IBM WebSphere Business Integration Adapters

Connector Development Guide for C++

Note!
Before using this information and the product it supports, read the information in "Notices" on page 355.

20February2004

This edition of this document applies to IBM WebSphere InterChange Server, version 4.2.2, IBM WebSphere Business Integration Adapters, version 2.4, and to all subsequent releases and modifications until otherwise indicated in new editions.

To send us your comments about this document, email doc-comments@us.ibm.com. We look forward to hearing from you.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About this document

The IBM(R) WebSphere(R) Business Integration Adapters portfolio supplies integration connectivity for leading e-business technologies and enterprise applications. The system includes tools and templates for customizing, creating, and managing components for business process integration.

This document describes the development of C++ connectors in the IBM WebSphere business integration system.

Audience

This document is for connector developers. It assumes proficiency in the C++ programming language. The document also assumes a basic familiarity with the IBM WebSphere business integration system, including connectors and business objects.

Related documents

The complete set of documentation describes the features and components common to all WebSphere Business Integration Adapters installation, and includes reference material on specific components.

Note: This document covers the development of connectors written in C++. The development of Java connectors is documented in the *Connector Development Guide for Java*.

You can install the documentation or read it directly online at the following sites:

- For general adapter information, for using adapters with WebSphere message brokers (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker), and for using adapters with WebSphere Application Server, you can refer to the IBM WebSphere Business Integration Adapters InfoCenter at:

  `http://www.ibm.com/websphere/integration/wbiadapters/infocenter`
- For using adapters with IBM WebSphere InterChange Server as your integration broker, you can refer to the IBM WebSphere InterChange Server InfoCenter at:

  `http://www.ibm.com/websphere/integration/wicserver/infocenter`
- For more information about message brokers (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker):

  `http://www.ibm.com/sofware/integration/mqfamily/library/manualsa/`
- For more information about WebSphere Application Server:

  `http://www.ibm.com/sofware/webservers/appserver/library.html`

These sites contain simple directions for downloading, installing, and viewing the documentation.

Typographic conventions

This document uses the following conventions:

`courier font`	Indicates a literal value, such as a command name, file name, information that you type, or information that the system prints on the screen.
bold	Indicates a new term the first time that it appears.
italic, *`italic`*	Indicates a variable name or a cross-reference.
blue outline	A blue outline, which is visible only when you view the manual online, indicates a cross-reference hyperlink. Click inside the outline to jump to the object of the reference.
{ }	In a syntax line, curly braces surround a set of options from which you must choose one and only one.
`[ ]`	In a syntax line, square brackets surround an optional parameter.
`...`	In a syntax line, ellipses indicate a repetition of the previous parameter. For example, `option[,...]` means that you can enter multiple, comma-separated options.
`< >`	In a naming convention, angle brackets surround individual elements of a name to distinguish them from each other, as in `<server_name><`*`connector_name`*`>tmp.log`.
/, \	In this document, backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes. All WebSphere Business Integration Adapters product pathnames are relative to the directory where the product is installed on your system.
`%text%` and *`$text`*	Text within percent (%) signs indicates the value of the Windows `text` system variable or user variable. The equivalent notation in a UNIX environment is *`$text`*, indicating the value of the *`text`* UNIX environment variable.
ProductDir	Represents the directory where the product is installed. For the IBM WebSphere InterChange Server environment, the default product directory is "IBM\WebSphereICS". For the IBM WebSphere Business Integration Adapters environment, the default product directory is "WebSphereAdapters".

Markup conventions

In some chapters, you will find text identified by the following markup:

WebSphere InterChange Server
Describes functionality of the IBM WebSphere business integration system when InterChange Server is the integration broker.

WebSphere MQ Integrator Broker
Describes functionality of the IBM WebSphere business integration system when WebSphere MQ Integrator Broker is the integration broker.

New in this release

This chapter describes the new features of IBM WebSphere business integration systemthat are covered in this document.

New in WebSphere Business Integration Adapters v2.4.0

February 2004

With this update to the IBM WebSphere Business Integration Adapter 2.4.0 release, the *Connector Development Guide for C++* is provided *only* as part of the IBM WebSphere Business Integration Adapter documentation set. It is no longer provided as part of the IBM WebSphere InterChange Server documentation set.

In addition, information about how to create connector startup scripts has been enhanced. For more information, see "Creating startup scripts" on page 206.

December 2003

For a list of updates and new features for the IBM WebSphere Business Integration Adapter 2.4.0 release, please refer to "New in WebSphere InterChange Server v4.2.2 and WebSphere Business Integration Adapters v2.4.0."

New in WebSphere InterChange Server v4.2.2 and WebSphere Business Integration Adapters v2.4.0

The IBM WebSphere InterChange Server 4.2.2 release and the IBM WebSphere Business Integration Adapter 2.4.0 release provide the following new functionality in the C++ connector library:

- A C++ connector now uses the IBM Java Object Request Broker (ORB) instead of the third-party VisiBroker ORB.
- A C++ connector now uses STL libraries instead of the Cayenne libraries. You no longer need to include the `cayenne_include` directory in source files of your C++ connector.
- The C++ Connector Development Kit (CDK) now provides a more consistent way to create startup scripts for C++ connectors. It also provides a template for the creation of this startup script. For more information, see "Starting up a new connector" on page 205.

New in WebSphere InterChange Server v4.2.1 and WebSphere Business Integration Adapters 2.3.1

The IBM WebSphere InterChange Server 4.2.1 release and the IBM WebSphere Business Integration Adapter 2.3.1 release provide the following new functionality in the C++ connector library:

- The setLocale() method (in the `BusinessObject` class) allows you to set the locale that is associated with a business object. This new method complements the getLocale() method that has already been defined in this same class.

New in WebSphere Business Integration Adapters 2.2.0

The IBM WebSphere Business Integration Adapter 2.2.0 release provides the following new functionality in the C++ connector library:

- The ″CrossWorlds″ name is no longer used to describe an entire system or to modify the names of components or tools, with are otherwise mostly the same as before. For example ″CrossWorlds System Manager″ is now ″System Manager″ and ″CrossWorlds InterChange Server″ is now ″WebSphere InterChange Server″.
- The C++ connector library now supports duplicate event elimination to provide guaranteed event delivery. Duplicate event elimination is most often used by JMS-enabled adapters that have event stores that are *not* implemented as JMS queues. Use the `DuplicateEventElimination` connector property to enable this functionality. For more information, see “Guaranteed event delivery for connectors with non-JMS event stores” on page 131.
- Chapter 2, “Designing a connector,” on page 37 now provides more information on how to internationalize a connector.
- Chapter 8, “Adding a connector to the business integration system,” on page 199 now provides more information on how to add a C++ connector to the WebSphere business integration system, including:
  - How to create an initial configuration file for a connector
  - How to create a startup script for a C++ connector from a sample startup file
  - Use of the new `CWConnEnv.bat` (Windows) or `CWConnEnv.sh` (UNIX) file for system-variable settings
- The C++ connector library new supports two new methods in the `ReturnStatusDescriptor` class to provide access to the status value in the return-status descriptor:
  - `getStatus()`
  - `setStatus()`

New in WebSphere Business Integration Adapters 2.1

The changes made in the IBM WebSphere Business Integration Adapter 2.1 release do not affect the content of this document.

New in WebSphere Business Integration Adapters 2.0.1

The IBM WebSphere Business Integration Adapter 2.0.1 release provides an internationalized version of the C++ connector library. This internationalized connector library enables you to develop adapters that can be localized for many different locales (A locale includes culture-specific conventions and a character code set.). The structure of connectors has changed in the following ways to accommodate locales:

- The connector framework now has a locale associated with it. This locale is determined either from the operating system locale or from configuration properties. The C++ connector library provides the `getEncoding()` and `getLocale()` methods in the `GenGlobals` class to access this information from within the connector.
- A business object has a locale associated with it. This locale is associated with the data in the business object, *not* with the name of the business object definition or its attributes. The C++ connector library provides the `getLocale()` method in the `BusinessObject` class to obtain the name of this locale from within the connector.

For more information, see "An internationalized connector" on page 55.

New in WebSphere Business Integration Adapters 2.0

The IBM WebSphere Business Integration Adapter 2.0 release provides support for adapters. An *adapter* is a set of software modules that communicate with an integration broker and with applications or technologies to perform tasks such as executing application logic and exchanging data. For an introduction to adapters and integration brokers, see "Adapters in the WebSphere business integration system" on page 3.

In addition, the structure of IBM WebSphere business integration system documentation for the development of connectors has changed in this release. The following guides have been combined to create a single document that covers the development of C++ connectors:

Connector Development Guide	Material on how to develop a connector is now found in Parts I and II of this new document.
Connector Reference: C++ Class Library	Reference material on the C++ connector library is now found in Part III.

Part 1. Getting started

Chapter 1. Introduction to connector development

This chapter provides a brief overview of connectors in the IBM WebSphere business integration system. It also introduces the C++ Connector Development Kit (CDK) and summarizes the development steps you need to follow to implement a connector. This chapter contains the following sections:

Adapters in the WebSphere business integration system

The *IBM WebSphere business integration system* consists of the following components, which allow heterogeneous business applications to exchange data:

- A set of IBM WebSphere Business Integration Adapters

  An IBM WebSphere Business Integration Adapter, called simply an *adapter*, provides the components to support communication between an integration broker and either applications or technologies to perform tasks such as executing application logic and exchanging data.
- An integration broker

  The task of an *integration broker* is to integrate data among heterogeneous applications. The IBM WebSphere business integration system can include either of the integration brokers in Table 1..

Table 1. Integration brokers in the WebSphere business integration system

Integration broker	For more information	Documentation set
WebSphere message brokers (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, WebSphere Business Integration Message Broker)	*Implementing Adapters for WebSphere Message Brokers*	WebSphere Business Integration Adapters
WebSphere Application Server	*Implementing Adapters for WebSphere Application Server*	WebSphere Business Integration Adapters
IBM WebSphere InterChange Server (ICS)	*Implementation Guide for WebSphere InterChange Server*	WebSphere InterChange Server

In the IBM WebSphere business integration system, the integration broker communicates to these applications through adapters. The following adapter components actually provide this communication:

- "Business objects" on page 5, whose role is to hold information about an application event
- "Connectors" on page 6, whose role is to send information about an application event to an integration broker or to receive information about a request from the integration broker.

