Integrating CORBA & EJB

Seamless Integration and Interoperability between CORBA & EJB with the Inprise Application Server

January 2001

William Edwards & Salil Deshpande
The Middleware Company

The Middleware Company
PO Box 80049 Austin, TX
78708-0049
Austin, TX USA
1 (877) 866-5282
http://www.middleware-company.com

1 Table of Contents

2 Overview

This paper explains in detail the relationship between Enterprise Java Beans (EJB) and CORBA. It discusses in depth why it is important for EJB-compliant application servers to be based on CORBA, spells out the benefit of such an approach to developers and end-users, and discusses scenarios which address frequently asked questions about using CORBA and EJB together.

Detailed examples of CORBA-EJB interoperability using Inprise Application Server (IAS) are provided.

3 Distributed Computing Infrastructures

In the mid-nineties, companies began to build distributed computing infrastructures using "Remote Procedure Call" infrastructures such as Sun's Open Network Computing (ONC) and OSF's DCE. In many cases, companies built their own in-house infrastructure necessary to support this architecture.

In the 1990s, a new distributed computing model became widely adopted, built around the concept of "Distributed Object Computing." Virtually every enterprise IT organization is either prototyping or has deployed distributed object applications.

In this area, two architectures emerged for building distributed object applications. One – COM/DCOM or what is now called COM+ -- is Microsoft's proprietary distributed object technology for the Windows platform. It was often adopted for departmental, Windows-only environments, and is supported by Inprise's Windows development tools.

However, because the enterprise is still a heterogeneous environment, there was a huge rise in interest throughout the 1990s in CORBA as a distributed object infrastructure for heterogeneous enterprise computing. CORBA is an open standard, developed by the Object Management Group with now over 850 members. Practically every enterprise customer is currently using CORBA in one form or another.

4 CORBA: Any Language, Any Platform, Any Vendor

This section reviews the main benefits of CORBA –subsequent sections will discuss how these benefits apply to EJB containers and EJB-based applications.

4.1 CORBA provides Interoperability across programming languages

Interfaces to CORBA objects are written in a programming-language-independent notation called IDL (Interface Definition Language). These language-independent interfaces are then *mapped* to language-specific interfaces, according to rules standardized by the OMG, known as *language mappings*. Standardized mappings exist for Java, C, C++, COBOL, Ada, and Smalltalk. De-facto standard mappings exist for other popular languages such as TCL, Perl, Delphi (Object Pascal), and Python.

As a result, CORBA objects and servers, and their clients, can be written in any language. It is not abnormal for a CORBA-based distributed application to consist of pieces that are written in completely different languages.

The ability to painlessly make cross-language calls is extremely important because the amount of IT budget allocated for maintaining and integrating non-Java production systems is, and will remain for the foreseeable future, many times larger than that spent on new development in Java or in any other new language.

4.2 CORBA provides Interoperability across platforms (hardware + operating systems)

The protocol that CORBA uses to allow distributed objects to communicate with each other, is specified by the OMG, and is known as IIOP (more generally, GIOP).

Just as TCP/IP stacks from multiple vendors can seamlessly interoperate with each other without explicit cooperation with each other (because TCP/IP is a well-specified standard), CORBA implementations from multiple vendors can also interoperate with each other.

Different CORBA products can be used on different platforms. Furthermore, most popular ORB products have themselves been ported to multiple hardware platforms and operating systems. For example, the VisiBroker products are available on a number of platforms – Windows, Solaris, Linux, HP-UX, SGI, IBM's AIX, Digital Unix, IBM OS/390, and others. The availability of all-Java ORBs such as VisiBroker for Java, allows CORBA applications to be deployed anywhere Java runs.

4.3 CORBA provides Independence from CORBA vendors

Standardized APIs (IDL, language mappings, etc.) and the standardized wire protocol (IIOP) together provide developers and their customers with independence from the CORBA products and vendors.

The availability of interoperable CORBA ORBs from a number of different vendors means that developers do not depend on any single vendor so long as they do not use vendor-specific add-on features.

4.4 CORBA provides source portability

When the CORBA standards were initially released, source portability (the ability to recompile and run existing CORBA applications with a different CORBA product, without making changes) was promised, but not delivered.

Source portability for pure-clients was acceptable. With some care, one could write CORBA clients that would port across CORBA products.

Source portability for servers was not acceptable, however. Among the culprits was a key server-side API (known as the BOA), which was heavily under-specified and ambiguous in many places. This led CORBA vendors to make incompatible assumptions and extensions, sacrificing server-side source portability.

A new server-side specification, known as the Portable Object Adapter (POA), corrects this problem. In addition to being complete and unambiguous, it contains many more features than the old BOA specification. The POA is the BOA done right, and reinforces the fact that CORBA is the ideal substrate for all distributed applications.

4.5 CORBA provides a battle-hardened high-performance object messaging protocol

Interoperability is not the only benefit of CORBA's wire protocol, IIOP.

CORBA systems using IIOP have been deployed for some time in real enterprise systems. CORBA and IIOP have undergone considerable revision and evolution since its invention, based on experience obtained in real system development and deployment. It is, so to speak, battle-hardened. By standardizing on IIOP, EJB eliminates the risks and cost of developing another new distributed object protocol – more on this shortly.

5 Java, CORBA, RMI, and the Grand Unification

Java has quickly become the language of choice for writing applications. Doubts linger in some people's minds whether Java is reliable enough for mission-critical applications and can perform well on the server-side but compelling evidence from demonstrations, prototypes, pilots and success stories are eliminating those doubts rapidly.

5.1 Java & CORBA complement each other heavily

When Java first met CORBA, in the form of *VisiBroker for Java* (originally PostModern's *BlackWidow*, the first Java ORB), the outcome proved advantageous for both communities. The OMG/CORBA community pounced on Java as a much cleaner, more tractable, and ultimately, more cost-effective way to build many CORBA applications than the only other viable option at the time—C++. For Java programmers, it offered a robust distributed object infrastructure that leveraged battle tested investments in CORBA technology, as well as a clean way to exploit non-Java legacy assets.

5.2 Sun's decision to include non-CORBA-compatible middleware in the JDK confused the market

Simultaneously however, RMI (Remote Method Invocation) was emerging as a way to build distributed object systems in pure, native Java. RMI was Java-specific distributed object infrastructure pioneered by a renegade group of Sun engineers led by the opinionated Jim Waldo who had decided at that time that compatibility with CORBA was not important; the group managed to get it included into Sun's standard JDK1.1 distribution. Due to the popularity of Java, this move suddenly made RMI a more popular distributed object infrastructure than CORBA.

