from application servers.
- IDE tools do not support identifying EJBs based on their capabilities.
- IDE tools do not allow UML-based modeling to be used as part of the implementation, and they do not embed such information in the deployment.

The proposed solution for the above shortcomings is proposed here and it is called EJBrew. The solution is based on the soon-to-be-released Object Management Group's (OMG) CORBA 3.0 Component Model (CCM) and the currently-being-developed Java Community Process's UML Profile for EJB standards.

The CCM specification defines surface features of a component as "ports", and addresses application composition and component configuration. The UML Profile for EJB addresses standard extensions to UML for EJBs, to alleviate the problems created by specific UML extensions by individual tool vendors, and specifies XML DTD for the UML model to be embedded in the deployable EJB-JAR file.

The EJBrew process defines a development process and artifacts CcmEjbEngine, CcmEjb interfaces, related helper implementation classes, CcmEjb Naming Patterns, and CcmEjb UML tag names with pre-define tag values. The proposed process and the artifacts along with an extensible J2EE IDE tool would allow an EJB application to be composed by visual programmatic interconnections ("wiring").

The EJBrew process and the associated artifacts provide sufficient flexibility for the developer by providing alternate process steps and artifact use.

The proposed solution, EJBrew, adds the features needed for EJB-based J2EE application composition for the current IDE tools. The features of EJBrew enable

accelerated component-based EJB software development. It allows for the easy re-use of existing components, and common-off-the-shelf-components (COTS). The EJBrew is not a replacement for IDE tools; it simply supplements the existing tools with features for true component-based software development.

REFERENCES

- Arnold, K., & Gosling, J. (1998). The Java programming language (2nd. ed.). Reading, MA: Addison-Wesley.
- Carnegie Mellon Software Engineering Institute. (2000, July). Volume II: technical concepts of component-based software engineering. Retrieved November, 25, 2000, from the World Wide Web:
<http://www.sei.cmu.edu/publications/documents/00.reports/00tr008.html>
- Cobb, E. (2000, June 16). CORBA components: the industry's first multi-language component standard. Tutorial presented at the Object Management Group Meeting, Oslo, Norway.
- DeMichiel, L. G., Yalçinalp, L. U., & Krishnan, K. (2000, October 23). Enterprise JavaBeans specification (Version 2.0 – proposed final draft). Palo Alto, CA: Sun Microsystems Inc. Retrieved November 14, 2000, from the World Wide Web:
<http://java.sun.com/products/ejb/docs.html>
- D'Souza, D. F., & Wills, A. C. (1998). Objects, components, and frameworks with UML : The Catalysis approach. Reading, MA: Addison-Wesley.
- Greenfield, J. (2000). JSR #000026 - UML/EJB mapping specification. Palo Alto, CA: Sun Microsystems Inc. Retrieved January, 17, 2001, from the World Wide Web:
http://java.sun.com/aboutJava/communityprocess/jsr/jsr_026_uml.html

- Hamilton, G. (Ed.). (1997, July 24). JavaBeans specification (Version 1.01). Palo Alto, CA: Sun Microsystems. Retrieved December 23, 1999, from the World Wide Web: <http://java.sun.com/products/javabeans2/docs/spec.html>
- Heiss, J. J. (2000, November 16). Matchmaker for enterprise JavaBeans components. Palo Alto, CA: Sun Microsystems Inc. Retrieved December 27, 2000, from the World Wide Web: <http://java.sun.com/features/2000/11/compsource.html>
- Kapur, G. K. (n.d.). IT resource shortage – The artificial crisis. West Yarmouth, MA: The Standish Group. Retrieved January 22, 2001, from the World Wide Web: http://www.pm2go.com/profession/resourcemnmt/itresource_pl.html
- Object Management Group (OMG). (1999, August 2). CORBA components - volume 1. OMG TC Document orbos/99-07-01.
- Orfali, R., Harkey, D., & Edwards, J. (1998). Client/server programming with Java and CORBA (2nd. ed.). New York: Wiley.
- Matena, V. & Hapner, M. (1999, December 17). Enterprise JavaBeans specification (Version 1.1). Palo Alto, CA: Sun Microsystems. Retrieved January 10, 2000, from the World Wide Web: <http://java.sun.com/products/ejb/docs.html>
- Patterson, S. (2000, June). Open-market component-based development with enterprise JavaBeans technology. Palo Alto, CA: Sun Microsystems Inc. Retrieved December 10, 2000, from the World Wide Web: <http://java.sun.com/features/2000/06/componentsource.html>

- Pharoah, A., Siegel, J., & Brooke, C. (2000). Creating commercial components: CORBA component model. Retrieved January 5, 2001, from the World Wide Web: <http://componentsource.com/build/corbawhitepaper.asp>
- Project Management Resources. (n.d.). Unsuccessful information technology projects. Retrieved February 6, 2001, from the World Wide Web Page: <http://pmri.home.mindspring.com/itpro.htm>
- Roman, E. (1999). Mastering EJB and the Java 2 platform, enterprise edition. New York: John Wiley and Sons.
- Sessions, R. (1997). COM and DCOM: Microsoft's vision for distributed objects. New York: John Wiley & Sons.
- Sun Microsystems Inc. (2000, March 20). Java 2 platform, enterprise edition blueprints. Retrieved July 10, 2000, from the World Wide Web: <http://java.sun.com/j2ee/blueprints/apmTOC.html>
- Sun Microsystems Inc. (2000, May 23). Java 2 SDK, enterprise edition, (Version 1.2.1). Retrieved July 15, 2000, from the World Wide Web: <http://java.sun.com/j2ee/download.html>
- Sun Microsystems Inc. (n.d.). Industry momentum: enterprise JavaBeans supporters. Palo Alto, CA. Retrieved January 18, 2001, from the World Wide Web: <http://java.sun.com/products/ejb/tools1.html>
- Szyperski, C. (1998). Component software: Beyond object-oriented programming. Reading, MA: Addison-Wesley.

- Thomas, A. (1998). Enterprise JavaBeans technology: server component model for the Java platform. Boston: Patricia Seybold Group. Retrieved December 22, 1999, from the World Wide Web:
http://java.sun.com/products/ejb/white/white_paper.html
- Ward-Dutton, N. (2000, January). Containers: A sign components are growing up. Retrieved April 7, 2000, from the World Wide Web:
http://www.adtmag.com/Pub/jan2000/com_strat.shtml
- Zona Research. (1999). Enterprise JavaBeans: A business benefits analysis. Redwood City, CA: Zona Research. Retrieved March 10, 2000, from the World Wide Web: <http://java.sun.com/products/ejb/pdf/zona.pdf>