Figure 1 shows how these components transfer information from an application to an integration broker.

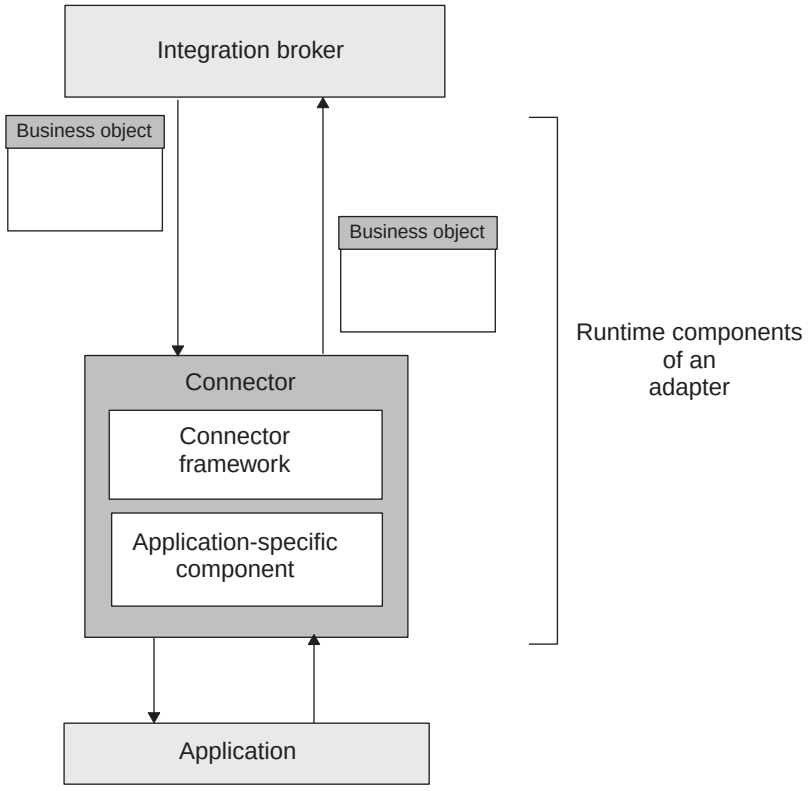

Figure 1. Adapter components that provide information transfer

Note: An adapter also includes configuration and development components. For more information, see "Tools for adapter development" on page 27.

Figure 2 shows the WebSphere business integration system and the role that connectors play within this system.

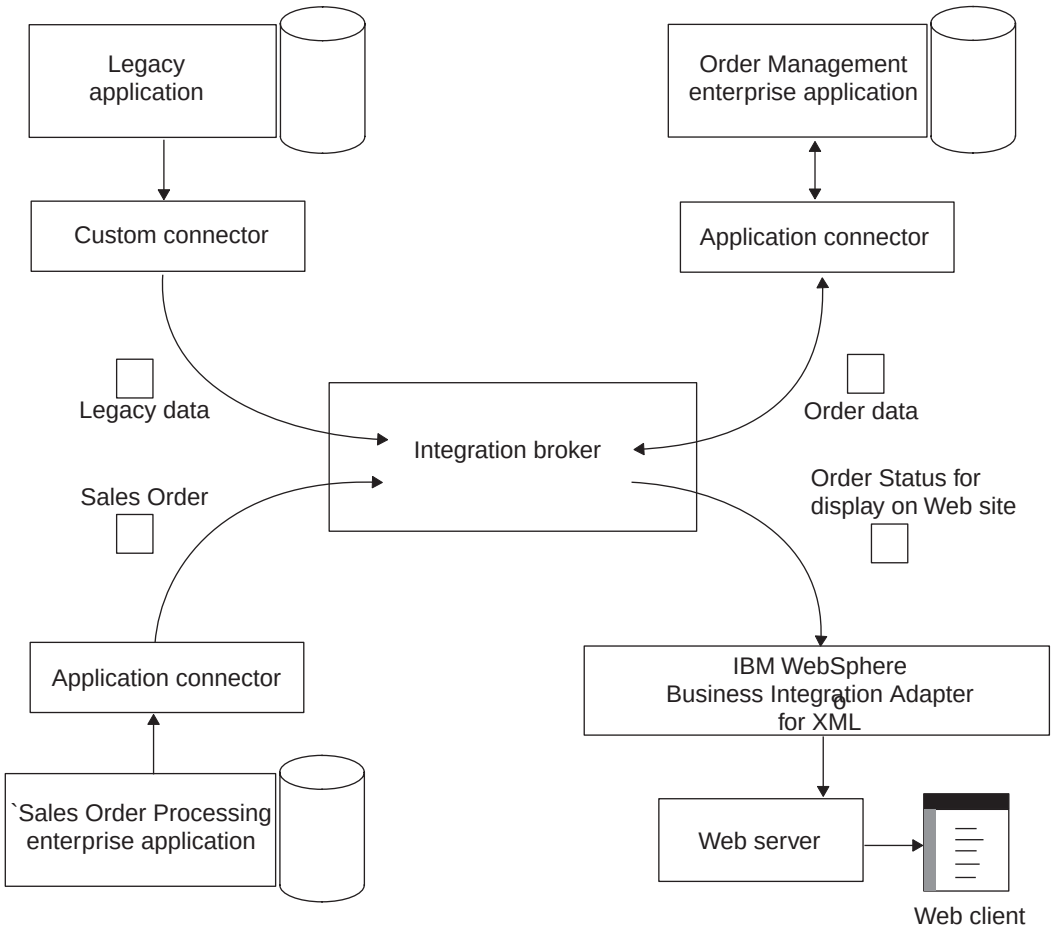

Figure 2. WebSphere business integration system

Business objects

As Table 2 shows, a business object is a two-part entity, consisting of a repository definition and a runtime object.

Table 2. Parts of a Business Object

Repository entity	Runtime object
Business object definition	Business object instance (often called a "business object")

Business object definition

A *business object definition* represents a group of attributes that can be treated as a collective unit. For example, a business object definition can represent an application entity and the operations that can be performed on the entity, such as create, retrieve, update, or delete. A business object definition can also represent other programmatic entities, such as the data contents of a business transaction form submitted from a Web browser. A business object definition contains *attributes* for each piece of data in the collective unit.

Note: For more information on the structure of a business object definition, see "Processing business objects" on page 100..

When you "develop a business object," you create a business object definition. You can create business objects definitions with the Business Object Designer tool, which provides an easy-to-use, graphical user interface (GUI) that allows you to define attributes of the business object. It supports saving the business object definition in the repository or in an external XML file.

Within Business Object Designer, you can create the business object definition in either of two ways:

- Manually, by using the dialogs of Business Object Designer to define attributes and other information for the business object definition.
- With an Object Discovery Agent (ODA), which automatically generates a business object definition by:
  - Examining specified entities within the application
  - "Discovering" the elements of these entities that correspond to business object attributes

Note: For information on how to use Business Object Designer to create business object definitions in either of these ways, see the *Business Object Development Guide*.

Business object instance

While the business object definition represents the collection of data, a business object instance (often just called a "*business object*") is the runtime entity that contains the actual data. For example, to represent a customer entity in your application, you can create a Customer business object definition that defines the information in the customer entity that needs to be sent to other applications. At runtime, the Customer business object, which is an instance of this business object definition, contains the information for a particular customer.

Connectors

The role of a *connector* is to send information about an application event to an integration broker or to receive information about a request from the integration broker.

WebSphere InterChange Server

When InterChange Server is the integration broker, a connector is a set of software modules and data maps that connect WebSphere Business Integration collaborations to an enterprise application or an external application. A *collaboration* represents a business process that can involve several applications. The connector acts as an intermediary for one or more collaborations, using an enterprise application's API, or some other program logic, to support a business process.

The information that the connector sends or receives is in the form of a business object. Therefore, each connector supports one or more business object definitions. These business object definitions have been designed to correspond to application data models or to other types of external entities. The business object closely reflects the data entity that it represents. Its structure and content match that of the entity.

WebSphere InterChange Server

When InterChange Server is the integration broker, the business integration system uses two kinds of business objects. The business object that a connector processes is called an *application-specific business object*. The business object that a collaboration processes is called a *generic business object*. For more information, see "Mapping services" on page 12..

Other integrator brokers

When a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Intregration Message Broker) or WebSphere Application Server is the integration broker, the business integration system uses a single kind of business object, the business object that a connector processes. Although this business object is an application-specific business object, the "application-specific" qualifier is often omitted because this is the only kind of business object used.

The connector uses information in its supported business object definitions to perform its major roles, as Table 3 shows.

Table 3. Operations on business objects for the different roles of a connector

Connector role	Operation on business object
"Event notification" on page 22	When an event that affects an application entity occurs (such as when a user of the application creates, updates, or deletes application data), a connector: • Creates a business object, based on the information in its business object definition • Fills this business object with data from an application entity • Sends this business object as an event to an integration broker
"Request processing" on page 25	When the integration broker requests a change to the connector's application or when the broker needs information from the connector's application, the connector: • Receives a business object from an integration broker • Uses information in the business object and its business object definition to create the appropriate application command that performs an operation • Sends any appropriate response information back to the integration broker

Note: Every connector *must* implement request processing. Implementation of event notification is optional (though does require some minor coding).

Connector components

The connector represents the application in the WebSphere business integration system, performing tasks in support of the application. For example, the connector polls the application for events and sends business objects that represent events to the integration broker. The connector also performs tasks in support of integration-broker requests, such as retrieving or modifying application data.

Figure 3 illustrates the components of a C++ connector. The figure includes the C++ translation layer, which translates business objects between the C++ and Java environments. The figure also shows the C++ connector library in the generic services that the connector framework provides.