Architecturally, RMI is very similar to CORBA. It makes remote invocations on objects through established interfaces, using automatically generated proxies and skeletons to manage the marshaling and communication transparently. In many details, RMI is quite different. It uses Java's native type system to describe remote interfaces and their parameters, whereas CORBA uses a programming-language independent IDL (Interface Definition Language). A Java/RMI programmer identifies an interface that is intended for remote access by defining the interface as an extension of the appropriate remote base interface. Remote methods can take parameters of virtually any Java class (specifically, it must be serializable). In contrast, CORBA operations had at that time a much more limited palette of data types. Since it was inherently cross-language, any CORBA-defined data type must have an equivalent representation in a variety of programming languages (including, C++ and COBOL).

However, at the time, RMI had its own completely separate infrastructure, sharing essentially nothing with CORBA infrastructure. It had its own messaging protocol (JRMP), its own registry mechanism, and was developing its own set of services that were similar to, but not compatible with, existing CORBA services.

5.3 End-users forced the grand unification of CORBA & RMI

Corporate IT organizations weren't particularly enthusiastic about being forced to maintain and support two gratuitously different infrastructures. A clear (and harshly worded) message was delivered to both communities: *we want to program in Java, using both CORBA and RMI programming models where appropriate, but we don't want two infrastructures. Make that happen!*

The result became known as "**RMI over IIOP**". Understanding what this really means hinges on understanding the relationship between distributed programming models and their underlying protocols, and is explained more fully in a separate white paper from Inprise entitled *RMI, IIOP, & EJB.*

So, the question "CORBA vs. RMI" has become moot. Due to the following developments, CORBA and RMI are becoming one and the same:

- Extension of the IIOP protocol to accommodate RMI
- Availability of implementations of RMI-over-IIOP
- CORBA's new ability of passing objects by value (a strength of RMI)
- Specifications for reverse-mapping RMI-style interfaces to CORBA IDL interfaces

RMI has effectively become a Java-specific API for CORBA!

6 Server-side Components: EJB & CCM

6.1 EJB is a server component model

Enterprise Java Beans is to Server Components, what JavaBeans is to Components. That's where the similarity ends, though. Although they are both specifications for component models, JavaBeans addresses issues of application assembly in builder tools, while EJB describes a server framework into which server components can be deployed.

Also, do not make the mistake of thinking that JavaBeans is for client-side development and EJB is for server-side development. JavaBeans is perfectly appropriate for building non-visual server-side components. It is EJB, however, that provides the framework, the APIs and hooks, and advanced services that most server components need.

6.2 Why do we need a server component model?

A server component model is useful because developing server components is difficult – most developers agree that it is much more difficult than developing client or GUI components.

Developing server components is difficult because in addition to writing the application logic, developers must account for numerous server-side issues: threading & concurrency, access to data (usually stored in relational databases), access to legacy systems, security, authorization, server process lifecycle, object lifecycle (creation, initialization, removal of objects), transactions (including distributed and two-phase-commit transactions), integration with naming services, and resource management (judicious use of objects, memory, threads, connections, etc.).

Without a server component model, a server developer is forced to play two roles: that of an application programmer, and that of a systems programmer. A server component model allows the developer to focus only on the application logic.

EJB nicely separates the roles of the *bean developer* (developer of components containing application logic), *application assembler*, *deployer*, and *administrator*. For more information please read the EJB specifications or one of the several white papers or tutorials at http://java.sun.com/ that provide an overview of the specification.

6.3 What's a component?

The word is overused. For our purposes, think of a component as a piece of code that follows certain rules and conforms to certain conventions, which allows it to exist in a certain execution environment, and easily take advantage of services offered by that execution environment.

6.4 How does EJB work?

Interfaces to server components are written in Java. These are in fact, "RMI-style" interfaces. Support for several design patterns are built into the framework. Developers concentrate on writing Java classes that encapsulate application logic; IDEs will often generate the remainder of the code (including special interfaces like the Home and the Remote) required by the framework. Special compilers will generate support for invoking EJBs remotely.

6.5 What is the CORBA Component Model (CCM)?

EJB allows the development of components in only one language, Java. The CORBA Component Model generalizes EJB to make it programming-language-independent. Like EJB, support for several design patterns is built-in. Developers concentrate on writing business logic, in any language for which there is a *language mapping* from IDL. Instead of writing interfaces to server components in Java, they are written in IDL.

The CCM standardization was very recently completed by the OMG, and there are currently no products in the market that support CCM. Expect this to change, but slowly. Java and EJB have much more momentum, by far, and those standards are likely to be much better supported than the CCM in the near future.

7 CORBA vs. EJB?

7.1 Should CORBA fans care about EJB?

If you're an application developer planning to write applications with CORBA, should you care about EJB? The short answer is yes. Assuming you can write your server components in Java, writing to EJB APIs can make your job dramatically easier, because you're programming at a dramatically "higher level". The execution environment (the EJB container) can automatically integrate your application component with naming services, transaction services, and security services, to name just three. Application developers need not write any database code if they don't need to – data can be moved from databases to instance variables of your application components transparently. And so on.

7.2 Should EJB fans care about CORBA?

If you're an application developer planning to write applications with EJB, should you care about CORBA/IIOP? The short answer again is yes. You may not need to care a lot, but you need to care. Support for CORBA/IIOP is mandated by the EJB specifications for interoperability across multiple languages, and interoperability among EJB containers from multiple vendors, among other things.

8 Inprise Application Server: CORBA & EJB

It is possible to build an EJB container from the ground up, including a custom implementation of RMI over IIOP. Although possible, it is not particularly advisable. It may not be obvious to the casual observer, but the most effective way to build a serious EJB run-time system is to start with a high-quality, up-to-date (i.e., version 2.3) Java ORB. The EJB container interfaces can be implemented as a very thin API layer on top of CORBA. The CORBA POA, for example, provides all of the capability needed to implement the required Bean life cycle policies and transactional policies. In fact, the authors of the EJB specifications anticipated that the EJB APIs would be layered atop the CORBA POA APIs, although the specifications do not explicitly require this.

Moreover, if the underlying ORB supports the CORBA Object Transaction Service, then the transactional requirements for EJB are already met. EJB specifies the use of JTS for distributed transaction support. Though it is not commonly understood, JTS is simply the Java mapping for the IDL interfaces of CORBA OTS.

9 OTS & JTS: Distributed Transactions in the Inprise Application Server

As mentioned before, it is not commonly understood that JTS is simply the Java mapping for the IDL interfaces of CORBA OTS. In addition to the JTS, Sun has specified another API, the Java Transaction API (JTA) which some consider an "ease of use" layer for the JTS.

