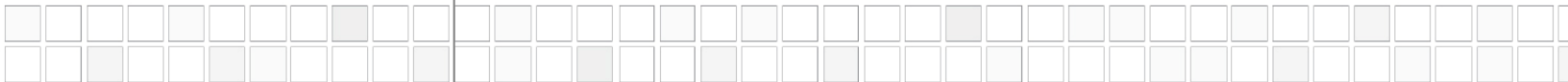




# MIGRATING FROM VISIBROKER TO ORBIX





## PREFACE

This document provides an overview of the changes required to migrate from VisiBroker, which complies with the CORBA 2.0 specification<sup>1</sup>, to Orbix 6, which complies with the CORBA 2.6 specification.

### *Audience*

This document is intended for programmers that want to migrate their VisiBroker applications to any CORBA 2.2 or later compliant ORB, in particular to Orbix.

### *Scope*

This document covers the following aspects of the migration:

- ❑ Code changes: IDL, C++ and Java.
- ❑ System administration differences.

### *Document Organization*

The remainder of this document is organized as follows:

**Section 2, Introduction**, provides some background on the need to migrate legacy VisiBroker applications, and outlines the benefits of migrating to Orbix 6.

**Section 3, IDL**, describes the changes due to updates in the IDL compiler specification, and differences between Orbix and VisiBroker.

**Section 4, Client**, outlines the changes that need to be made on the client side.

**Section 5, Interface Implementation (Servants)**, describes the changes that need to be made in the interface implementation classes both for the inheritance and the delegation approaches.

**Section 6, Server Main Routine**, describes the changes needed in the server to initialize the ORB, create POAs, activate objects and initiate the event loop.

**Section 7, Administration**, outlines the administrative changes required when migrating to Orbix 6.

---

<sup>1</sup> Although VisiBroker 4.x, 5.x and 6.x support the CORBA 2.3 specification, they still support CORBA 2.0 features and proprietary extensions for backward compatibility. Some of these features are not portable when using strict CORBA 2.3+ compliant ORBs such as Orbix 6.



**Section 8, VisiBroker Proprietary Extensions**, describes the VisiBroker proprietary extensions to the CORBA specification and how to migrate them to CORBA-compliant features in Orbix 6.

**Section 9, Automating the Migration**, discusses how the idlgen utility can help in the migration effort.

## INTRODUCTION

This section provides some background on the need to migrate applications from VisiBroker, and outlines the benefits of migrating to Orbix 6.

### *CORBA Compliance*

The CORBA 2.0 specification left some essential aspects unspecified. These aspects affect mostly the server side and include:

- > How clients locate remote objects at runtime and how servers publish those objects
- > How server objects are registered with the Basic Object Adapter (BOA) in order to make them available for remote invocations through calls to object references
- > How to create server objects on-demand
- > How to create multithreaded servers

Because of these flaws in the specification, the code written to use a CORBA 2.0 compliant ORB implementation is not easily portable to other ORB implementations. This causes vendor lock-in that is totally contrary to the spirit of the CORBA standard.


The CORBA 2.2 specification further defined the above functionality in a way that is independent of ORB implementation, thereby making CORBA 2.2 applications portable across ORB implementations.

Legacy applications that use VisiBroker comply with the CORBA 2.0 specification. They need, therefore, to be changed in order to run with ORBs that comply with the CORBA 2.2 or later specifications. This white paper describes the changes that need to be made to make an application comply with the latest CORBA specification.

### *Benefits of Migrating to Orbix*

The following are the benefits of migrating to Orbix:

- > **Strict compliance with the CORBA 2.6 specification** - this translates in improved interoperability, future-proofing your application, and providing support for the latest standard features. It also implies breaking from vendor lock-in by making your code independent of the ORB implementation that you choose.

- 
- > **Highly scalable Portable Object Adapter (POA) features** - servant locator, servant activator, and default servant: With these features, it is possible to create servers that hold large numbers of objects while maintaining limited memory usage. This translates into servers that do not grow as the number of serviced objects increases.
  - > **Fault tolerance and load balancing** - Orbix Enterprise Edition allows you to build highly available systems without single points of failure. This facilitates the smooth operation of highly critical systems. This feature applies to the internal services used by Orbix (locator daemon, naming service, and so on), but can be expanded to your servers too. This allows your servers to be fault resilient and provides the capability of distributing the load among a group of servers. This is accomplished by configuring the locator daemon and is, therefore, transparent for both clients and servers.
  - > **Security** - Orbix 6 provides a Security Framework (iSF) that allows clients and servers to transparently use Enterprise Security System (ESS) features without making explicit security calls in the application code. It integrates seamlessly with LDAP and SiteMinder. Orbix also provides support for Transport Layer Security (TLS) v1.0 and Secure Sockets Layer (SSL).
  - > **Management** - Orbix 6 provides support for Simple Network Management Protocol (SNMP), which allows integration with Enterprise Management Systems such as IBM Tivoli™ and HP OpenView™.
  - > **Transport plugins** - Orbix' Adaptive Runtime Architecture (ART) facilitates the use of plugins that allow you to replace and enhance the functionality of the ORB. A good example where this plugin architecture shows its flexibility is the implementation of the transport mechanism that the ORB uses for communication. Besides the default IIOp plugin, Orbix also comes with a multicast plugin and a shared-memory plugin. With the multicast plugin, the ORB uses multicast (UDP) for the communication between clients and servers. With the shared-memory plugin, the communication between the co-hosted client and server is optimized to use shared memory rather than TCP/IP to increase its throughput.
  - > **Active Connection Management (ACM)** - Allows servers to handle more connections than the physical limit imposed by the operating system. This feature is extremely important for servers that need to scale to a large number of clients. Using ACM makes CORBA clients easier to implement because they do not need to deal with connection management. Migrating to Orbix allows you to remove client code that takes care of closing connections—Orbix ACM handles it automatically.
  - > **Improved performance** - Translates into a more efficient CPU usage, increased throughput, and shorter response times, which together translates into lowered operation costs.
  - > **Centralized configuration** - With this option, the changes to configuration are done at a central location and they do not need to be replicated in every machine in the system. This translates into lowered operation costs.

- > **Session management** - Used to manage server-side objects associated with client sessions.
- > **Asynchronous messaging interface** - Allows the implementation of asynchronous calls on top of synchronous operations. This affects the client only, and is useful when there is a need to decouple the clients from the server but without compromising error detection.