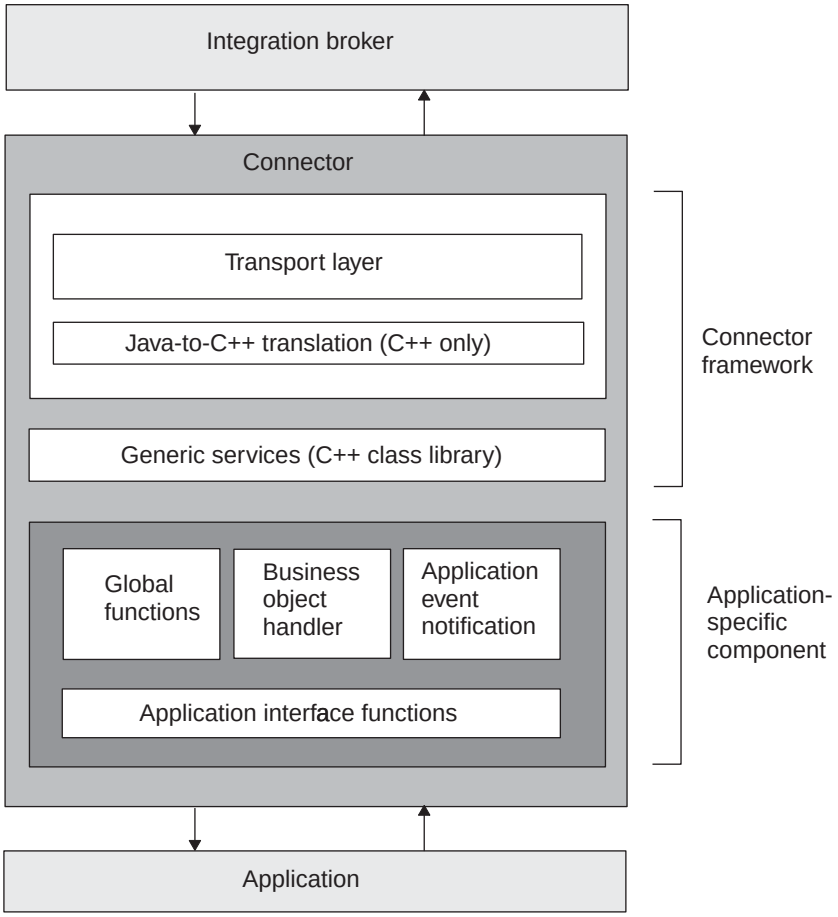

Figure 3. Components of a C++ connector

As Figure 3 shows, a connector has the following components:

- "Connector framework"—Provided as part of the WebSphere Business Integration Adapters product to communicate with the integration broker.
- "Application-specific component" on page 20—Contains code you write to specify the actions of the application-specific tasks of the connector, such as basic initialization and setup methods, business object handling, and event notification.

Connector framework

The connector framework manages interactions between the connector and the integration broker. IBM provides this component to ease connector development. The connector framework is written in Java and includes a C++ extension to allow the development of the application-specific component in C++.

Other integration brokers

In an IBM WebSphere business integration system that uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker), or WebSphere Application Server as its integration broker, the connector framework is a nondistributed component; that is, it resides entirely on the adapter machine. Figure 4 shows the high-level connector architecture with the WebSphere message broker or WebSphere Application Server. For information on the connector architecture with InterChange Server as the integration broker, see "Connector controller" on page 10..

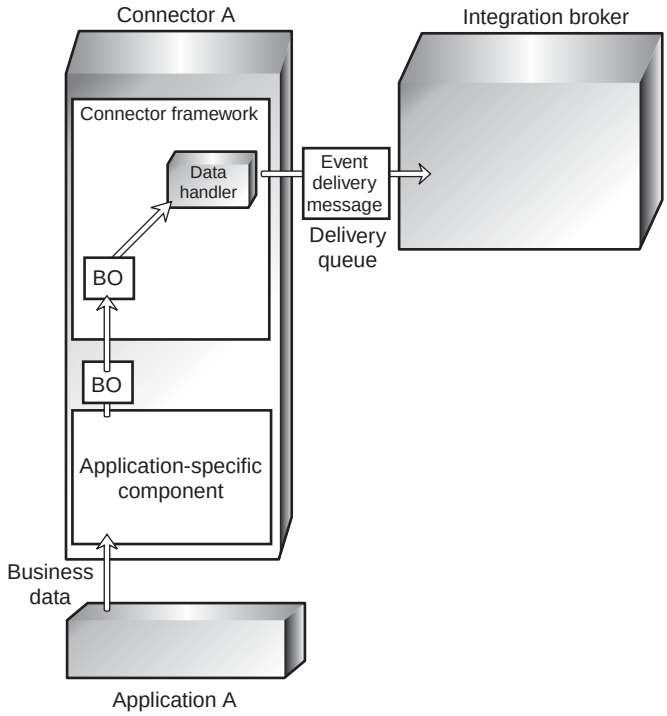

Figure 4. High-level connector architecture with a WebSphere message broker

The connector framework provides the services that Table 4 summarizes.

Table 4. Services of the connector framework

Component	Services
"Connector controller" on page 10 (InterChange Server only)	• Provides mapping between application-specific and generic business objects, and manages business object transfers between the connector and collaborations running in InterChange Server. • Provides other management services, such as monitoring the status of the connector

Table 4. Services of the connector framework (continued)

Component	Services
"Transport layer" on page 14	• Handles the exchange of business objects between the connector and the integration broker • Manages the exchange of startup and administrative messages between the connector controller and the client connector framework • Keeps a list of subscribed business objects
C++ translation layer	• Provides a Java-to-C++ translation layer that translates business objects between the C++ and Java environments.
C++ connector library on page "C++ connector library" on page 19.	• Provides generic services to the application-specific component in the form of C++ classes and methods

Connector controller

In an IBM WebSphere business integration system that uses InterChange Server as its integration broker, the connector framework is distributed to take advantage of services that InterChange Server provides. This distributed connector framework contains the following components:

- The *client connector framework* runs as part of the connector process on the client machine. It includes a transport layer, the C++ translation layer, and the C++ connector library. For more information on these components, see Table 4 on page 9..
- The *connector controller* runs within InterChange Server on the server machine.

Figure 5 illustrates the basic components of a connector within the InterChange Server system. InterChange Server, collaborations, and connector controllers run as a single process, and each connector runs as a separate process.

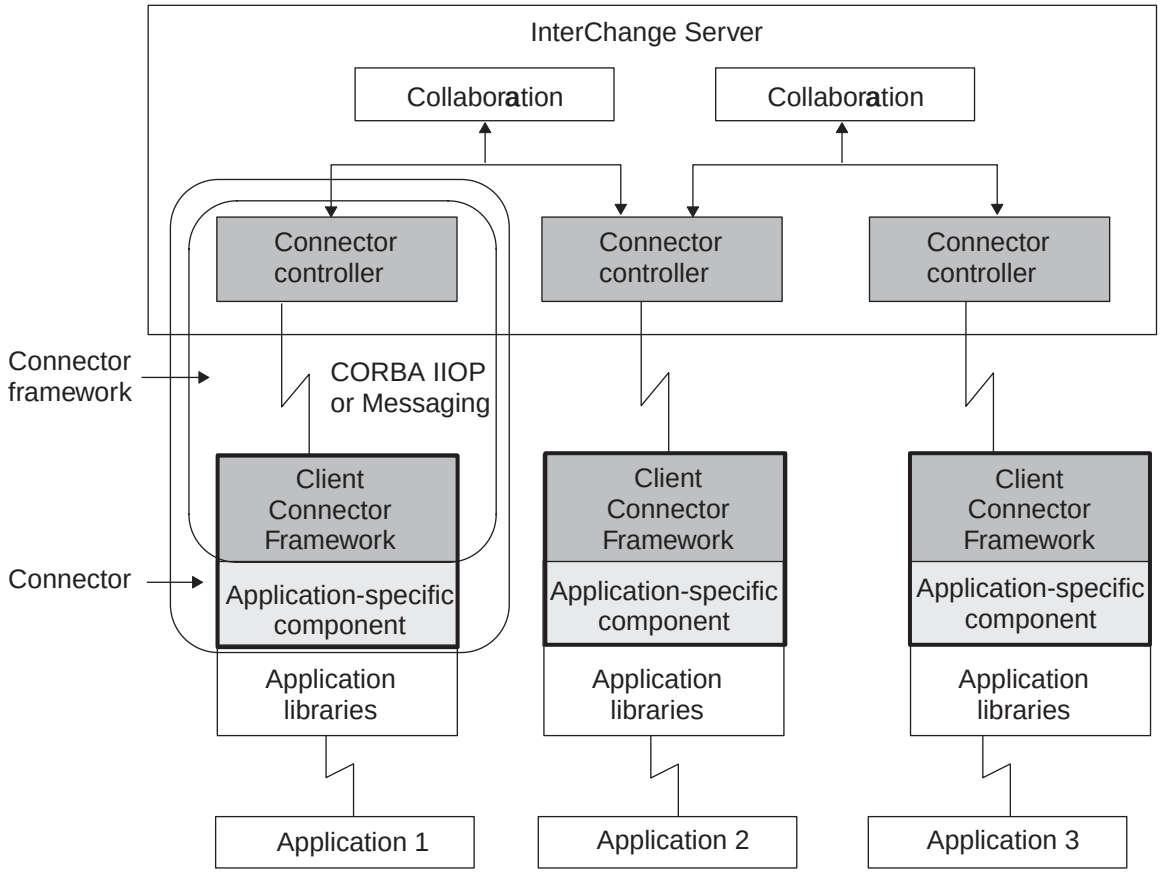

Figure 5. High-level connector architecture with WebSphere InterChange Server

The connector controller manages communication between the connector framework and collaborations. The primary type of information that connector components exchange is a business object. Other types of connector communication include startup information and administrative messages.