From 1996 to 1998, Inprise Corporation spent numerous man-hours in developing the Integrated Transaction Service (ITS), a reliable and robust ground-up implementation of the CORBA OTS. ITS was therefore the first implementation of JTS. Even Sun Microsystems chose to license Inprise's ITS as their JTS implementation.

9.1 Two-phase Commit (2PC)

Among the most important features of Inprise's ITS, is the ability to perform two-phase-commit (2PC) transactions. Support for 2PC allows transactions to span multiple heterogeneous transactional resources. For example, a single transaction could span multiple Oracle databases, Sybase databases, Informix databases, MQ Series, CICS, & IMS. ITS uses an industry-standard protocol, called XA, for communication between the transaction coordinator and resource managers. Resource managers allow transactional resources (such as an Oracle database) to participate in 2PC transactions, and are provided by the vendors of the resources.

The EJB specification requires that the EJB container provide support for JTA. Some EJB application vendors provide minimal support for JTA, and do not provide support for 2PC. These products may be chosen because development groups may not consider 2PC an important feature; however, it may become an extremely important feature at deployment time.

Your container must support 2PC transactions if:

- Your transactions must span EJBs that use multiple or different kinds of databases
- You may have only a single database, but your transactions must span EJBs that are distributed across multiple containers
- You want to involve legacy systems, other transactional middleware, or queuing systems in your transactions
- You have none of the above needs right now, but want to allow for any of the above in the future, without making code changes

None of these needs may be apparent at development time, but at deployment time, the need for increased performance, repartitioning, load-balancing, and failover, can create the need for distributed (2PC) transactions.

10 Introduction to EJB-CORBA Interoperability

The ability for CORBA clients and servers to successfully interoperate with Enterprise Java Beans (EJBs) is one of the overall goals of the EJB architecture. The realization of this goal enables EJBs to be deployed in large heterogeneous enterprise systems without placing unnecessary restrictions on the system architects.

The ability of CORBA components to successfully interoperate with EJB components is based on the unification of RMI and CORBA. Six main areas allow this unification:

10.1 Wire Protocol (IIOP)

The Inprise Application Server 4 (IAS4), which is based on the CORBA/IIOP industry standard, ensures interoperability between CORBA based implementations and EJBs in heterogeneous environments.

In order for components in a distributed system to successfully communicate, some common protocol is required. IIOP is the de-facto industry standard wire protocol for such distributed systems. The use of IIOP as the EJB on the wire protocol is not a requirement for EJB 1.1 compliance; however, IIOP is required for interoperability among application servers from multiple vendors. A later release of the Java 2 Platform, Enterprise Edition (J2EE) is likely to explicitly require that a J2EE platform vendor implement IIOP. The Inprise Application Server uses IIOP as its communication protocol today!

10.2 CORBA Objects-by-Value

In order to allow EJB clients and servers to use IIOP as the messaging protocol, IIOP supports the semantics of RMI, primarily the ability to pass objects by value. Version 2.3 of the CORBA specification extends the basic CORBA typing system to include the ability to define value types (i.e., local programming objects) in IDL and pass them by value as parameters in requests. Value types were designed to be able to express Java objects, but are not limited to that. Value types are language independent.

In conjunction with VisiBroker 4, Inprise's CORBA 2.3 implementation, Inprise Application Server fully supports CORBA objects by value. This implementation allows all possible types that an EJB may use to be passed from a CORBA client or to a CORBA server. An overview of some of the important issues that arise from this unification is provided in Appendix 1.

10.4 Java-to-IDL Mapping

In addition to the Object Management Groups (OMG) standard mapping from CORBA Interface Definition Language (IDL) to Java, a mapping from Java to IDL has been defined. This allows the Java types used by EJBs to be mapped in a standard manner to CORBA IDL.

10.5 Mapping of Naming Services

This mapping defines how CORBA's COS Naming service can be used to locate EJB Home objects.

The Inprise Application Server uses a Java Naming and Directory Interface (JNDI) implementation layered on top of the CORBA Interoperable Naming Service. This scheme allows EJB/CORBA clients to access the Naming Service by using the JNDI API as required by the EJB specifications; plain CORBA clients access the same Naming Service by using the API defined by the CORBA Interoperable Naming Service specification.

10.6 Mapping of Transaction Services

This mapping defines how EJB transaction features map to the CORBA Object Transaction Service.

The Inprise Application Server EJB runtime uses an implementation of the CORBA Object Transaction Service (OTS) for transaction support. For enterprise beans that execute within the scope of a transaction started by the client, the transaction context is implicitly propagated to the application server if needed, i.e. if any of the EJBs methods use the "Supports", "Required" or "Mandatory" transaction policy.

10.7 Mapping of Security Services

This mapping defines how security features in EJB map to CORBA security.

The main security concern for EJBs is access control, which requires the EJB server to determine the client's identity. The Inprise Application Server uses a CORBA Security Service implementation, which utilizes IIOP-over-SSL as a secure transport layer. The client's identity is the X.500 distinguished name of the subject obtained from the X.509 certificate during SSL client authentication.

11 Interoperability Case-by-Case: A Detailed Introduction

The remainder of this paper provides a flavor of how to use the CORBA-EJB interoperability features provided by an EJB application server based on CORBA middleware, such as IAS4.

In the sections that follow, we'll discuss the various cases that arise when interoperating with Enterprise Java Beans using CORBA.

Figure 1 shows all the relevant cases.