## IDL

This section describes the changes due to the IDL compiler, including:

- > The executables and their command-line arguments (see 3.1 Executables)
- > The generated classes as specified by the CORBA 2.2 specification are different from the ones generated by the VisiBroker's IDL compiler (see 3.2 Generated Classes)

### Executables

The executables and their command-line arguments need to be changed. *Table 1* shows the commands used in VisiBroker and their equivalent commands in Orbix. One of the major changes is that only one IDL compiler is required for C++, Java, and for populating the IFR.

**Table 1: IDL Commands**

VisiBroker Executable	Orbix Equivalent
idl2cpp	idl -base -poa <options>
idl2java	idl -jbase -jpoa <options>
idl2ir	idl -R
java2idl	no equivalent
java2iiop	no equivalent
vbj	it_java <sup>2</sup>

---

<sup>2</sup> From Orbix v6.0 onwards, this wrapper is no longer needed.

The Orbix IDL compiler offers more options. For more information on those options, please refer to the Orbix CORBA Programmer's Guide [IONA03c].

## Generated Classes

There are some slight differences in the names of some generated classes and filenames. This subsection describes these differences.

### C++

In C++, the following changes in the generated classes are required:

- > **Skeleton class and tie template:** The skeleton class and `tie` template names have been standardized
- > **Default filenames:** The default file names for the generated files are different from VisiBroker. However, with the Orbix IDL compiler it is possible to specify any file name suffix

### Skeleton Class and tie Template

The C++ skeleton and `tie` template classes have different names in CORBA 2.2. *Table 2* shows the classes generated by the VisiBroker IDL compiler and the equivalent classes generated by a CORBA 2.2 compliant IDL compiler for the `Invoice` interface. It also shows the generalized standard name for any `<interface>`.

**Table 2: Skeleton and tie Template Classes**

Class	VisiBroker	CORBA 2.2	Generalised
Skeleton class	<code>_sk_Invoice</code>	<code>POA_Invoice</code>	<code>POA_&lt;Interface&gt;</code>
tie template class	<code>_tie_Invoice</code>	<code>POA_Invoice_tie</code>	<code>POA_&lt;Interface&gt;_tie</code>

### Default Filenames

*Table 3* shows the default names for files that the IDL compiler generates for the `Invoice.idl` file and the generalised name for any given `<filename>`. Note that with Orbix it is possible to specify any name suffix for these files.

**Table 3: Default Filenames**

File	VisiBroker	Orbix	Generalized
Stub header	Invoice_c.hh	Invoice.hh	<filename>.hh
Stub implementation	Invoice_c.cc	InvoiceC.cxx	<filename>C.cxx
Skeleton header	Invoice_s.hh	InvoiceS.hh	<filename>S.hh
Skeleton implementation	Invoice_s.cc	InvoiceS.cxx	<filename>S.cxx

### Java

For Java, several class names have been standardized. The generated Java classes have different names in CORBA 2.2. *Table 4* shows the classes generated by the VisiBroker IDL compiler and the equivalent classes generated by a CORBA 2.2 compliant IDL compiler for the `Invoice` interface. It also shows the generalized standard name for any `<interface>`.

**Table 4: Generated Java Classes**

Class	VisiBroker	CORBA 2.2	Generalized
Stub class	_st_Invoice	_InvoiceStub	_<interface>Stub
Implementation base class	_InvoiceImpl Base or _sk_Invoice	InvoicePOA	<interface>POA
tie class	_tie_Invoice	InvoicePOATie	<interface>POATie

### *The Context Clause*

Although Orbix 6 supports the IDL context clause, its use is discouraged because it breaks CORBA type safety [Henn99] and can potentially be removed from the CORBA specification in the future.

The alternative to using IDL contexts is to use service contexts in conjunction with portable interceptors.



## The Principal Type

The General Inter-ORB Protocol (GIOP) 1.0 and 1.1 specifications deprecate the use of the principal IDL type, and the GIOP 1.2 completely removes its use in the request header. VisiBroker uses GIOP 1.0, whereas Orbix supports GIOP 1.0, 1.1 and 1.2. Orbix has some limited on-the-wire support for the principal type, in order to support interoperability with Orbix 3.x applications. If the principal type is used to provide security verification, it is better to rely on mechanisms that are supported by the current CORBA specification, such as service contexts. Orbix provides additional security mechanisms such as the leasing (session management) plugin or the security service. The latter is available in Orbix 6.0 and later versions.

## CLIENT

This section outlines the changes that need to be made on the client side. These changes are minimal because the CORBA specification has not changed much on the client side.

### ORB Initialization

ORB initialization is already specified in the CORBA 2.0 specification. However, most of the command-line arguments used to control the ORB differ. If these arguments are hard-coded, they need to be changed. For Java, the same can be said about property name-value pair objects that are passed to the `ORB.init` call.

### Object Location: `_bind`

The `_bind` function is a VisiBroker proprietary extension to the CORBA specification. Because it is not part of the CORBA standard, Orbix does not support it. This function makes several assumptions about object references and is not interoperable across ORBs from different vendors.

There are other ways to deal with object location, including:

- > Using an OMG-compliant naming service
- > Using an OMG-compliant trader service
- > Using object references as operation call parameters
- > Using `string_to_object()` calls with URL-style object references (for example, `corbaloc` and `corbaname`)
- > Using `resolve_initial_references()`

Which of these ways is appropriate should be decided on a case-by-case basis. In most cases, however, using a naming service is the best solution. A wrapper

class can be implemented to hide the solution being used. This effectively decouples the clients and server from the decision and provides greater flexibility.

### **CORBA::Any type (C++ only)**

Recent revisions of the CORBA specification deprecated the following member functions:

```
// CORBA::Any Constructor.
CORBA::Any(
    CORBA::TypeCode_ptr tc,
    void *value,
    CORBA::Boolean release = 0)

// CORBA::Any::replace() function.
void replace(
    CORBA::TypeCode_ptr,
    void* value,
    CORBA::Boolean release = 0
);
```

### **DII**


The VisiBroker DII differs from the CORBA 2.2 specification in the return value of certain `CORBA::Request` operations (*Table 5* shows these operations). The CORBA 2.2 specification deprecated the use of `CORBA::Status`. Therefore, while VisiBroker member functions return `CORBA::Status`, the CORBA specified member functions have a void return type.