Note: A connector controller is instantiated by InterChange Server for each connector that has been defined in the InterChange Server repository. You do *not* need to provide code for the connector controller, as this component is internal to InterChange Server.

In addition to the features that the client connector framework provides, the connector controller provides the services that Table 5 summarizes.

Table 5. Services of the connector controller

Connector controller service	Description
"Mapping services" on page 12	The connector controller calls the map associated with each business object to transfer data between generic business objects and application-specific business objects.
"Business object subscription and publishing" on page 13	The connector controller manages collaboration subscriptions to business object definitions. It also manages connector queries about subscription status for a business object.

Table 5. Services of the connector controller (continued)

Connector controller service	Description
Service call requests (For more information, see "Initiating a request with InterChange Server" on page 25.)	The connector controller delivers collaboration service call requests to connectors. It also accepts return status messages and business objects from the connector and forwards them to InterChange Server.
Communication between components (For more information, see "Transport mechanism with InterChange Server" on page 15.)	The connector controller contains a transport driver to handle its side of the mechanism for exchanging business objects and administrative messages between the connector controller and client connector framework. It also performs remote-end synchronization to manages high-level synchronization between itself and the client connector framework. These services enable the connector controller to communicate with the connector, which might be installed remotely.

Note: The connector controller handles its own internal errors as well as errors from the client connector framework. In general, the connector controller logs exceptions and then evaluates whether further action is needed. When status messages are returned by the client connector framework, the connector controller forwards the messages to the collaboration.

Mapping services: The client connector framework sends and receives information in an application-specific business object. However, a collaboration generates information in a generic business object. Because application-specific business objects can differ from generic business objects, the InterChange Server system must convert business objects from one form to another so that data can be transmitted across the system. Data is transformed between generic and application-specific business objects by data *mapping*.

Data mapping converts business objects from generic to application-specific and from application-specific to generic forms. An application-specific business object closely reflects the data entity that it represents. Its structure and content match that of the entity. A generic business object, on the other hand, typically contains a superset of attributes that represents a typical, cross-application view of an entity's data. This type of business object is a composite of common information that many applications have about a particular entity. A generic business object serves as an intermediate point between data models.

Mapping is initiated by the connector and executed at runtime. For example, when a connector needs to map an application-specific business object to a generic business object, it runs an associated map to transfer data between the application-specific business object and the generic business object before sending the generic business object to a collaboration.

Mapping is handled by the connector controller. Figure 6 illustrates the connector in the InterChange Server system and shows the components of the connector.

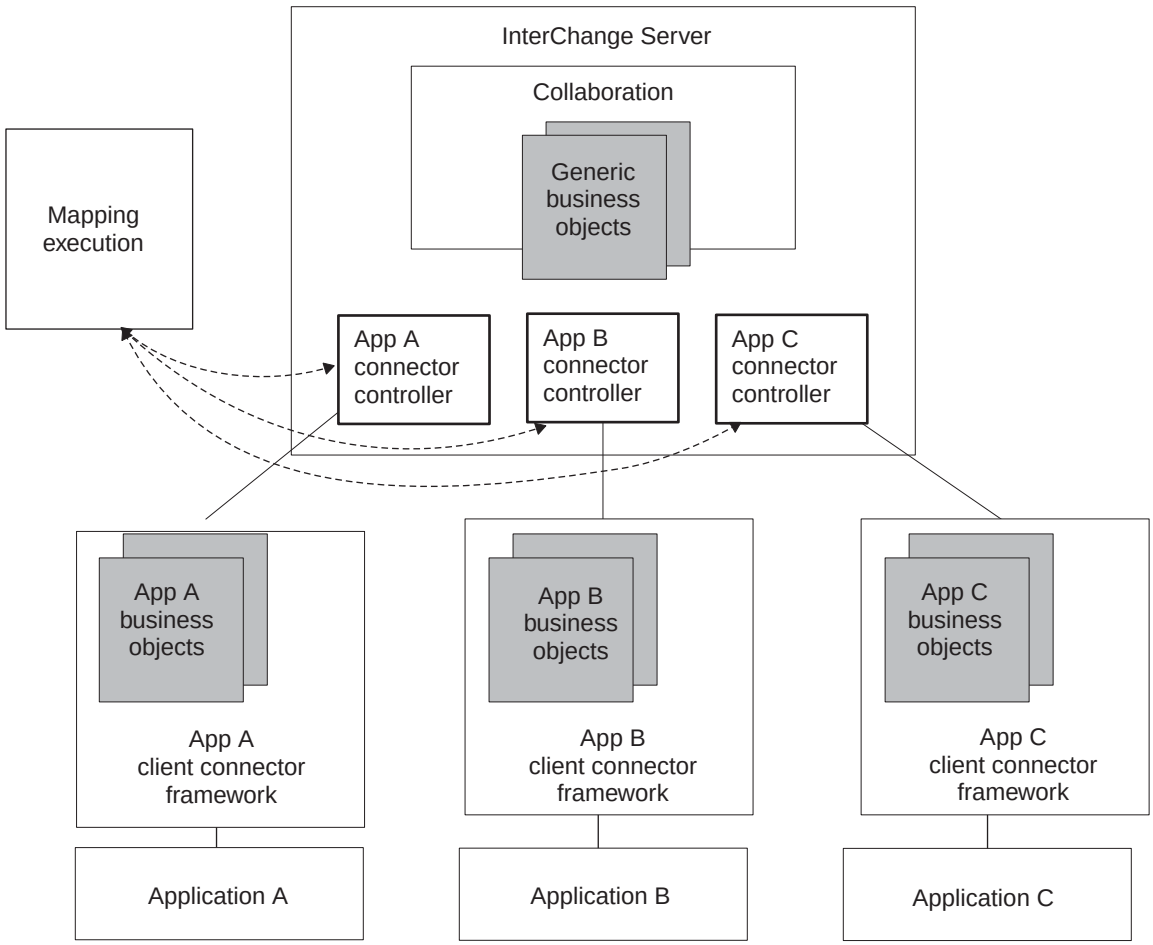

Figure 6. Mapping in the InterChange Server System

For more information on data mapping, refer to the *Map Development Guide* in the IBM WebSphere InterChange Server documentation set.

Business object subscription and publishing: Subscription handling is managed through a *subscription list*, which is a list of business objects to which collaborations have subscribed. Both the connector framework and the connector controller maintain a subscription list, as follows:

- The connector controller maintains a list of business objects to which collaborations have subscribed.

  When collaborations start, they subscribe to the business objects that they are interested in by informing the connector controller of their interest. The connector controller stores this information in a subscription list, which contains the name of the subscribing collaboration and the business object definition name and verb.

  When the connector controller receives a business object from the client connector framework, it checks its own subscription list to determine which collaborations have subscribed to this type of business object. It then forwards the business object to the subscribing collaboration.
- The connector framework also maintains a list of business objects to which collaborations have subscribed. However, this subscription list is a consolidated version of the connector controller's subscription list.

At initialization, the connector downloads its business object definitions and configuration properties from the InterChange Server repository. It also requests the subscription list from the connector controller. The subscription list that the connector controller sends to the client connector framework contains only the names of the business object definitions and verbs for these subscribed business objects. The connector framework stores this subscription list locally. Whenever a new collaboration starts up and subscribes to a business object, the connector controller notifies the connector framework so that the local subscription list is kept current.

As part of the initialization of the client connector framework, the connector framework instantiates a subscription manager. The *subscription manager* tracks all subscribe and unsubscribe messages that arrive from the connector controller and maintains a list of active business object subscriptions. Through the subscription manager, the application-specific connector component can query the connector framework to find out whether any collaborations are interested in a particular kind of business object.

Figure 7 illustrates the connector architecture for subscription handling.

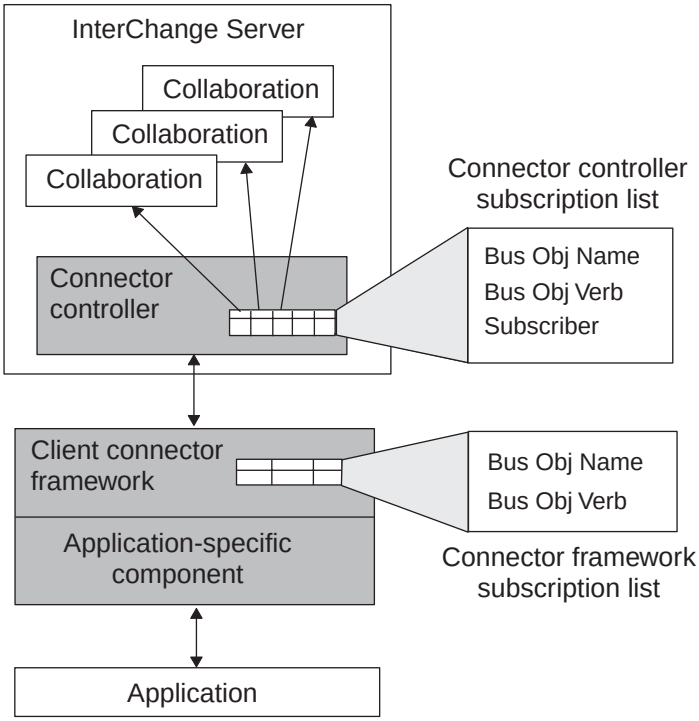

Figure 7. Subscription handling

For more information on subscriptions, see "Request processing" on page 25.

Transport layer

The transport layer of the connector framework handles the exchange of information between the connector and the integration broker. The transport layer of the connector framework provides the following services:

- Receives business objects from the integration broker and sends business objects to the integration broker:

Message service	Description
"Request processing" on page 25	Receives a business object from the integration broker and sends it to the application-specific component of the connector
"Event notification" on page 22	Receives a business object from the application-specific component of the connector and sends it to the integration broker

- Manages the exchange of startup and administrative messages between the connector and the integration broker.
- Keeps a list of business objects that are subscribed to

The transport mechanism of the transport layer depends on the integration broker in your business integration system:

- "Transport mechanism with InterChange Server"
- "Transport mechanism with other integration brokers" on page 18

Transport mechanism with InterChange Server: If the integration broker is InterChange Server (ICS), the transport layer handles the exchange of information between the connector controller, which resides within ICS, and the client connector framework.