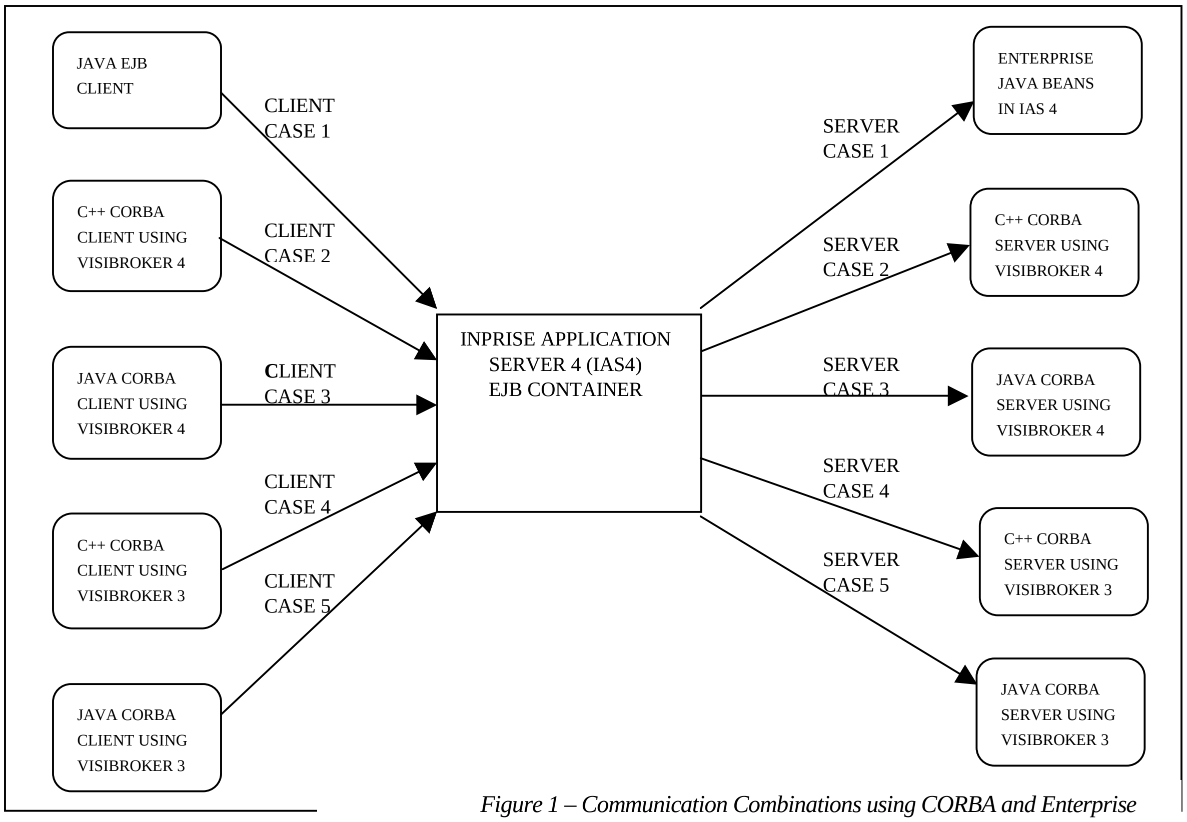
JAVA EJB CLIENT
C++ CORBA CLIENT USING VISIBROKER 4
JAVA CORBA CLIENT USING VISIBROKER 4
C++ CORBA CLIENT USING VISIBROKER 3
JAVA CORBA CLIENT USING VISIBROKER 3
CLIENT CASE 1
CLIENT CASE 2
CLIENT CASE 3
CLIENT CASE 4
CLIENT CASE 5
INPRISE APPLICATION SERVER 4 (IAS4) EJB CONTAINER
SERVER CASE 1
SERVER CASE 2
SERVER CASE 3
SERVER CASE 4
SERVER CASE 5
ENTERPRISE JAVA BEANS IN IAS 4
C++ CORBA SERVER USING VISIBROKER 4
JAVA CORBA SERVER USING VISIBROKER 4
C++ CORBA SERVER USING VISIBROKER 3
JAVA CORBA SERVER USING VISIBROKER 3

Figure 1 – Communication Combinations using CORBA and Enterprise

11.1 Overview of the Client Cases

11.1.1 Client Case 1 (The standard case, which most application servers support)

Client Case 1 is the case of a client written in Java, and using EJB APIs to invoke methods on EJBs being hosted in IAS4. This involves:

- Using JNDI to look up EJB objects
- Using the Home and Remote interfaces of EJBs directly

This is the standard case, and the simplest, from the perspective of both users, and application server vendors. **All EJB application servers support this client case – most support only this one**.

11.1.2 Client Case 2 (A crucial case, which most application servers do not support)

Client Case 2 is the case of a client written in C++, and using CORBA APIs to invoke methods on EJBs hosted in IAS4. This involves:

- Using the C++ mapping of the CORBA Naming Service IDL to look up EJB objects
- Mapping the Java interfaces of the EJBs to CORBA IDL (using Inprise's *java2idl*)
- Mapping these CORBA IDL interfaces to C++ (using Inprise's *idl2cpp*)
- Using this C++ mapping of the IDL interfaces of EJB objects to invoke methods on EJB objects
- Potentially providing C++ definitions for any Java complex types that you expect to pass across C++ and Java (much more on this later in this paper)

Of all the client cases, this is potentially the most interesting and complex case. **Most application servers do not support this case**, for one or more of the following reasons:

- The application server is not built atop CORBA middleware and therefore do not support CORBA/IIOP clients written in any language, including C++
- The application server does use IIOP as the wire protocol, but the application server vendor does not conform to the CORBA 2.3 objects-by-value specification, which makes it impossible to pass complex types by value across the Java-C++ boundary.
- The application server vendor claims to use CORBA/IIOP middleware with a CORBA 2.3 objects-by-value compliant implementation, but has not conducted the remainder of the engineering required to allow C++ clients to invoke EJBs seamlessly. Symptoms to the end user are bugs, missing tools, limitations on what can be expressed in EJB interfaces, and performance problems.

11.1.3 Client Case 3 (The mostly ignored case)

This case is similar to Client Case 1 in that the client is written in Java. The difference, however, is that in Client Case 1, the Java client is an "EJB client" (i.e., it uses EJB APIs, as previously described). In Client Case 3, the Java Client uses CORBA APIs. This involves:

- Using the Java mapping of the CORBA Naming Service IDL to look up EJB objects
- Mapping the Java interfaces of the EJBs to CORBA IDL (using Inprise's *java2idl*)
- Mapping these CORBA IDL interfaces to Java (using Inprise's *idl2java*)
- Using this Java mapping of the IDL interfaces of EJB objects to invoke methods on EJB objects

The list above may surprise the casual reader -- it may not be obvious why one would want to map the interface of an EJB (which is always in Java) to IDL, and then map that IDL back to Java. Why, if a client were indeed written in Java, would it need to use CORBA APIs to invoke EJBs instead of the more straightforward EJB APIs?

The answer is that support for this case is required to support both new and existing CORBA applications. Recall from the discussion earlier in this paper that one of the main reasons organizations use CORBA to build applications is its support for multiple languages. Building a particular portion of a CORBA application (be it a client or server object) in Java does prevent the organization from easily rewriting that component in another language later. The "look and feel" of CORBA APIs from that other language will be directly analogous to the look and feel of CORBA APIs from Java. It can be argued that forcing such a component use Java-specific APIs, such as the EJB APIs, reduces or eliminates the programming-language independence that CORBA has been providing to these organizations.