**Table 5: CORBA::Request Operations that use CORBA::Status**

<b>VisiBroker</b>	<b>CORBA 2.2</b>
<code>CORBA::Status invoke()</code>	<code>void invoke()</code>
<code>CORBA::Status send_oneway()</code>	<code>Void send_oneway()</code>
<code>CORBA::Status send_deferred()</code>	<code>Void send_deferred()</code>
<code>CORBA::Status get_response()</code>	<code>void get_response()</code>

### **System Exceptions**

There are two areas of concern when dealing with system exceptions:

- 
- > **Semantics** - The semantics for `CORBA::COMM_FAILURE` and `CORBA::INV_OBJREF` exceptions were revisited in the CORBA 2.3 specification. In addition, `CORBA::TRANSIENT` and `CORBA::OBJECT_NOT_EXIST` exceptions were introduced.
  - > **Minor codes** - Minor code values are not mandated by the CORBA specification. They, therefore, vary from ORB to ORB, and portable code should not rely on them.


## Semantics

The semantics of `CORBA::COMM_FAILURE` and `CORBA::INV_OBJREF` exceptions changed in the CORBA 2.3 specification. Two new exception types were added that match the semantics of old exceptions types. These are:

- ❑ `CORBA::TRANSIENT`, which is equivalent to `CORBA::COMM_FAILURE`
- ❑ `CORBA::OBJECT_NOT_EXIST`, which is equivalent to `CORBA::INV_OBJREF`

The exception semantics in CORBA 2.3 are:

- > **CORBA::COMM\_FAILURE:** This exception is raised if communication is lost while an operation is in progress, after the client sends the request, but before the reply from the server has been returned to the client. Before the CORBA 2.3 specification this exception was also raised when the ORB attempted to reach an object and failed; the CORBA 2.3 specification clarified the semantics of the `CORBA::TRANSIENT` exception specifically to account for this condition.
- > **CORBA::TRANSIENT:** This exception indicates that the ORB attempted to reach an object and failed. It does not indicate that an object does not exist. Instead, it simply means that no further determination of an object's status was possible because it could not be reached. For example, this exception is raised if an attempt to establish a connection fails because the server or the implementation repository is down.
- > **CORBA::INV\_OBJREF:** This exception indicates that an object reference is internally malformed. For example, the repository ID might have incorrect syntax or the addressing information might be invalid. `ORB::string_to_object` raises this exception if the passed string does not decode correctly. An ORB implementation might detect calls via nil references (although it is not obliged to detect them). `CORBA::INV_OBJREF` is used to indicate this. Before the CORBA 2.3 specification this exception was also raised when an invocation was performed on an object that was not in the server's address space; the CORBA 2.3 specification clarified the semantics of the `CORBA::OBJECT_NOT_EXIST` exception specifically to account for this condition.
- > **CORBA::OBJECT\_NOT\_EXIST:** This exception is raised whenever an invocation on a deleted object is performed. It is an authoritative *hard* fault report. Anyone receiving it is allowed (even expected) to delete all copies of this object reference and to perform other appropriate *final recovery* style procedures.



Any code that explicitly catches the `CORBA::COMM_FAILURE` and `CORBA::INV_OBJREF` exceptions needs to be revised to determine if this is the appropriate exception to catch. Since these exceptions can still be raised, they should be caught also, either explicitly or by catching `CORBA::SystemException` afterwards.

## Minor Codes

Minor codes are used to differentiate between subcategories within a particular `CORBA::SystemException`. The CORBA specification leaves the assignment of minor codes values to each ORB implementation. A CORBA compliant application, therefore, should not rely on these minor codes other than for logging purposes.

System exception minor codes are used mainly in two ways:

- > For logging and debugging purposes. This use is fine because the logic does not depend on the minor code; it merely records the number.
- > To take a specific action based on the minor code returned. This use is not CORBA-compliant because the logic depends on the actual minor code. If the minor codes still need to be used, the application needs to be changed to use Orbix minor codes.

## INTERFACE IMPLEMENTATION (SERVANTS)

There are two approaches to define the interface implementation classes:

- > Inheritance (see subsection 5.1)
- > Delegation (see subsection 5.2)

In CORBA 2.2, these approaches have been standardised to make them portable across different ORBs and they differ slightly from the scheme used by VisiBroker.

A major change in the specification has been the distinction between a CORBA object and servant. Subsection 5.3 describes this change and its implications.

Another small change in the CORBA 2.2 C++ specification is the introduction of special types for out parameters. Subsection 5.4 gives details of the implications of this change.

### *Inheritance Approach*

For the inheritance approach the name of the skeleton class has been standardised. The following subsections give an example of the changes that need to be made to C++ and Java code that uses these classes.

#### **C++**

In C++, the skeleton class changed from `_sk_<interface>` to `POA_<interface>`. The implementation class, therefore, needs to be changed.

In addition, the initialization of the base class is not needed anymore because object activation is done after the object is instantiated (see subsection 6.2 for details). For example, if the `InvoiceImpl` class is as follows:

```
class InvoiceImpl : public _sk_Invoice
{
    public:
        InvoiceImpl(const char *po_id=NULL) :
            _sk_Invoice(po_id) { . . . }
    . . .
}
```

When migrated to a CORBA 2.2 compliant ORB, it would look like this:

```
class InvoiceImpl : public virtual POA_Invoice
{
    public:
        InvoiceImpl() { . . . }
    . . .
}
```

Note that the inheritance from `POA_Invoice` needs to be virtual in order to use `InvoiceImpl` in a hierarchy that uses multiple inheritance. Virtual inheritance provides a way to disambiguate common base class members.

The `po_id` constructor parameter (Purchase Order Identifier) is only used when the object is activated (see subsection 6.2 for details).


## Java

In Java, the skeleton class changed from `<interface>ImplBase` to `<interface>POA`. The implementation class, therefore, needs to be changed. In addition, the initialization of the base class is not needed anymore because object activation is done after the object is instantiated (see section 6.2 for details). For example, if the `InvoiceImpl` class appears as follows:

```
class InvoiceImpl extends _InvoiceImplBase
{
    public InvoiceImpl(String po_id) {
        super(po_id);
    }
    . . .
}
```

When migrated to a CORBA 2.2 compliant ORB, it would look like this:

```
class InvoiceImpl extends InvoicePOA
{
    public InvoiceImpl() { . . . }
    . . .
}
```



The `po_id` constructor parameter (Purchase Order Identifier) is only used when the object is activated (see section 6.2 for details).