Note: For more information, see "Connector controller" on page 10.

As Figure 8 shows, the transport layer for a connector that communicates with InterChange Server might include two transport drivers, one for the Common Object Request Broker (CORBA) and one for some message-oriented middleware (MOM).

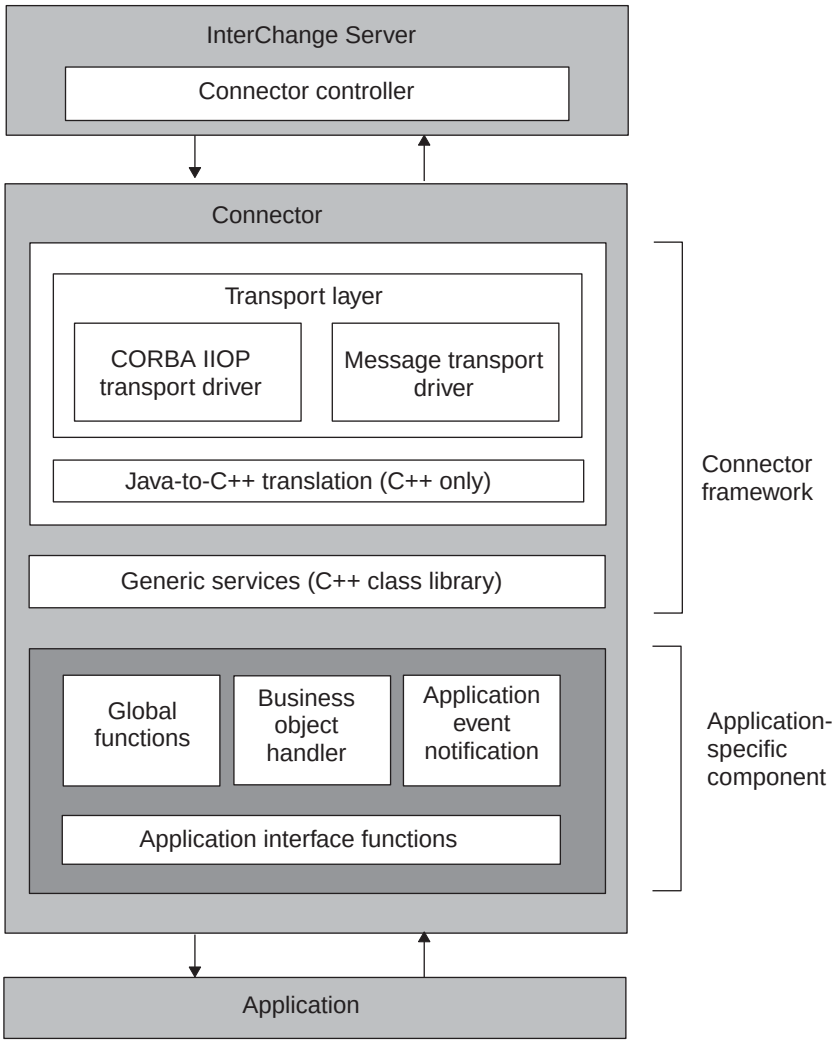

Figure 8. Connector architecture for communicating with InterChange Server

Table 6 summarizes the tasks that the transport layer performs and the transport mechanisms it can use.

Table 6. Tasks of the transport layer

Transport-layer task	Transport mechanism
Connector startup and exchange of startup messages between the connector controller and the client connector framework	CORBA
Administrative messages about the state of the client connector framework	CORBA
Sending business objects to the connector, initiated with a collaboration service call request	CORBA
Sending business objects from the connector, initiated with an event delivery	CORBA A message-oriented middleware system, including one of the following: • WebSphere MQ • Java Messaging Service (JMS)

This transport mechanism has the following tasks:

- At connector startup, the transport layer uses the Common Object Request Broker Architecture (CORBA) to transfer information from InterChange Server to the memory of the connector process.

  In the CORBA architecture, objects communicate through the Object Request Broker (ORB). The ORB is a set of libraries and services that connects an object, such as a connector controller, with another object, such as a client connector framework. The ORB enables objects to find each other at startup and to invoke methods on each other at runtime.

  With the ORB, the CORBA architecture provides a Naming Service that allows an object on the ORB to locate another object by name. At startup, the client connector framework uses the Naming Service to connect to the InterChange Server. The client connector framework then uses the ORB to request its application-specific connector configuration properties and its list of supported business object definitions from the repository. For more information, see "Starting up a connector" on page 63..

  Once the client connector framework and connector controller are active and connected, the client connector framework requests its list of business object subscriptions. At this point, connector initialization is complete, and the connector starts polling for events.
- For administrative messages about the state of the connector, the transport layer uses CORBA to send and receive state information for the connector controller.

  Changes in state of the client connector framework can be initiated from System Manager in the WebSphere Business Integration Toolset. Such changes include start, stop, pause, and resume operations, as well as retrieving the status. In addition, administrative messages can specify remote message logging.
- For sending business objects to the connector, initiated with a collaboration service call request, the transport layer also uses CORBA.

  CORBA technology includes the Internet Inter-ORB Protocol (IIOP) transport protocol. CORBA IIOP provides a lightweight, high-performance, synchronous communication mechanism that the connector controller and the client connector framework use to interact. Because the IIOP communication mechanism is synchronous, connector components can quickly determine whether a business object exchange was successful and can take appropriate action if necessary.
- For sending business objects from the connector, initiated with an event delivery, the connector can be configured to use either CORBA or a message-oriented middleware (MOM) system.

  When CORBA is used for business object subscription delivery, multiple business objects can be delivered concurrently, improving performance for subscription delivery. Using CORBA as a communication mechanism provides particularly good performance on a high-bandwidth LAN network.

  A messaging system provides asynchronous message delivery across a network, enabling connector components to send a message and continue processing without waiting for a response. The messaging system also provides persistent messaging, allowing the connector controller and client connector framework to operate independently.

  Note: In this case, connector components continue to use CORBA for startup and administrative messages.

  In the messaging communication mechanism, message transport is handled by transport drivers in the client connector framework and the connector controller. The message transport driver implements the low-level mechanism for exchanging data between InterChange Server and the underlying message

queuing software. Messages between the components of the connector are transported in a format defined by the messaging software.

This business integration system uses CORBA technology provided by the IBM Object Request Broker (ORB). Figure 9 illustrates the CORBA communication mechanism.

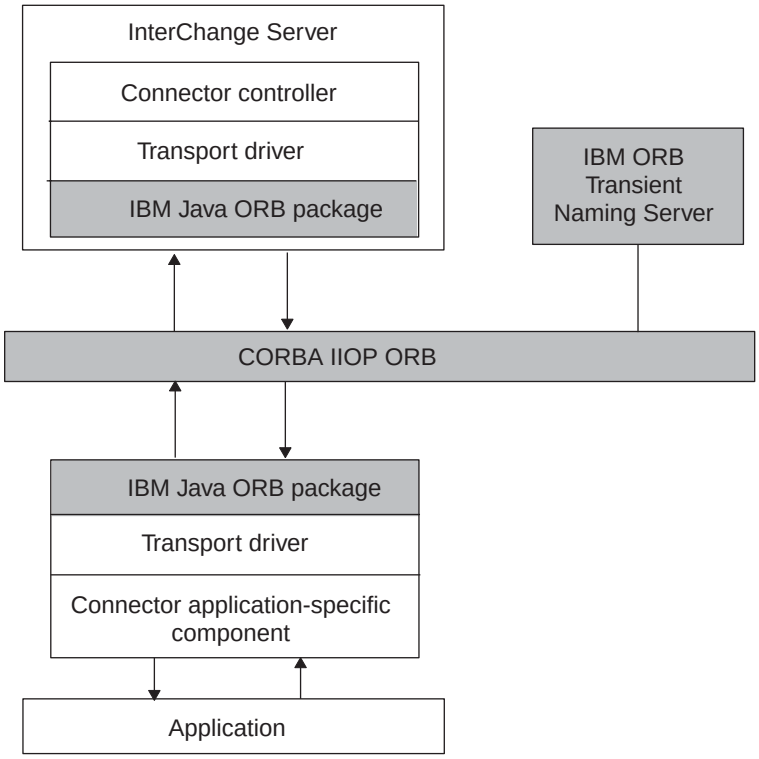

Figure 9. Communication within a connector using CORBA IIOP

Supported message-oriented middleware includes:

- IBM WebSphere MQ messaging suite. In this system, each active connector requires one unidirectional message queue. WebSphere MQ manages the queue using a queue manager. In this business integration system, each InterChange Server has one queue manager for all system components.
- Java Messaging Service (JMS)

Note: To configure a connector's transport mechanism for event delivery, set the `DeliveryTransport` standard property. For more information on this property, see Appendix A, "Standard configuration properties for connectors," on page 313

Transport mechanism with other integration brokers: If the integration broker is a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the transport layer handles the exchange of information between the connector framework and the integration broker. The transport layer for a connector that communicates with the broker includes a single transport driver for the IBM WebSphere MQ messaging suite. Data is exchanged between applications by means of application-specific business objects, which are transported between the connector framework and the integration broker as

WebSphere MQ messages. The integration broker removes the message from the MQ queue, and passes it through the message flow for the queue.

This transport mechanism uses WebSphere MQ messages to perform the following tasks:

- For sending business objects *to* the connector, which initiates request processing, the transport layer converts the business object to an MQ message and puts this message onto the appropriate WebSphere MQ queue.
- For sending business objects *from* the connector, which initiates an event delivery, the transport layer takes the MQ message off the appropriate WebSphere MQ queue and converts it to an application-specific business object.

The connector framework uses a custom data handler to transform the application-specific business object to and from an MQ message of the appropriate wire format for the destination WebSphere MQ queue.