Even if programming-language-independence is not the main concern, the Java client in Client Case 3 might be part of a huge (when compared to new EJB-based development) existing/legacy CORBA-centric application. To avoid having to train existing CORBA developers in EJB development, or to avoid making major changes to existing CORBA applications, avoid introducing new API sets, and in general, to control software complexity, it might make sense to present CORBA applications with Java-CORBA APIs to EJBs, rather than new EJB APIs.

To put things in perspective, however, the sentiment of the software engineering community regarding Java is extremely positive; it is reminiscent of the 1980s where C was the darling of the community, until the late 1980s and early 1990s, when it was replaced by C++. For this reason, all cases other than Client Case 1 and especially potentially confusing cases such as Client Case 3 are sometimes ignored by the community more often than they should be.

11.1.4 Client Cases 4 and 5 (Dealing with Older CORBA Implementations)

These cases are almost identical to Client Cases 2 and 3 respectively, with one big difference… VisiBroker 3.x, like other older implementations of the CORBA standard, predates CORBA 2.3, and therefore does not contain support for the objects-by-value specifications. This influences one's approach toward CORBA-EJB interoperability, as discussed below.

11.2 Overview of the Server Cases

11.2.1 Server Case 1 (EJBs invoking other EJBs in a different container)

This case is the server-side version of Client Case 1. This case occurs when one EJB must invoke methods on another EJB object, potentially in another EJB container, to accomplish what its client is requesting. It may involve one session bean invoking another session bean, a session invoking an entity bean, or an entity bean invoking another entity bean.

This requires the first EJB to take the following steps:

- Use JNDI to look up EJB objects
- Use the Home and Remote interfaces of EJBs directly

From a simplistic viewpoint, this case is identical to Client Case 1,with the client itself also being an EJB.

There is one additional consideration: transaction and security contexts that have been propagated from the client to the first EJB must be propagated transparently to the second EJB, and onward. In fact, all the server cases share this consideration; we will revisit this issue after the overview of server cases.

11.2.2 Server Case 2 (EJBs invoking CORBA Objects written in C++)

In this case, an EJB deployed within an EJB container, must invoke methods on CORBA objects, deployed within a C++ executable, to accomplish what its client is requesting.

We can assume that the CORBA objects have IDL interfaces.

This requires the EJB to take the following steps:

- Use the Java mapping of the CORBA Naming Service IDL to look up EJB objects
- Map the CORBA objects' IDL interfaces to Java (using Inprise's *idl2java*)
- Use this Java mapping of the IDL interfaces of EJB objects to invoke methods on EJB objects
- Potentially provide Java definitions for any value types being used by the C++ CORBA object that you expect to pass across C++ and Java (much more on this later in this paper)

The issue raised by the last item of the above list is similar to the one that arises in Client Case 2. (Compare the last item in the above list to the last item in the list in Client Case 2).

11.2.3 Server Case 3 (EJBs invoking CORBA Objects written in Java)

Like Server Case 2, EJBs deployed within an EJB container, must invoke methods on CORBA objects, except that here the CORBA objects have been written in Java.

As always, we can assume that the CORBA objects have IDL interfaces. Moreover, this case is simpler than Server Case 2, because both sides are Java, and it is unlikely that we would have to provide Java definitions for any value types being used by IDL interfaces.

Our EJB is thus required to do only the following:

- Use the Java mapping of the CORBA Naming Service IDL to look up EJB objects
- Map the CORBA objects' IDL interfaces to Java (using Inprise's *idl2java*)
- Use this Java mapping of the IDL interfaces of EJB objects to invoke methods on EJB objects

Note that this list is identical to the list in Server Case 2, except that the last item in Server Case 2 becomes unnecessary here.

11.2.4 Server Cases 4 and 5 (EJBs invoking CORBA objects using older CORBA products)

These cases are almost identical to Server Cases 2 and 3 respectively, and in fact simpler.

Recall that VisiBroker 3.x, like other older implementations of the CORBA standard, predates CORBA 2.3, and therefore does not contain support for the objects-by-value specifications. It is therefore impossible that server programmers using older CORBA products would use the objects-by-value feature. That issue, which sometimes complicates the other cases, thus never arises in Server Cases 4 and 5.

11.3 Another Reminder about Propagation of Transaction and Security Contexts

Note that in all of the cases shown in Figure 1, clients and servers are free to participate in distributed transactions. For example, your CORBA clients could begin transactions and these will propagate to EJBs in the Inprise Application Server. Similarly, your clients are able to use security, (e.g. use client certificates, make secure invocations over SSL connections, etc.), and the Inprise Application Server will perform access-control checks, propagate security information, etc.

Furthermore, if the EJB invokes other objects, these transaction and security contexts are transparently propagated to the other objects, whether they are other EJBs or CORBA objects (written in Java, C++, or any other language).

This is not true of all application servers. For example, even in EJB application servers that do not support IIOP at all, it is still possible to construct Server Cases 1 through 5… but because they do not support CORBA/IIOP, they cannot propagate security and transaction contexts in those cases.

12 Interoperability Case-by-Case: Making It Work

12.1 Client Case 1 – EJB/Java CORBA Clients (CORBA 2.3)

When an EJB is deployed, IAS4 can produce EJB client stubs, in the form of a .jar file containing .class files. A client stub generation utility is also provided with IAS4, should this step need to be repeated later.

It is straightforward for a Java EJB client to use these stubs. There is never a need to first produce CORBA IDL and then produce stubs from this IDL.

As mentioned earlier, these clients can use JNDI to obtain a handle to the Enterprise Java Bean Home object and make invocations on the Home to create EJBs. Java EJB clients written in this way have no restrictions placed upon them with regard to the objects they may pass as method parameters or return types. They can use Java primitive types, Java native complex types and user defined complex types in exactly the same way as the EJB itself.

12.2 Client Case 2 – C++ Clients using VisiBroker 4 (CORBA 2.3)

In this case, three different strategies can be employed when writing a C++ client for the Enterprise Java Bean. Briefly these are:

1) **The C++ client communicating with a Java CORBA Object that in turn communicates with the EJB**
2) **Taking care to write EJB methods that take or return only native CORBA data types (since it can be difficult to translate complex Java types to C++)**
3) **Writing EJB methods without special consideration for CORBA clients, and implementing complex Java data types in C++ as required**

12.2.1 Why more than one strategy?

The most general strategy is Strategy (3). However, strategies (1) and (2) are presented as alternatives because strategy (3) directly leads us to using the CORBA objects-by-value feature, which is non-trivial to use. It is described in detail in the discussion on strategy (3).