## **Delegation Approach**

The delegation approach with CORBA 2.2 looks similar to VisiBroker's. The following subsections show the differences both in C++ and Java.

### **C++**

For C++, the implementation class does not need to be changed, only the instantiation of the delegate and servant objects. For example, in VisiBroker the `tie` servant is instantiated as follows:

```
InvoiceImpl tied_object;  
_tie_Invoice<InvoiceImpl> tieServer(  
    tied_object, "PO12345");
```

When migrated to a CORBA 2.2 compliant ORB, however, it looks like this:

```
the_poa = ...  
InvoiceImpl* tied_object = new InvoiceImpl();  
POA_Invoice* the_tie =  
    new POA_Invoice_tie<InvoiceImpl>(tied_object, the_poa);
```

Although this is just one of many constructors that the `tie` class provides, it is better to specify the actual POA to be used because otherwise the root POA will be used.

The constructor parameter "PO12345" (Purchase Order Identifier) is only used when the object is activated (see subsection 6.2 for details).

### **Java**


For the delegation approach with CORBA 2.2 for Java, the name of the operations interface is the same as is in VisiBroker; that is, `<interface>Operations`. The implementation class, therefore, does not need to be changed. However, the name of the `tie` class has changed and, therefore, the instantiation of the `tie` servant needs to be changed.

For example, in VisiBroker the `tie` servant is instantiated as follows:

```
InvoiceImpl tied_object = new InvoiceImpl();  
_tie_Invoice the_tie = new _tie_Invoice(tied_object);
```

When migrated to a CORBA 2.2 compliant ORB, however, it looks like this:

```
the_poa = ...  
InvoiceImpl tied_object = new InvoiceImpl();  
InvoicePOATie the_tie =  
    new InvoicePOATie(tied_object, the_poa);
```



Although this is just one of many constructors that the `tie` class provides, it is better to specify the actual POA to be used because otherwise the root POA will be used.

## ***CORBA Object and Servant Lifecycle***

A major change in the CORBA 2.2 specification is the distinction between CORBA object and servant. The servant is the actual implementation object whereas the CORBA object is represented by the object reference used by the client. This distinction stems from the fact that a client can make calls to a CORBA object that has no active servant associated with it (for example, when the servant is activated on demand), or that many CORBA objects can be associated with a single servant.

The lifecycle of a CORBA object and servant are separate; that is:

- > A CORBA object can *outlive* the servant used to create it. If this is the case, the CORBA object is said to be *persistent*. In contrast, a CORBA object that is only valid while the servant is active is said to be *transient*.
- > The servant can be associated with one or many CORBA objects.
- > The creation or destruction of an object reference does not imply either the creation or destruction of the servant, and vice-versa.

### **CORBA::release, <interface>::\_release and <interface>::\_duplicate**

Because of the distinction between servants and object references, you need to change any code that treats the target object as an object reference for memory management purposes. Any use of the functions `CORBA::release`, `<interface>::_release` and `<interface>::_duplicate` need to be revisited in the server because the servant objects cannot be used directly.

## **Managing Object References**


Given that object references and servants are separate, since CORBA 2.2, an object reference must be explicitly created when needed. There are many ways to get an object reference, including:

- > `_this()`
- > `POA::servant_to_reference()`
- > `POA::id_to_reference()`
- > `POA::create_reference()` and `POA::create_reference_with_id()`

All of the above functions return newly created object references. You must, therefore, release them once you are finished with them (or use `_var` types).

## ***Member Function Signatures: out Parameters***

The CORBA 2.2 C++ specification defines special types for `out` parameters. An IDL `out` parameter of type `T` is now mapped to C++ as a `T_out` parameter.



These new parameter types are used in conjunction with `T_var` types to prevent memory leaks. Any C++ member function signatures must be updated to use these `T_out` types.

## SERVER MAIN ROUTINE

In VisiBroker, only one instance of the BOA is created in the server. Because of that, all the objects in each server share the same object adapter and behave in the same manner. With the CORBA 2.2 specification it is possible to create multiple POAs. That way a server process can contain servants with different behaviors.

This section shows the changes needed to initialize the ORB, create POAs, activate objects and initiate an event loop.

### *ORB and BOA Initialization*

In VisiBroker, the ORB and the BOA need to be explicitly initialized. The ORB initialization needs to be revisited, in the same manner as in the client side, because the command-line arguments differ. If these arguments are hard-coded, they need to be changed. For Java, the same can be said about property name-value pair objects that are passed to the `ORB.init` call.

This subsection shows how to obtain a reference to the root POA, how to create a POA with default policies, how to specify POA policies, and how to create a POA with custom policies.

### **Portable Object Adapter (POA)**

BOA initialization must be replaced with the creation of a POA (or POAs). The behavior of the POAs can be controlled through objects called *policies*. The specification defines a POA that is created implicitly when the ORB is created, that is called the *root POA*. This POA has a set of default policies that are useful only to the simplest servers.

To obtain a reference to the root POA, a call to `CORBA::ORB::resolve_initial_references` needs to be made as follows:

#### **C++:**

```
// Initialize ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
//get an object reference to the root POA
CORBA::Object_var tmp_ref =
    orb->resolve_initial_references( "RootPOA" );
PortableServer::POA_var root_poa = POA::_narrow( tmp_ref );
// _narrow returns a _nil object if it fails, must verify
root
```



POA

```
// before making any calls assert(!CORBA::is_nil(root_poa));
```

### Java:

```
// Add the ORBClass and ORBSingletonClass system properties
// These can be specified through command-line arguments
// too.
```

```
Properties sys_properties = System.getProperties();
sys_properties.put("org.omg.CORBA.ORBClass",
"com.ionacorba.art.artimpl.ORBImpl");
sys_properties.put("org.omg.CORBA.ORBSingletonClass",
"com.ionacorba.art.artimpl.ORBSingleton");
```

```
ORB orb = ORB.init(args, sys_properties);
org.omg.CORBA.Object tmp_ref =
orb.resolve_initial_references(
"RootPOA" );
POA root_poa = POAHelper.narrow( tmp_ref );
// If narrow fails, it throws an exception here
```

The `_narrow` call is needed to *narrow* from the generic CORBA object type to the specific `PortableServer::POA` type.

For more complex servers, you need to create your own POA with its own set of policies. The following subsections show you how to:

- ❑ Pick the appropriate POA policies according to an applications requirements.
- ❑ Create a POA with a custom set of policies.