For more detailed information on the use of MQ messages and a connector, see the implementation guide for your integration broker.

C++ connector library

The connector framework includes the C++ connector library, which provides generic services and utilities for connector development. The primary services provided by the C++ connector library are:

- Business object definition directory – Manages access to the business object definitions supported by a connector. Business object definitions are cached to improve connector performance in a distributed environment.
- Business object class – Provides methods for processing application information. This class allows the connector to handle application data in an object-oriented manner.
- Subscription manager – Enables the connector to check whether any collaborations are interested in a particular kind of business object.
- Logging utility – Enables the connector to post messages to the connector's standard output. Functionality includes configurable output destination and allows assigning error levels for all logged messages.
- Tracing utility – Enables the connector to generate trace messages for debugging purposes.

Note: For a summary of the C++ connector library and its classes, see Chapter 9, "Overview of the C++ connector library," on page 217Chapter 9, "Overview of the C++ connector library," on page 237.

The C++ connector library is available on both Windows and UNIX operating systems to support the execution of C++ connectors:

- For Windows systems, the C++ connector library is a dynamic link library (DLL) called `CwConnector.dll`. It resides in the following directory:

  ProductDir`\bin`

  Development versions of this library are included in the Connector Development Kit for C++ (CDK).

  Important: The CDK is supported *only* on Windows systems. For more information, see "Connector Development Kit" on page 29.

- For UNIX-based systems, the C++ connector library is a shared library called `libCwConnector`. The file extension depends on the particular UNIX system you are using. This shared library file resides in the following directory:

  ProductDir`/lib`

Because Java is operating-system-independent, the Java connector library is available on all systems that the WebSphere Business Integration Adapters product supports

Application-specific component

The application-specific component of the connector contains code tailored to a particular application. This is the part of the connector that you design and code. The application-specific component includes:

- A connector base class to initialize and set up the connector
- A business object handler to respond to request business objects initialized by integration-broker requests
- If needed, an event notification mechanism to detect and respond to application events.

You develop your code for the application-specific component to use services provided by the connector framework. The connector class library provides access to these services. You can write your connector code in C++ or Java depending on the application programming interface (API) provided by the application.

If the application API is written in C or C++, you write the application-specific portion of the connector in C++, accessing services of the connector framework through the C++ connector library. At runtime, the application-specific component is invoked from a Java class in the connector framework.

Event-triggered flow

The C++ connector library contain an API that allows a user-defined application-specific component to communicate with an integration broker through business objects. Applications can exchange information with other applications that the integration broker handles.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector can communicates with other applications through executing a collaboration. A *collaboration* represents a business process that can involve several applications. A connector transforms data and logic into a business object that carries information about an event in the connector's application. The business object triggers a collaboration business process and provides the collaboration with information that it needs for the business process.

Note: An external process can also initiate execution of collaborations through a call-triggered flow. For more information, see the *Access Development Guide* in the IBM WebSphere InterChange Server documentation set.

WebSphere Message Brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker), the connector might request information from or send information to other applications through WebSphere MQ workflows. The MQ workflow routes the information as appropriate.

When an event occurs in the application, the connector's application-specific component creates a business object to represent this event and sends the event to the integration broker. An *application event* is any event that affects an entity associated with a business object definition. To send an event to an integration broker, the connector initiates an *event delivery*. This event contains a business object. Therefore, the flow trigger that a connector initiates is called an *event-triggered flow* (see Figure 10).

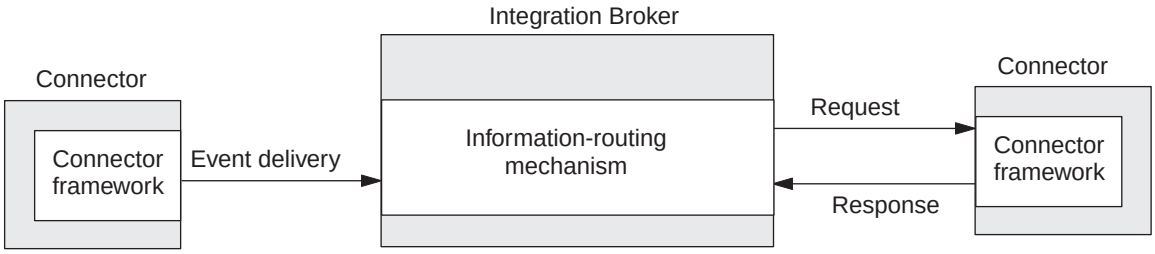

Figure 10. Event-triggered flow for WebSphere business integration system

Figure 10 shows event-triggering flow within the IBM WebSphere business integration system, which involves the following steps:

1. The connector creates the *triggering event*, which it sends to the integration broker during event delivery.

   When an event that affects an application entity occurs (such as when a user of the application creates, updates, or deletes application data), a connector creates a business object, which contains data from the application entity and a verb that indicates the operation performed on this data.
2. The application-specific component of the connector calls the `gotApplEvent()` method of the C++ connector library to send the triggering event to the connector framework. Through this method call, the connector performs an *event delivery*, which initiates the event-triggered flow.
3. The connector framework performs any needed conversion of the triggering event to a business object, then sends this event to the integration broker.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector controller receives the triggering event, performing any needed mapping of the application-specific business object data to the appropriate generic business object. The connector controller then sends the triggering event to the specified collaboration to trigger its execution. This collaboration is one that has subscribed to the business object that the event represents. The collaboration receives this business object in its incoming port.

4. The integration broker uses whatever logic it provides to route the event to the appropriate application. If it is so programmed, it might perform a *request*, routing the event information to the connector of some destination application, which would receive the event containing its request business object. In addition, this destination connector might send a request *response* back to the integration broker.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the collaboration might perform a *service call request* to send a business object to the connector controller of the destination connector, which is bound to its outgoing port. This connector controller performs any needed conversion from the resulting generic business object to the appropriate application-specific business object. It then performs a *service call response* to send the event response to the connector controller, which routes it back to the collaboration.

As Figure 10 shows, a connector can participate in one of two roles:

- "Event notification"—the connector sends an event (in the form of a business object) to the integration broker to notify it of some operation that has occurred in the application.
- "Request processing" on page 25—the connector receives a request business object from an integration broker.

Each of these connector roles is described in more detail in the following sections.

Event notification

One role of a connector is to detect changes to application business entities. When an event that affects an application entity occurs, such as when a user of the application creates, updates, or deletes application data, a connector sends an event to the integration broker. This *event* contains a business object and a verb. This role is called *event notification*.

This section provides the following information about event notification:

- "Publish-and-subscribe model"
- "Event-notification mechanism" on page 23

Publish-and-subscribe model

A connector assumes that the business integration system uses a *publish-and-subscribe model* to move information from an application to an integration broker for processing:

- An integration broker *subscribes* to a business object that represents an event in an application.

WebSphere InterChange Server

If your business integration system uses InterChange Server, a collaboration *subscribes* to a business object that represents an event in an application, and then the collaboration waits.

- A connector uses an event-notification mechanism to monitor when application events occur. When an application event does occur, the connector *publishes* a

notification of the event in the form of a business object and a verb. When the integration broker receives an event in the form of the business object that it has subscribed to, it can begin the associated business logic on this data.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector controller checks its own subscription list when it receives a business object from the connector framework to determine which any collaborations have subscribed to this type of business object. If so, it then forwards the business object to the subscribing collaboration. When a collaboration receives the subscribed event, it begins executing.

Event-notification mechanism

An *event-notification mechanism* enables a connector to determine when an entity within an application changes. When an event occurs in an application, the connector application-specific component processes the event, retrieves related application data, and returns the data to the integration broker in an business object.

Note: This section provides an introduction to event notification. For more information on how to implement an event-notification mechanism, see Chapter 5, "Event notification," on page 109.

The following steps outline the tasks of an event-notification mechanism:

1. An application performs an event and puts an event record into the event store.

   The *event store* is a persistent cache in the application where event records are saved until the connector can process them. The *event record* contains information about the change to an event store in the application. This information includes the data that has been created or changed, as well as the operation (such as create, delete, or update) that has been performed on the data.
2. The connector's application-specific component monitors the event store, usually through a polling mechanism, to check for incoming events. When it finds an event, it retrieves its event record from the event store and converts it into an application-specific business object with a verb.
3. Before sending the business object to the integration broker, the application-specific component can verify that the integration broker is interested in receiving the business object.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector framework does *not* assume that the integration broker is always interested in every supported business objects. At initialization, the connector framework requests its subscription list from the connector controller. At runtime, the application-specific component can query the connector framework to verify that some collaboration subscribes to a particular business object. The application-specific connector component can send the event *only* if some collaboration is currently subscribed. The application-specific component sends the event, in the form of a business object and a verb, to the connector framework, which in turn sends it to the connector controller within ICS. For more information, see "Mapping services" on page 12.

Other integration brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the connector framework assumes that the integration broker is interested in *all* the connector's supported business objects. If the application-specific connector component queries the connector framework to verify whether to send the business object, it will receive a confirmation for *every* business object that the connector supports.

4. If the integration broker is interested in the business object, the connector application-specific component sends the event, in the form of a business object and a verb, to the connector framework, which in turn sends it to the integration broker.

Figure 11 illustrates the components of the event-notification mechanism. In event notification, the flow of information is from the application to the connector and then to the integration broker.