12.2.2 Strategy 1- Using a Co-Located CORBA Wrapper

In IAS4, it is possible to collocate CORBA Objects along with EJBs in the EJB container. In other words, CORBA objects can be deployed in the same Java Virtual Machine (JVM) that is being used by the EJB container. **This is not possible in most application server products**.

This feature makes it possible to write a CORBA object whose only purpose is to make an EJB's interface easier to use by CORBA clients. We call such a CORBA object, a *CORBA wrapper* for the EJB. The collocation eliminates any performance penalty that wrapping EJBs with CORBA interfaces would create in other application servers.

A C++ client can connect to this CORBA wrapper, which in turn makes invocations on the Enterprise Java Bean. The purpose of this exercise is to allow the CORBA Object to act as a bridge between the IDL data types used by the C++ client and the native Java data types used by the EJB.

For example, suppose that we write an EJB method that takes a *java.util.ArrayList* as a parameter and returns a *java.util.HashMap*. The author of the CORBA wrapper may decide that the ArrayList is most conveniently exposed to CORBA programmers as an IDL sequence of strings, and that the HashMap is best exposed as an IDL sequence of structs. (The actual mapping, which would use the objects-by-value feature, would be much more complicated).

The CORBA wrapper receives the sequence of strings, converts it to an ArrayList, and makes the invocation on the EJB. After the EJB method returns the HashMap, the CORBA wrapper converts this to a sequence of structures and returns it to the C++ client. Figure 2 illustrates this strategy.

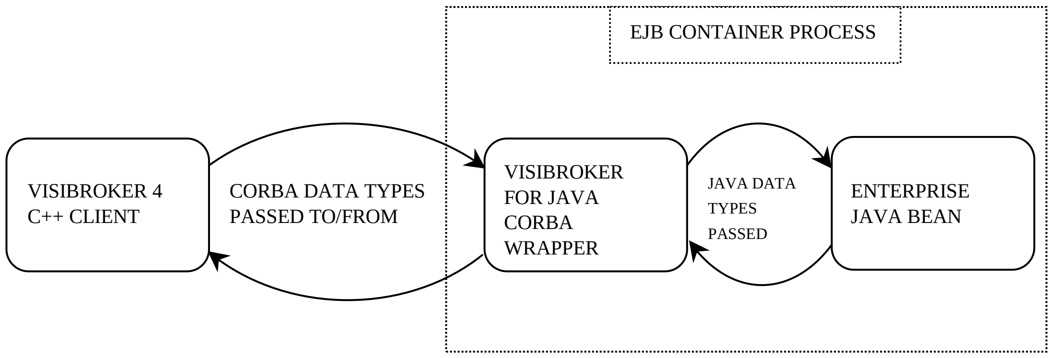

Figure 2 – Using a co-located CORBA wrapper

More details on implementing the CORBA Wrapper strategy are provided later in this paper, in section More Details on Using Server Wrappers to Allow CORBA Client Access, starting on page 19.

12.3 Strategy 2 – Writing Enterprise Java Beans for a Client Known to be non-Java

If it is clearly known at design time that the client types for the EJB may be non-Java then the best strategy is for the EJB writer to avoid using native Java complex types as parameters or return types and instead use Java types with a more natural mapping to IDL.

For example, a sequence of strings in IDL maps to an array of strings in Java; and a sequence of structures maps to an array of Java Objects that have only public data members.

Therefore, if at design time the EJB writer knows that C++ client access is a possibility then rather than use ArrayList's and HashMap's as parameters and return types it would be preferable to use arrays of Strings and arrays of Objects since these will have a more natural mapping for IDL.

As another example, given the following EJB method:

```
float get_price(String symbol) {...}
```

The generated IDL from java2idl would be:

```
float get_price (in ::CORBA::WStringValue symbol);
```

The EJB method would then be invoked by the C++ CORBA client code:

```
 CORBA::WStringValue_ptr symbol = new CORBA::WStringValue(L"INPR");

CORBA::Float price = ejbReference->get_price(symbol);
cout << "The price of INPR is " << price << endl;
```

12.3.1 Strategy 3 – Implementing all the EJB Data Types in C++

We mentioned before that in the general case, there is a complete mapping that describes how each Java data type maps to CORBA IDL. Since the Inprise Application Server uses IIOP as its native communication protocol it is therefore possible to use the Java to IDL mapping tool (Inprise's *java2idl*) to create IDL definitions for all of the EJB parameters and return types, produce C++ stubs from the IDL (using Inprise's *idl2cpp*), and write C++ clients that use these stubs to communicate directly with EJBs. There are a number of steps involved in implementing such a solution:

1) **Produce IDL from the compiled EJB Home by using the Inprise command *'java2idl'* on the EJB's Home**
2) **Produce stubs for use by the C++ client. In this step, we use the Inprise VisiBroker 4 *'idl2cpp'* compiler to produce stub code for use by the client.**

   Note: Since the mapping of some Java classes to IDL may produce code dependencies that cannot be resolved by C++ compilers using nested classes it is important to produce stubs that use C++ namespaces. Using the '-namespace' option to the idl2cpp compiler does this.
3) **Implement the pass-by-value user defined and Java native complex types in C++.**

   In order for the data types that are passed by value between the EJB and the C++ client to be correctly converted between the different languages it is necessary to provide C++ implementations of these types; this applies to both the native Java types and then user defined types.

Figure 3 illustrates these steps.

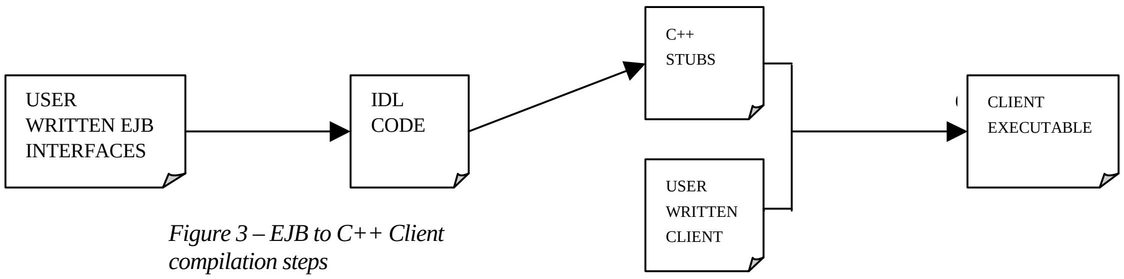

Figure 3 – EJB to C++ Client compilation steps

Although implementing Java native complex types in C++ may seem daunting initially, remember that each of the Java types will always map to exactly the same IDL code. Therefore, a Java native complex type needs only to be implemented once, and can from that point on, be used for every project that requires it. Furthermore, many commercial C++ libraries already exist that implement similar functionality to the Java native complex types; examples include the Standard Template Library (STL), and Rogue Wave's Tools.h++ Pro. Implementing CORBA/EJB functionality using these libraries is simply a matter of providing a CORBA wrapper around these existing classes that translate the Java method signatures to the appropriate commercial library API and implementing any additional data members or methods required by the CORBA mapping.