### POA Policies

*Table 6* shows the different types of policy, the values that they can take, their default value and what value the root POA is created with (for details see [KB00a]).

**Table 6: POA Policy Types**

Policy Type	Values	Default Value	rootPOA
Servant Retention	RETAIN, NON_RETAIN	RETAIN	RETAIN
ID Uniqueness	UNIQUE_ID, MULTIPLE_ID	UNIQUE_ID	UNIQUE_ID
Request Processing	USE_ACTIVE_OBJECT_MAP_ONLY USE_SERVANT_MANAGER USE_DEFAULT_SERVANT	USE_ACTIVE_OBJECT_MAP_ONLY	USE_ACTIVE_OBJECT_MAP_ONLY
Thread Policy	ORB_CTRL_MODEL, SINGLE_THREAD_MODEL	ORB_CTRL_MODEL	ORB_CTRL_MODEL
LifeSpan	PERSISTENT, TRANSIENT	TRANSIENT	TRANSIENT
Id Assignment	USER_ID, SYSTEM_ID	SYSTEM_ID	SYSTEM_ID
Implicit Activation	IMPLICIT_ACTIVATION NO_IMPLICIT_ACTIVATION	NO_IMPLICIT_ACTIVATION	IMPLICIT_ACTIVATION

### How to Pick your POA Policies

Picking up your policies can look like a complex job, given the large number of combinations that they can take [KB99a]. However, not all combinations are possible and there is a correct set of policies depending on what you want to accomplish. The following subsections have pseudo-code that shows you how to selecting the right POA combination.

#### Basic Models for Servant Activation

The following pseudo-code helps determine the model that should be used for servant activation:

```

If all the servants are kept in memory then
  Use RETAIN and UNIQUE_ID policies
  If you want to activate all of your servants at
  start up then
    Use the policy USE_ACTIVE_OBJECT_MAP_ONLY
  otherwise //POA activates servants on demand
  //and they remain active
    // Servant Activator
  
```

```

        Use the policy USE_SERVANT_MANAGER
    endif
otherwise //servants are not kept in memory
    If your servants contain state then
        // Servant Locator (your servant locator
        //must implement a cache of servants)

        Use USE_SERVANT_MANAGER and UNIQUE_ID
        policies
    otherwise //stateless servants
        Use the MULTIPLE_ID policy (same servant
        implements several objects)
        If all your servants belong to a few
        interface types then
            //Default Servant
            Use the policy USE_DEFAULT_SERVANT
        otherwise
            //Servant Locator with a servant for
            //each interface.
            //ObjectID must encode interface type
            Use USE_SERVANT_MANAGER
        endif
    endif
endif
endif

```

## Threading

```

    If your servant code is thread-safe then
        //Multi-threaded POA
        Use the ORB_CTRL_MODEL
    otherwise
        //Single-threaded POA
        Use the SINGLE_THREAD_MODEL
    endif

```

## Life Span

```

    If you need to contact the object, even after the
    process is killed then
        Use the PERSISTENT policy
    otherwise
        Use the TRANSIENT policy
    endif

```

## ID assignment

```

    If your objects are associated with external data
    then
        Use the USER_ID usually used with PERSISTENT
        (may be used with TRANSIENT)
    otherwise
        Use the SYSTEM_ID (only used with TRANSIENT)
    endif

```

## Implicit activation

```

    If you want your servants to be implicitly
    activated when you try to create object references

```

```

for them then
    // Not advisable
    //Must use with TRANSIENT, SYSTEM_ID, RETAIN
    and USE_ACTIVE_OBJECT_MAP_ONLY
    Use the IMPLICIT_ACTIVATION policy.
otherwise
    // Recommended
    Use the NO_IMPLICIT_ACTIVATION policy
endif

```

## Creating POA with Custom Policies

A POA policy list is provided to the POA when it is created. The policy list overrides the default policy values; that is, you only need to specify those policies that are different from the default policies values.

In the following example code we are creating a POA with the following characteristics:

- > All servants are kept in memory.
- > Servants are activated at startup.
- > Our servant is thread-safe and we want to dispatch requests using multiple threads.
- > We want object references to be valid even after the server process exits.
- > During servant activation, we want to assign object IDs to our servants.
- > Servants are going to be activate explicitly.

*Table 8* shows the policy selection based on the above requirements and the default value for each policy type.

**Table 8: Sample Policy Selection**

Policy Type	Default value	Required Value
Servant Retention	RETAIN	RETAIN
ID Uniqueness	UNIQUE_ID	UNIQUE_ID
Request Processing	USE_ACTIVE_OBJECT_MAP_ONLY	USE_ACTIVE_OBJECT_MAP_ONLY
Thread Policy	ORB_CTRL_MODEL	ORB_CTRL_MODEL
Life Span	TRANSIENT	PERSISTENT
ID Assignment	SYSTEM_ID	USER_ID
Implicit Activation	NO_IMPLICIT_ACTIVATION	NO_IMPLICIT_ACTIVATION

From this table we determine that the only policies that need to be overridden are:

- > Life span: PERSISTENT
- > ID Assignment: USER\_ID

## C++

```
//get an object reference to the root POA
PortableServer::POA_var poa = . . .

assert(!CORBA::is_nil(root_poa));

//Obtain the root POA manager to assign it to new POA
PortableServer::POAManager_var root_poa_manager
    = root_poa->the_POAManager();

//create policy list
CORBA::PolicyList policies;
policies.length (2);
policies[0] = poa->create_lifespan_policy
    (PortableServer::PERSISTENT)
policies[1] = poa->create_id_assignment_policy
    (PortableServer::USER_ID)

//create a POA for persistent objects
poa = poa->create_POA( "persistentPOA", root_poa_manager,
    policies );
```

## Java

```
//get an object reference to the root POA
POA root_poa = . . .

//Obtain the root POA manager to assign it to new POA
POAManager root_poa_manager = root_poa.the_POAManager();

//create policy list
org.omg.CORBA.Policy[] policies = new
org.omg.CORBA.Policy[2];
policies[0]=root_poa.create_lifespan_policy(
    LifespanPolicyValue.PERSISTENT);
policies[1]=root_poa.create_id_assignment_policy(
    IdAssignmentPolicyValue.USER_ID);

// Create the POA
POA my_poa = root_poa.create_POA("persistentPOA",
    root_poa_manager, policies);
```

## POAUtility Class

In order simplify the creation of POAs, Progress provides a class called POAUtility [IONA04a]. This class and its documentation can be downloaded from Progress's developer center at: <http://www.iona.com/devcenter> in the CORBA section under the heading *Creation of POA Hierarchies made Simple*.