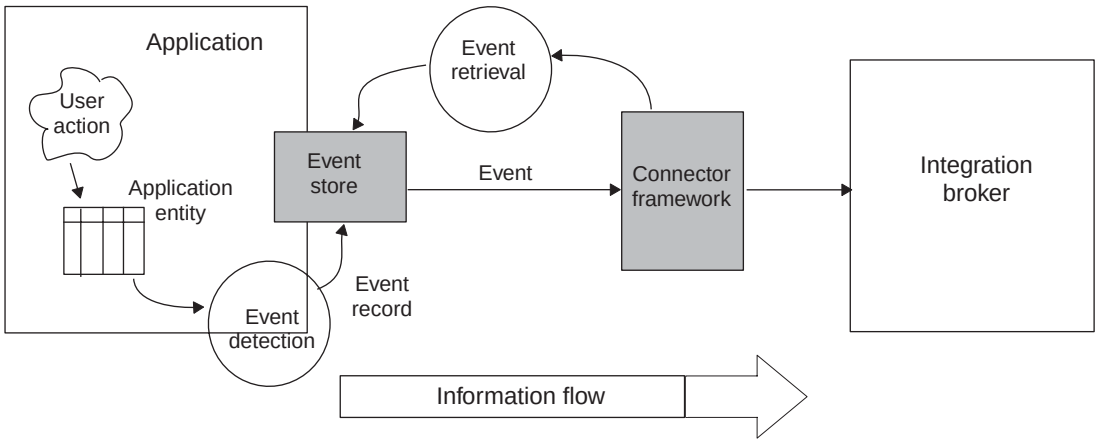

Figure 11. Event detection and retrieval

Request processing

In addition to detecting application events, another role of a connector is to respond to requests from the integration broker. A connector receives a *request business object* from a integration broker when the broker requests a change to the connector's application or needs information from the connector's application. In general, connectors perform create, retrieve, and update operations on application data in response to requests from a collaboration. Depending on the application's policies, the connector might also support delete operations. This role is called *request processing*.

WebSphere InterChange Server

If your business integration system uses InterChange Server, request processing can sometimes be called "service call request processing". The connector receives a business object from its connector controller, which receives it from a service call of a collaboration.

Note: This section provides an introduction to request processing. For more information on how to implement request processing in your connector, see Chapter 4, "Request processing," on page 73.

Request processing involves the following steps:

1. As Figure 10 on page 21 shows, an integration broker initiates request processing by sending a request to the connector framework. This request is in the form of a business object, called the *request business object*, and a verb. For more information, see "Initiating a request" on page 25.
2. The connector framework has the task of determining which *business object handler* in the application-specific component should process the request business object. For more information, see "Choosing a business object handler" on page 26.
3. The connector framework passes the request business object to the business object handler defined for it in its business object definition.

   The connector framework does this by calling the `doVerbFor()` method defined in the business object class and passing in the request business object. The business object handler then processes the business object, converting it to one or more application requests.
4. When the business object handler completes the interaction with the application, it returns a return-status descriptor and possibly a response business object to the connector framework. For more information, see "Handling a request response" on page 27.

Initiating a request

The way a request is initiated depends on the integration broker in your IBM WebSphere business integration system:

- "Initiating a request with InterChange Server"
- "Initiating a request with other integration brokers" on page 26

Initiating a request with InterChange Server: If your business integration system uses InterChange Server, the collaboration initiates a *service call request*, sending the request over one of its collaboration ports. When you bind a port of a collaboration object, you associate the port with a connector (or another collaboration object). Collaboration ports enable communication between bound entities, so that the

collaboration object can accept the business object that triggers its business processes, and then send and receive business objects as service call requests and responses.

Note: For more information on how to define collaboration ports, see the *Collaboration Development Guide*. For information on how to bind ports of a collaboration object, see the *Implementation Guide for WebSphere InterChange Server*. Both these documents are in the IBM WebSphere InterChange Server documentation set.

One the service call request is initiated, the InterChange Server system takes the following steps:

1. The connector controller for the connector bound to the collaboration port receives the service call request. If necessary, the connector controller maps the generic business object to an application-specific business object before sending the request to the connector framework.
2. The connector controller forwards the service call request to the connector framework. The connector controller sends the request business object as a C++ object.

Initiating a request with other integration brokers: If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the integration broker initiates a request by sending a message to the WebSphere MQ queue associated with the connector. One the request is initiated, the connector framework gets the WebSphere MQ message off using its transport layer and converts the message to the appropriate business object using a custom data handler.

For more information on the IBM WebSphere business integration system and request processing, see the implementation guide for your integration broker.

Choosing a business object handler

A *business object handler* is the Java class that is responsible for transforming the request business object into a request for the appropriate application operation. An application-specific component includes one or more business object handlers to perform tasks for the verbs in the connector's supported business objects. Depending on the active verb, a business object handler can insert the data associated with a business object into an application, update an object, retrieve the object, delete it, or perform another task.

Based on this response business object's business object definition, the connector framework obtains the correct business object handler for the associated business object:

- When the connector starts up, the connector framework receives from the connector controller the list of business objects that the connector supports.
- The connector framework calls the `getBOHandlerforBO()` method (defined in the connector base class) to instantiate one or more business object handlers.
- For each supported business object, the `getBOHandlerforBO()` method returns a reference to a business object handler, and this reference is stored in the business object definition in the memory of the connector process.

All conversions between business objects and application operations take place within the business object handler (or handlers).

For more information about how to implement the `getBOHandlerforBO()` method, see "Obtaining the business object handler" on page 66..

Handling a request response

Once a connector has processed this request and completed the interaction with the application, it can return a response to the integration broker.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector framework returns a *service call response* to the collaboration. Using information in the return-status descriptor, the collaboration can determine the state of its service call request and take appropriate actions.

Other integration brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the connector framework's response includes:

- A status indicator, which contains the information return-status descriptor
- Any business object messages, which contain the optional response business objects

The connector framework puts this response information onto the connector's queue. However, for the message transport to be synchronous (that is, for some program to wait for a response), a program must post its request message to the integration broker on a synchronous request queue and expect its response from the broker on a synchronous response queue. A correlation ID on the response message identifies the message request to which it is responding.

Tools for adapter development

In the IBM WebSphere business integration system, the *connector* is a component of a WebSphere Business Integration adapter. As discussed in "Adapters in the WebSphere business integration system" on page 3, an *adapter* includes runtime components to support communication between an integration broker and applications or technologies. The adapter also includes an *adapter framework*, which includes components for the configuration, runtime, and development of custom adapters in cases where a prebuilt adapter for a particular legacy or specialized application is not currently available as part of the WebSphere Business Integration Adapters product.

The adapter framework includes configuration tools that assist in the development of the adapter components listed in Table 7..

Table 7. Adapter framework support for the development of a connector

Adapter component	Configuration tool	API
Business object	Business Object Designer	Not applicable
Object Discovery Agent (ODA)	Business Object Designer	Object Discovery Agent Development Kit (ODK)
Connector	Connector Configurator	C++ Connector Library

In addition to the adapter framework, the WebSphere Business Integration Adapters product also provides the *Adapter Development Kit* (ADK). The ADK is a toolkit that provides code samples of connectors, ODAs, and data handlers. For more information, see "Adapter Development Kit" on page 29.

Development support for business objects

Table 8 shows the tools that the WebSphere Business Integration Adapters product provides to assist in the development of business objects.

Table 8. Development tools for business object development

Development tool	Description
Business Object Designer	Graphical tool that assists in the creation of business object definitions, either manually or through an ODA.

For a brief introduction to business objects, see "Business objects" on page 5. For more information on the use of the Business Object Designer, see the *Business Object Development Guide*.

Development support for ODAs

Table 8 shows the tools that the WebSphere Business Integration Adapters product provides to assist in the development of an ODA.

Table 9. Development tools for ODA development

Development tool	Description
Business Object Designer	Graphical tool that assists in the creation of business object definitions, either manually or through an ODA.
Object Discovery Agent Development Kit (ODK)	Set of Java classes with which you can create a custom ODA.

In addition, the ADK provides sample ODAs in the following product subdirectory:

```
DevelopmentKits\Odk
```

For a brief introduction to ODAs, see "Business objects" on page 5. For more information on the use of the Business Object Designer and the development of ODAs, see the *Business Object Development Guide*.

Development support for connectors

Table 10 shows the tools that the WebSphere Business Integration Adapters product provides to assist in the development of connectors.

Table 10. Development tools for connector development

Development tool	Description
Connector Configurator	Graphical tool that assists in the configuration of the connector
Adapter Development Kit	Includes sample code for C++ connectors and ODAs

The supported operating-system environment for connector development is Windows 2000. Connectors can be written in either C++ or Java, depending on the language of your application API.

Connector Configurator

Connector Configurator is a graphical tool that allows you to configure a connector. It provides the ability to set the following information:

- Connector configuration properties
- Supported business objects
- Associated maps (with InterChange Server only)
- Log and message files
- Data-handler configuration (for guaranteed event delivery)

This graphical tool runs on Windows 2000 and Windows XP. Therefore, these platforms are for connector configuration.

Note: For more information on the use of Connector Configurator, see Appendix B, "Connector Configurator," on page 329.

Adapter Development Kit

The Adapter Development Kit (ADK) provides files and samples to assist in the development of an adapter. It provides samples for many of the adapter components, including an Object Discovery Agent (ODA), a connector, and a data handler. The ADK provides these samples in the `DevelopmentKits` subdirectory of your product directory.

Note: The ADK is part of the WebSphere Business Integration Adapters product and it requires its own separate Installer. Therefore, to have access to the development samples in the ADK, you must have access to the WebSphere Business Integration Adapters product and install the ADK. Please note that the ADK is available *only* on Windows systems.

Table 11 lists the samples that the ADK provides for the development of a connector, as well as the subdirectory of the `DevelopmentKits` directory in which they reside.

Table 11. ADK samples for connector development

Adapter Development Kit component	Description	DevelopmentKits subdirectory
C++ Connector Development Kit (CDK)	Provides sample code for a C++ connector.	`cdk`

Connector Development Kit: The ADK includes the C++ Connector Development Kit (CDK), which provides components for use in the development of a connector. The components of the CDK reside in the following *ProductDir*`\DevelopmentKits` subdirectory:

```
DevelopmentKits\cdk
```

Table 12 describes the contents of the subdirectories in the cdk directory.