12.4 Client Case 3 – Java CORBA Clients using VisiBroker 4 (CORBA 2.3)

In this case, as described earlier on page 13, the Java CORBA client uses stubs generated by producing IDL from the EJB interfaces using the Inprise 'java2idl' compiler followed by the production of Java stubs using the 'idl2java' compiler.

We discussed scenarios where this approach might be required, but also discussed other reasons that make this approach unlikely to be used.

Although similar to Client Case 1, this approach requires that all the Java native complex types defined by the generated IDL be re-implemented by the Java client programmer. Although it is true that these types could be simply re-implemented by providing a CORBA wrapper around the Java type, it is cumbersome for the client programmer.

This approach should therefore only be considered if one of the reasons mentioned on page 13 hold.

12.5 Client Case 4 – C++ CORBA Clients using VisiBroker 3 (CORBA 2.2)

In this case, it is necessary to bridge a client written using CORBA 2.2 with an EJB whose interfaces are exposed to the client using CORBA 2.3. This raises a number of general issues and the issue of Objects-by-Value support in particular. The IDL interfaces which are generated from an EJBs home and remote interface require the use of complex data types, which are only supported using Objects-by-Value. This means that only clients using CORBA 2.3 or newer runtimes, such as VisiBroker 4, can make use of EJB objects directly.

In order for a VisiBroker 3 client to access Inprise Application Server 4.0 hosted EJBs IAS 4.0, we again suggest a design consisting of CORBA wrappers around your EJBs:

1) **Write IDL wrapper interfaces for your EJBs.**
2) **Implement those wrapper interfaces using VisiBroker for Java 4, and host them either in the container as described previously, or in their own process.**
3) **Call those wrapper interfaces from VisiBroker for C++ clients, or any other IIOP clients.**

12.6 Client Case 5 – Java CORBA Clients using VisiBroker 3 (CORBA 2.2)

With the exception of the language in which the client is written, this case is the same as Case 4.

12.7 More Details on Using Server Wrappers to Allow CORBA Client Access

Client Cases 2, 4 and 5 all mentioned the idea of using a server side wrapper to allow CORBA client access to EJBs. This concept warrants further discussion. In order to demonstrate some of the issues involved with this approach we'll look at the following example:

Given the following EJB interface:

```
public interface Foo extends EJBObject {
     String method(float arg);
}
```

You might implement the following CORBA wrapper, in IDL:

```
interface FooWrapper {
     string method(in float arg);
};
```

In this case, the CORBA server implementation of FooWrapper should be written in Java using IAS generated stubs (see Client Case 1 above). The implementation object would simply forward the method() requests to the Foo EJB method().

Two approaches for implementing wrappers are:

- An instance of FooWrapper can be created in the same process space as the Inprise EJB Container by writing a Java main() method that instantiates an instance of the FooWrapper implementation and then starts the Container by invoking com.inprise.ejb.Container.main() passing in any required command line arguments as a Java String array, as follows:

```
import javax.ejb.*;
import org.omg.PortableServer.*;

public class ContainerWithWrappers {
    public static void main(String[] args) {
      try {
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
        org.omg.CORBA.Policy[] policies = {
                  rootPOA.create_lifespan_policy(
                  LifespanPolicyValue.PERSISTENT)
        };
       POA myPOA = rootPOA.create_POA(
                   "wrapper_poa", rootPOA.the_POAManager(),policies );
       FooWrapperImpl wrapperServant = new FooWrapperImpl();
       byte[] wrapperId = "FooWrapper".getBytes();
       myPOA.activate_object_with_id(wrapperId, wrapperServant);
       rootPOA.the_POAManager().activate();

       System.getProperties().put(“EJBNoClassLoader”, “true”);
       String[] params = {"ejbcontainer", “wrappedbean.jar”, "-jts", "-jns", "-jss" };
       com.inprise.ejb.Container.main(params);
     }
     catch(Exception e) {
       e.printStackTrace();
     }
   }
}
```

- A second approach would be to instantiate an instance of the FooWrapper implementation in the EJBs constructor or ejbCreate() method.

Both of these two approaches to creating in-process wrapper objects can provide either a one-to-many or a one-to-one relationship between the CORBA wrapper object and the EJB itself. Other approaches could provide more generic CORBA wrapper objects that require specific meta-data to be passed on each invocation so that the wrapper can route the invocation to the correct EJB instance and method.

Each approach has advantages and disadvantages. The approach that provides for one wrapper instance per EJB instance will require management of the wrappers life cycle by the programmer but may provide for more concurrency. The wrapper implementation approach that provides a generic interface to an EJBs methods but requires the passing of meta-data may be harder for the client programmer to use but provides a very simple life-cycle model.

12.7.1 Approach for Client Cases 4 and 5 may be the best approach for Client Case 2

For Client Cases 4 and 5, we suggested writing CORBA wrappers in Java and deploying them in the EJB container. In particular, the IDL of the CORBA wrappers for Client Cases 4 and 5 would not contain any value types (i.e., would not use the CORBA objects-by-value feature).

It may also be best to use this approach when using C++ VisiBroker 4 clients. This is because it is probably simpler for a Java wrapper to convert complex IDL types (e.g., structs, sequences, sequences of structs, etc.) to complex Java types and back, than to re-implement complex Java types in C++. For example, a java.util.Hashtable using strings as keys and floats as values could be passed via an IDL as follows:

```
struct Quote { string symbol; float price; };
typedef sequence<Quote> Quotes;
```

Similarly, the wrapper will have to perform a translation from the RMI Remote exception to a CORBA exception declared in IDL. JDK 1.3 provides utility functions in the javax.rmi.CORBA.Util package that facilitate converting RMI RemoteExceptions to/from CORBA. In essence, this process is the same as handling other complex types.

12.7.2 Publishing the object reference of the CORBA wrapper

Finally, the location of the CORBA wrapper object should be published in a naming service so that clients can look it up. This can be done using either the Inprise Application Server naming service, (using either JNDI calls or CORBA naming service calls), or via the Inprise Smart Agent, or both. VisiBroker 3 or 4 clients can then access the wrapper by performing either a CORBA Name Service lookup, or a Smart Agent *bind*.