## Object Name and Activation

VisiBroker makes objects available by registering them with the server's BOA using `BOA::obj_is_ready`. The CORBA 2.2 specification performs a similar object registration with the POA by calling `POA::activate_object` or `POA::activate_object_with_id`. Which operation you use depends on whether you want the POA to pick an ID for your object (`SYSTEM_ID` policy) or you want to provide an object ID (`USER_ID` policy). If you want the POA to pick an ID, use `POA::activate_object` and if you want to provide an object ID, use `POA::activate_object_with_id`.

### Persistent Objects

If an object name is provided when the object is instantiated then `POA::activate_object_with_id` must be used. This implies using a POA with `USER_ID`, `PERSISTENT` and `NO_IMPLICIT_ACTIVATION` policies. For example:

#### **VisiBroker**

```
InvoiceImpl anInvoice("P012345");
boa->obj_is_ready(&anInvoice);
```

#### **CORBA 2.2, C++**

```
InvoiceImpl anInvoice;
PortableServer::ObjectId_var oid =
    PortableServer::string_to_ObjectId("P012345");
persistent_poa->activate_object_with_id(oid, &anInvoice);
```

#### **CORBA 2.2, Java**

```
InvoiceImpl anInvoice = new InvoiceImpl();
byte[] oid="P012345".getBytes();
persistent_poa.activate_object_with_id(oid,anInvoice);
```

### Transient Objects

If the object name is not provided when the object is instantiated then `POA::activate_object` can be used. This implies using a POA with `SYSTEM_ID` and `TRANSIENT` policies. For example:

#### **VisiBroker**

```
// Create a new Invoice object.
InvoiceImpl anInvoice();
// Export the newly created object.
boa->obj_is_ready(&anInvoice);
```

#### **CORBA 2.2, C++**

```
InvoiceImpl anInvoice;
PortableServer::ObjectId_var oid
    = transient_poa->activate_object(&anInvoice);
```

### **CORBA 2.2, Java**

```
InvoiceImpl anInvoice = new InvoiceImpl();
byte[] oid= transient_poa.activate_object(anInvoice);
```

## **Event Loop**

The event loop in VisiBroker is initiated by calling the `BOA::impl_is_ready` function. With CORBA 2.2, in order to initiate the event loop, the following steps are required:

1. Activate every POA manager. If a POA manager is shared by several POAs, it needs to be activated only once. 

```
PortableServer::POA_var root_poa = ... // Obtain root POA
PortableServer::POAManager_var root_poa_manager =
root_poa->the_POAManager(); root_poa_manager->activate();
```
2. Call `ORB::run` to start the event loop: 

```
orb->run();
```

## **ADMINISTRATION**


This section outlines the administrative changes required when migrating to Orbix. Administration is one of the few areas not covered by the CORBA specification and is, therefore, specific to each ORB implementation.

### **Infrastructure Services**

The Implementation Repository (IMR) in Orbix is very similar to VisiBroker. The location and activation mechanisms differ on the level of granularity used. Whereas VisiBroker deals with persistent objects, Orbix deals with persistent POAs.

With VisiBroker, the OSAgent implements a fault-tolerant object location mechanism and the Object Activation Daemon (OAD) activates the servers on demand. The `obj_is_ready` call informs the OSAgent when a persistent object is being activated. The server process that holds the object needs to be explicitly registered with the OAD.

With Orbix, the object location mechanism is implemented by the Locator Daemon, which can also be replicated for fault tolerance. There is a node daemon on every host that holds servers with persistent POAs. The locator daemon manages these servers in conjunction with the node daemons. The locator daemon and the node daemons under its control are collectively called the *location domain*. The



`POA::create_POA` call informs the locator daemon when a persistent POA is created.

### ***Deployment: Server versus POA Registration***

In order to activate a server on demand, the server process that holds the object needs to be explicitly registered with the locator daemon through the `itadmin` utility. This utility can be invoked from the command-line and also provides a TCL interpreter that allows the invocation of commands using TCL scripts. This feature allows the registration process to be automated. In fact, Progress provides a set of administration utilities that take advantage of this feature to make the server administration and configuration easier [IONA04a].

### ***Command-Line Options***

CORBA only specifies the use of the `-ORB` prefix to control the behavior of the ORB through command-line options. The actual options are ORB-dependent. Therefore, command-line options that are passed to clients and servers need to be revisited.

### ***Build Changes***

The libraries and compiler flags used in Orbix are completely different from those used in VisiBroker. In order to modify the build system one should take as an example the build system provided with Orbix demos. Alternatively, it is possible to generate a sample `makefile` from the IDL source code by using `idlgen` (see section 9 for details on how to generate an application using `idlgen`).

## **VISIBROKER PROPRIETARY EXTENSIONS**

This section describes the VisiBroker proprietary extensions to the CORBA specification and how to migrate them to CORBA compliant features, including:

- > Activator
- > Persistent object implementations
- > BindOptions
- > Thread management
- > Event handlers
- > Interceptors and object wrappers
- > Smart stubs
- > Location service
- > ORB manager interfaces



## Activator

VisiBroker's Activator allows the loading of objects on demand. It provides hooks to create objects on demand when the BOA first receives a request for them, and to destroy them when the BOA deactivates them.

CORBA 2.2 provides the following mechanisms in order to load objects on demand:

**Servant Activator:** Provides a way to instantiate an object on demand when the POA receives the first request meant for that object. Useful to implement a lazy load mechanism to load objects from persistent storage only when they are really needed. It is meant for a relatively small number of active servants because once a servant is loaded into memory, it remains there until the server exits or it is removed under very exceptional circumstances.

**Servant Locator:** Also a mechanism for loading objects on demand. The difference between it and the Servant Activator is that the POA requests a servant from the Servant Locator for every single call. This approach is ideal to implement a servant cache that limits the memory usage of active servants by evicting unused servants. Because it limits the memory usage, the Servant Locator with cache scales very well for large numbers of servants.


**Default Servant:** With this mechanism a single servant services requests meant for multiple CORBA objects. This approach scales very well because there is no need to instantiate new servants or spend time figuring out what servant to use. Since the same servant is used for multiple objects, only stateless servants can be used. Because of this, the servant code usually interfaces directly with a database to manage its state information.