12.8 Server Cases 1 – 5

There are no limits on CORBA 2.3 objects calling CORBA 2.2 objects. Therefore, Inprise Application Server 4.0 hosted EJBs, which are CORBA 2.3 objects, have no restrictions. Simply put, an EJB making a call to a CORBA server is simply an example of a Java CORBA client making invocations on a CORBA server. As mentioned in the overview, Inprise Application Server uses fully compliant versions of the transaction and security mapping; so the relevant transaction and security information will be propagated along with the method invocation. It is worth mentioning once again that such seamless operation does not occur in other application server products.

13 Conclusions

This paper has reviewed the issues surrounding the interoperation of CORBA applications with Enterprise Java Beans deployed using Inprise Application Server 4. It has shown that the Inprise Application Server provides CORBA-EJB interoperability that solves these problems, without the limitations that competitors' solutions often contain.

Inprise Application Server 4 supports heterogeneous client environments where any mixture of CORBA 2.2 or 2.3 client and server can interoperate. This is in stark contrast to the approaches of many other application servers, which fail to provide any CORBA-EJB interoperability due to their use proprietary protocols, or their lack of full support for CORBA 2.3 Objects-by-value.

In this paper, most of the challenging issues occurred in the Client Cases, and involved supporting C++ clients, and writing wrappers for clients that could not use the EJB client stubs that IAS automatically generates. The Server Cases, which are more likely, did not cause any complications. Inprise Application Server 4.0 hosted EJBs, which are CORBA 2.3 objects, have no problems invoking other EJBs, CORBA 2.3 objects written in C++, CORBA 2.3 objects written in Java, pre-CORBA 2.3 objects written in C++, and pre-CORBA 2.3 objects written in Java. Transaction and security contexts are transparently propagated across these calls.

With Inprise Application Server, Inprise leverages the years of experience in building CORBA based middleware including VisiBroker (the most deployed CORBA product), and Inprise Integrated Transaction Service (ITS). With Inprise Application Server, these features have been taken to the “next level” with stronger integration with enterprise naming service and special attention to Enterprise Java Beans.

14 Appendix 1: Overview of some of the important aspects of the Java to IDL mapping

- *Java Primitive Types*: Each of the Java primitive types such as float, int, etc. has a specified mapping to IDL:

Java	**OMG IDL**
`boolean`	`Boolean`
`char`	`Wchar`
`byte`	`Octet`
`short`	`Short`
`int`	`Long`
`long`	`long long`
`float`	`Float`
`double`	`Double`

- *Special Cases:* In the Java to IDL mapping there are some special cases. Although not an exhaustive coverage of the subject, the following are some of the more important cases:

  - Whenever a Java name collides with OMG IDL keyword, the Java name is mapped to OMG IDL by adding a leading underscore.

  - For Java names that have leading underscores, the leading underscore is replaced with "J_". For example, _fred is mapped to J_fred.

  - When used as a parameter type, return type, or data member, the Java String type is mapped to an object by value type CORBA::WStringValue. The reason for this special case is that in RMI, nulls can be passed in place of a java.lang.String parameter, whereas in CORBA, nulls can never be passed in place of any parameter. Using the value type WStringValue circumvents this problem because the string *contained by* the value type can be null without having to pass a null in place of the parameter itself.

  - When used as a parameter type, return type or data member a java.lang.Object is mapped to an IDL java::lang::_Object. In the generated IDL this type is simply a typedef for the IDL type "any".

- *Java Native Complex Types:* Native Java complex types such as ArrayLists, Vectors, HashTables, TreeMaps, etc. can be passed to and from EJBs using CORBA. Since these are not remote capable objects they are passed as value types. VisiBroker's Java to IDL compiler, java2idl, will produce IDL code that can be used to implement these value types. On the client side, an implementation of each value type is required. Although at first it may seem intimidating to implement native Java complex types in say C++ it should be noted that each Java type will always map to exactly the same IDL; for example a java.util.Vector will always map to java::util::Vector in IDL. Therefore, the implementation need only be performed once and can then be used for every case where a Vector is required.

- *User Defined Complex Java Types:* A user defined complex type will be passed by reference or as a value type depending on its structure. If the user-defined type conforms to the rules for a remote interface (see below) then the type will be passed by reference. Otherwise, it will be passed as a value type.

- *Defining Remote Interfaces:* A Java interface will map to an IDL interface if the interface inherits from **java.rmi.Remote** and all methods throw **java.rmi.RemoteException**. Note: EJB Home and Remote interfaces conform to these rules.

- *Method Overloading:* Given the current absence of method overloading in IDL a simple name mangling scheme for overloaded methods is used in the Java to IDL mapping. For overloaded Java methods, the mangled IDL name is formed by taking the Java method name and then appending two underscores, followed by each of parameter types of the arguments separated by two underscores. Any punctuation such as "::" or spaces are replaced with underscores. For example the Java methods:

```
public void create();
public void create(float d);
```

  Would generate the IDL:

```
void create__();
void create__float(in float arg0);
```

15 About The Authors

This white paper was written by The Middleware Company at Inprise's invitation to review their IAS technology.

William Edwards is a Senior Consultant for The Middleware Company, a company that provides training and consulting on Enterprise Java Beans (EJB) and other Enterprise Java and J2EE technologies. Will's areas of expertise cover technologies such as CORBA, RMI, Enterprise Java Beans, Internet and Intranet development, Application Servers, Relational Databases and the integration of legacy systems with new computer technology. Prior to The Middleware Company, Will delivered training and consulting on several CORBA projects, using CORBA products from Sun, Inprise/Visigenic, and Iona. Will received an MS in CS from the University of Texas in 1996.

Salil Deshpande is President of The Middleware Company, a worldwide training & consulting company focusing on enterprise software and middleware, and creators of theServerSide.com, the world's largest J2EE and Web Services community. In 1994 Salil founded CustomWare, a consulting and system integration company that focused on CORBA technology. In 1998, Salil spun out the old CustomWare group from Borland, to form a new company, The New CustomWare Company, which continued to focus on enterprise software, but with a special emphasis on on Java 2 Enterprise Edition (J2EE), Enterprise JavaBeans (EJB), and web services technologies. In 2002, in a move that was praised by the J2EE community, CustomWare merged with The Middleware Company, and thus became the obvious choice for J2EE and Web Services consulting, training, mentoring, skills transfer, and and custom software development. Salil holds a B.S. in Electrical Engineering from Cornell University, and an M.S. in Electrical Engineering / Computer Science (distributed operating systems & programming languages) from Stanford University.