Whereas the servant activator is a direct replacement to the VisiBroker Activator, it might be worth considering the servant locator or the default servant if a large number of servants is required.

## Persistent Object Implementations

The persistent `tie` template classes, `ptie`, are used to integrate with object databases. They differ from regular `tie` template classes in that they contain a *service name* and a *reference data*. The service name merely associates the object with an activator, and the reference data provides the identity of the persistent object.

CORBA 2.2 provides servant manager classes in order to load objects on demand. Objects are indirectly associated with a servant manager through the POA that activates them. This is equivalent to the `ptie` service name. In addition, objects



are associated with identifiers that are used to uniquely identify them within a POA; this object identifier is equivalent to the `ptie` reference data.

It is possible, therefore, that using servant managers allows you to achieve the same functionality as with the persistent `tie` template.

## ***BindOptions***

`BindOptions` control the behavior of `_bind`. While `_bind` is not part of the CORBA specification, it is possible to control the communication behavior of requests by using the ORB quality of service (QoS) `RebindPolicy` and `Timeout` policies (as defined in the CORBA 2.4 specification) in much the same way as VisiBroker's `BindOptions`. Orbix provides the option of configuring the QoS polices from the configuration and programmatically at the ORB, thread and object reference levels.

## ***Thread Management***

VisiBroker provides support for both single-threaded and multi-threaded servers by providing both single-threaded and multi-threaded libraries. For multi-threaded servers, VisiBroker provides two threading models:

- > Thread pooling
- > Thread-per-session

### **Single Threading**


The single threaded model can be easily accomplished with CORBA 2.2 with a POA that uses the `SINGLE_THREAD_MODEL` threading policy. Orbix does not require separate libraries to be linked to use this model. In fact, both multi-threaded and single-threaded POAs can cohabit in the same server process.

### **Thread Pooling**

With thread pooling, each request is dispatched by one of many threads from a pool of threads. Once a request is dispatched, the thread is returned to the pool to be reused by another request. The CORBA 2.2 specification defines the `ORB_CTRL_MODEL` model for multithreading. It is up to the ORB implementation to determine the best approach for this model. Orbix implements the `ORB_CTRL_MODEL` model using a thread pool.

### **Thread-per-session**

With thread-per-session, a new thread is created each time a new client connects to the server. There is no equivalent feature in the CORBA 2.2 specification. Orbix does not, therefore, support this approach out of the box. However, it is possible to implement this approach by creating a thread associated with each session. In order to dispatch requests, this thread constantly obtains and dispatches work items from a work queue. In order for requests to be placed in the right queue, it is necessary to create a separate POA for each session. Work queues are an Orbix enhancement to the CORBA specification.



Generally, the thread-per-session model is used for servants that have code that is not thread-safe. In this case, in order to remove any ORB proprietary extensions, the servant code needs to be made thread-safe so the POA can use the `ORB_CTRL_MODEL` policy.

## Event Handlers

Event handlers are generally used for two main purposes:

- ❑ Connection management
- ❑ Session management

### Connection Management

The number of TCP/IP connections that can be made to a single process is typically subject to an operating system limit. Some form of connection management is required if this limit is likely to be reached in a deployed system.

Orbix provides an active connection manager (ACM) that allows the ORB to reclaim connections automatically, and thereby increases the number of clients that can use a server beyond the limit of available file descriptors.

### Session Management


Event handlers can be used to implement an elementary session-tracking mechanism. The opening of a connection from a client defines the beginning of a session, and the closing of the connection defines the end of the session.

Support for session management in Orbix is provided by the *leasing plugin*. This plugin implements a scheme for automatically tracking client sessions, based on the idea that a client obtains a lease from the server for the duration of a client session, and renews the lease periodically. If the lease is never renewed the server determines that the client has ended and ends the client session.

## Interceptors and Object Wrappers

Interceptors are a VisiBroker proprietary mechanism used to intercept invocations on both the server and client sides. CORBA 2.4 introduced similar functionality, known as *Portable Interceptors*. There are a few differences with respect to VisiBroker's interceptors:

- > There is no direct equivalent to `BindInterceptor` because the `_bind` calls are not CORBA compliant.
- > For the client and server interceptors, although the member functions have the same or similar names, the APIs are completely different from the VisiBroker ones.

- 
- > Instead of interceptor factories, the interceptors are registered by way of the `PortableServer::ORBInitializer` interface.
  - > Portable interceptors provide a way to add additional information through service contexts. It also provides the capability to add tagged components to IORs through IOR interceptors.

Object wrappers are very similar to interceptors but are invoked at the stub and skeleton level, rather than at the ORB level. The client object wrapper `pre_method` and `post_method` can be replaced by the client portable interceptor `send_request` and `receive_reply`, respectively. Likewise, the server object wrapper `pre_method` and `post_method` can be replaced by `receive_request` and `send_reply`, respectively.

### **Smart Stubs**

The VisiBroker smart stubs feature is a proprietary mechanism for overriding the default implementation of the proxy class. This allows applications to intercept outbound client invocations and handle them within the local client process address space, rather than using the default proxy behavior of making a remote invocation on the target object. Smart stubs can be used for such purposes as client-side caching, logging, load balancing, or fault-tolerance.

The CORBA specification does not support functionality equivalent to smart stubs. The primary difficulty is that, in the general case, it is not possible for the client-side ORB to determine if two object references denote the same server object. The CORBA standard restricts the client-side ORB from interpreting the object key or making any assumptions about it. VisiBroker was able to avoid this limitation by making assumptions about the structure of the object key. This is neither CORBA-compliant nor interoperable with other ORBs.

At best, the ORB can only determine that two object references are equivalent if they have exactly the same server location (host and port in IIOP) and object key. Unfortunately, this can be an unreliable indicator if object references pass through bridges, concentrators, or firewalls that change the server location or object key.

In this case, it is possible for two object references denoting the same CORBA object to appear different to the ORB, and thus have two different smart proxy instances. Since smart proxies are commonly used for caching, having two smart proxy instances for a single CORBA object is unacceptable.

The best option to replace smart stubs is to replace them with wrapper objects that embed the proxy objects and that implement the same functionality as smart stubs.



## Location Service

VisiBroker's Location Service is a proprietary mechanism for obtaining location information about active object instances. In Orbix there is no direct equivalent to the Location Service because Orbix deals with POAs rather than actual object instances. However, Orbix provides IDL interfaces to access the Location Domain in order to obtain information on active POAs.

The Location Service also offers a callback mechanism, called *trigger*, which is used to inform of changes in the availability of a specific object instance. This is typically used for recovery when the connection to an object is lost. Orbix provides similar functionality with the *leasing plugin*. Please refer to subsection 8.5.2 for details.

## ORB Manager Interfaces

With the VisiBroker ORB management interface, client applications can monitor and manage the properties of object servers. These attributes correspond to ORB and BOA configuration options.


In Orbix, client and server configuration is specified through the configuration and can be overridden through command-line arguments. It is also possible to create Management Beans (Mbeans) in the servers. These integrate with the Orbix Management Service, which allows you to remotely manage servers through a central console or using a Simple Network Management Protocol (SNMP) that facilitates integration with Enterprise Management Systems such as IBM Tivoli™, and HP OpenView™.

## AUTOMATING THE MIGRATION

Orbix provides a code generation utility, called `idlgen`, which allows the creation of a working application from the IDL code. Such an application can be used as a starting point for migrating from VisiBroker to Orbix.

The `idlgen` tool consists of an IDL parser that provides access to its internal parse tree through TCL. With this tool it is possible to write TCL script that traverse the parse tree and generate code based on the tree contents. Orbix comes with some pre-built TCL scripts, called *genies*, which allow the generation of entire applications. These are:

- > `cpp_poa_genie.tcl`, which generates C++ applications.
- > `java_poa_genie.tcl`, which generates Java applications.



The following are some usage examples of these genies: `idlgen`  
`cpp_poa_genie.tcl -all file.idl idlgen`  
`java_poa_genie.tcl -all file.idl`

These examples generate code for:


- > The build system: make files for C++ and ant `build.xml` for Java.
- > Client mainline
- > Server mainline
- > Servant classes
- > Printout classes to print IDL types
- > Random classes to randomly generate IDL types

The genies provide many more options. To view all available options use the command: `idlgen cpp_poa_genie.tcl -h`

To find out what genies are available, use the following command: `idlgen -list`

## REFERENCES

- [BSC98a] BORLAND SOFTWARE CORP. *VisiBroker for C++ v. 3.3 Programmer's Guide*. November 1998.
- [BSC98b] BORLAND SOFTWARE CORP. *VisiBroker for Java v. 3.3 Programmer's Guide*. November 1998.
- [BSC98c] BORLAND SOFTWARE CORP. *VisiBroker for C++ v. 3.3 Reference*. November 1998.
- [BSC98d] BORLAND SOFTWARE CORP. *VisiBroker for Java v. 3.3 Reference*. November 1998.
- [BSC01a] BORLAND SOFTWARE CORP. *VisiBroker for C++ v. 4.5 Programmer's Guide*. March 2001.
- [BSC01b] BORLAND SOFTWARE CORP. *VisiBroker for Java v. 4.5 Programmer's Guide*. March 2001.
- [BSC01c] BORLAND SOFTWARE CORP. *VisiBroker for C++ v. 4.5 Reference*. March 2001.
- [BSC01d] BORLAND SOFTWARE CORP. *VisiBroker for Java v. 4.5 Reference*. March 2001.
- [KB99a] IONA Technologies Knowledge Base. What do all the POA policies mean, and how can I choose the right POA policies for my



application? December 1999. URL:  
<http://www.iona.com/support/articles/2318.67.xml>

[KB00a] IONA Technologies Knowledge Base. A summary of the POA policies: meanings, defaults and Root POA values. April 2000. URL:  
<http://www.iona.com/support/articles/2466.874.xml>

[KB00b] IONA Technologies Knowledge Base. How do I use ORB::perform\_work() ? September 2000. URL:  
<http://www.iona.com/support/articles/2481.523.xml>

[KB00c] IONA Technologies Knowledge Base. What is a POAManager for? December 2000. URL:  
<http://www.iona.com/support/articles/2667.146.xml>

[KB00d] IONA Technologies Knowledge Base. What are the dangers of using \_this and implicit activation? October 2000. URL:  
<http://www.iona.com/support/articles/2319.93.xml>

[Henn99] HENNING, M.; VINOSKI, S.: *Advanced CORBA Programming with C++*, Addison Wesley Professional Computing Series. Addison Wesley. 1<sup>st</sup> Edition. February 1999.

[IONA00] IONA TECHNOLOGIES. *VisiBroker to Orbix (3) Migration: Technical Issues version 1.0*. June 2000.

[IONA03a] CORBA Migration and Interoperability Guide. IONA Technologies. December 2003.

[IONA03b] IONA Technologies. CORBA Session Management Guide. Orbix version 6.1. December 2003

[IONA03c] IONA Technologies. CORBA Programmer's Guide C++. Orbix version 6.1. December 2003

[IONA03d] IONA Technologies. CORBA Programmer's Reference C++. Orbix version 6.1. December 2003


[IONA03e] IONA Technologies. Installation Guide. Orbix version 6.1. December 2003

[IONA04a] IONA TECHNOLOGIES. *CORBA Utilities*. April 2004. URL:  
<http://www.iona.com/devcenter/corba/utilities.htm>

[OMG96] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 2.0. July 1996.

[OMG97] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 2.1. August 1997.

[OMG98] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 2.2. February 1998.



[OMG99] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 2.3. June 1999.

[OMG00] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 2.4. October 2000.

[OMG01a] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 2.5. September 2001.

[OMG01b] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 2.6. December 2001.



**Worldwide Headquarters**

Progress Software Corporation, 14 Oak Park, Bedford, MA 01730 USA

Tel: +1 781 280-4000 Fax: +1 781 280-4095

[www.progress.com](http://www.progress.com)

**For regional international office locations and contact information, please refer to [www.progress.com/worldwide](http://www.progress.com/worldwide)**

© Copyright 2008 Progress Software Corporation.

All rights reserved. Progress, Actional, Apama, DataXtend, Shadow, Sonic ESB, and SonicMQ are trademarks or registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and other countries. Any other trademarks contained herein are the property of their respective owners. Specifications subject to change without notice.

© Copyright 2008 IONA Technologies PLC, a wholly-owned subsidiary of Progress Software Corporation.

All rights reserved. IONA, IONA Technologies, the IONA logo, Orbix®, High Performance Integration, Artix®, FUSE™ and Making Software Work Together® are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that may appear herein are the property of their respective owners.