Table 12. Components of the Connector Development Kit

Connector Development Kit component	Description	Subdirectory
C++ connector library	Provides development versions of the C++ connector library, which you need to use when you create a C++ connector. For more information, see Chapter 9, "Overview of the C++ connector library," on page 217.Chapter 9, "Overview of the C++ connector library," on page 237.	`lib`
Code samples	Sample code for a simple C++ connector	`samples`
Connector-class header files	Provide definitions for the classes of the C++ connector library	`generic_include`

The CDK includes the following code samples to help in the development of your C++ connector:

`DevelopmentKits\cdk\samples`

The CDK is supported only on Windows systems. To compile a C++ connector, use the compiler in the Microsoft Visual C++ 6.0 development environment. For more information, see "Compiling the connector" on page 200.

Note: The WebSphere Business Integration Adapters product also provides a Java version of the Connector Development Kit for use in development connectors in the Java programming language. For more information, see the *Connector Development Guide for Java*.

ODA samples: The Adapter Development Kit includes samples for an Object Discovery Agent (ODA). These samples reside in the following directory:

`DevelopmentKits\Odk`

For more information, see "Development support for ODAs" on page 28.

Overview of the connector development process

This section provides an overview of the connector development process, which includes the following high-level steps:

1. Install and set up the IBM WebSphere business integration system software.
2. Design and implement the connector.

Setting up the development environment

Before you start the development process, the following must be true:

- The IBM WebSphere business integration system software is installed on a machine that you can access.

WebSphere InterChange Server

If your business integration system uses InterChange Server, refer to the *System Installation Guide for UNIX* or *for Windows* (in the WebSphere InterChange Server documentation set) for information on how to install and start up the InterChange Server system.

Other integration brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere Integrator Broker, WebSphere Business Integration Message Broker), refer to the installation chapter of the *Implementing Adapters for WebSphere Message Brokers* for information on how to install and start up the IBM WebSphere business integration system. If your business integration system uses WebSphere Application Server, refer to the installation chapter of the *Implementing Adapters for WebSphere Application Server* for information on how to install and start up the IBM WebSphere business integration system.

- Ensure that the development environment can access the directories that contain the connector library files. To compile the connector, the compiler *must* be able to access the connector library.

  For information on compiling a connector, see "Compiling the connector" on page 200.

InterChange Server

- If your business integration system uses InterChange Server, the InterChange Server repository's database server and ICS are running.

  Note: This step is required only when you are ready to configure the connector with Connector Configurator. For development only, you can create the connector class, without connecting to ICS.

  For an overview of how to configure a connector, see Chapter 8, "Adding a connector to the business integration system," on page 199. For information on starting up the IBM WebSphere business integration system, see your system installation guide.

End of InterChange Server

Note: To create a connector, you do not need to run the messaging software. However, the messaging software must be running before you can execute and test the connector.

Stages of connector development

As part of the connector development process, you code the application-specific component of the connector and then compile and link the connector source files. In addition, the overall process of developing a connector includes other tasks, such as developing application-specific business objects. Here is an overview of the tasks in the connector development process:

1. Identify the application entities that the connector will make available to other applications, and investigate the integration features provided by the application.

InterChange Server

2. If your business integration system uses InterChange Server, identify generic business objects that the connector will support, and define application-specific business objects that correspond to the generic objects.

3. If your business integration system uses InterChange Server, analyze the relationship between the generic business objects and the application-specific business objects, and implement the mapping between them.

End of InterChange Server

4. Define a connector base class for the application-specific component, and implement functions to initialize and terminate the connector.
5. Define a business object handler class and code one or more business object handlers to handle requests.
6. Define a mechanism to detect events in the application, and implement the mechanism to support event subscriptions.
7. Implement error and message handling for all connector methods.
8. Build the connector.
9. Configure the connector.

WebSphere InterChange Server

If your business integration system uses InterChange Server, use Connector Configurator to create the connector definition and save it in the InterChange Server repository. You can call Connector Configurator from System Manager.

Other integration brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, use Connector Configurator to define and create the connector configuration file.

10. If WebSphere MQ will be used for messaging between connector components, add message queues for the connector.
11. Create a startup script for the new connector.
12. Test and debug the connector, recoding as necessary.

Figure 12 provides a visual overview of the connector development process and provides a quick reference to chapters where you can find information on specific topics. Note that if a team of people is available for connector development, the major tasks of developing a connector can be done in parallel by different members of the connector development team.

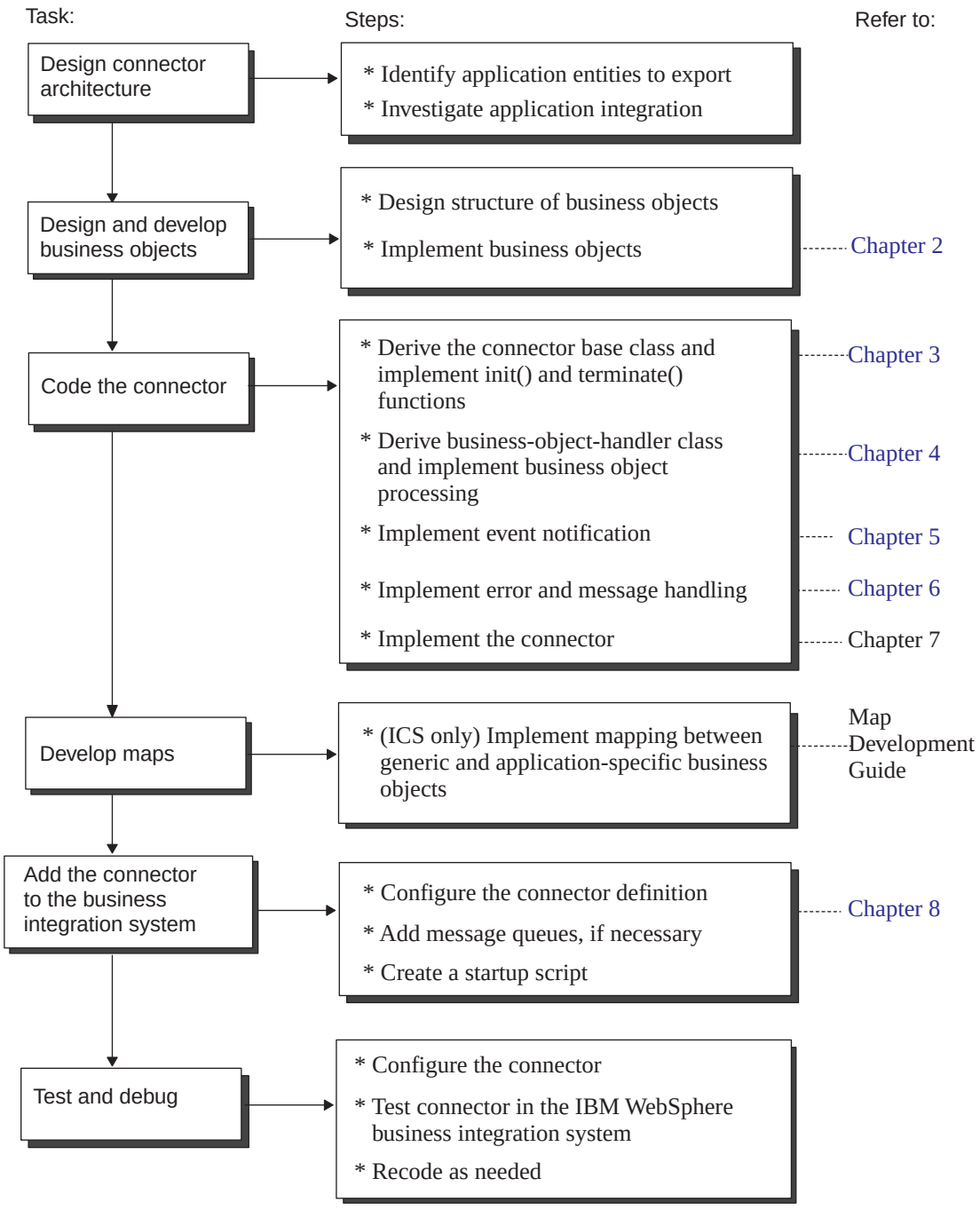

Figure 12. Overview of the C++ connector development process

Part 2. Building a connector

Chapter 2. Designing a connector

This chapter provides an overview of analysis and design issues to consider when planning a connector development project. The chapter presents topics that can help you judge the complexity of building a connector for your application or technology.

As with most software development projects, careful planning early in the connector development cycle helps prevent problems during later implementation phases. This chapter contains the following sections:

Scope of a connector development project

IBM provides a *connector framework* as part of the C++ Connector Development Kit. The connector framework contains all the code necessary for the connector to interact with an integration broker and provides a basic infrastructure for interaction with the application.

Your task as a connector developer is to code the application-specific component of a connector, and if necessary, develop the event notification mechanism. The complexity of the design for your connector and the time required for the connector's implementation will vary based on the application.

To understand the scope and complexity of a connector development project, you may want to develop a project plan before beginning a new connector. As you develop the project plan, you need to identify the business requirements for the connector, define the application data that the connector will handle, and determine what application business processes the connector and business objects will work with. Developing a project plan can help you understand application functionality in the areas of business objects, business object processing, and event management.

Working through the topics in this chapter can help you estimate the time and effort needed to complete the connector development task. Each topic provides a set of questions that are intended to develop understanding of specific aspects of an application that might increase or decrease the complexity of the connector development task. A complete set of answers to the questions for each topic provides a high-level architecture for your connector.

Step in connector design	For more information
Obtain information about the application that is relevant to the design of the connector architecture.	"Designing the connector architecture" on page 38

Step in connector design	For more information
Ensure that application-specific business objects adequately represent the application entities that the connector needs to export.	"Designing application-specific business objects" on page 43
Design the event notification mechanism so that the application can notify the connector of relevant events.	"Event notification" on page 51

Designing the connector architecture

To design the connector architecture, consider evaluating the following areas of the application that the connector is to support:

- "Understanding the application environment" on page 39
- "Determining connector directionality" on page 40
- "Getting data in and out of the application" on page 41

The specific areas within an application that affect connector design are illustrated in Figure 13.. In this figure, the clouds show the high-level tasks required for connector development.

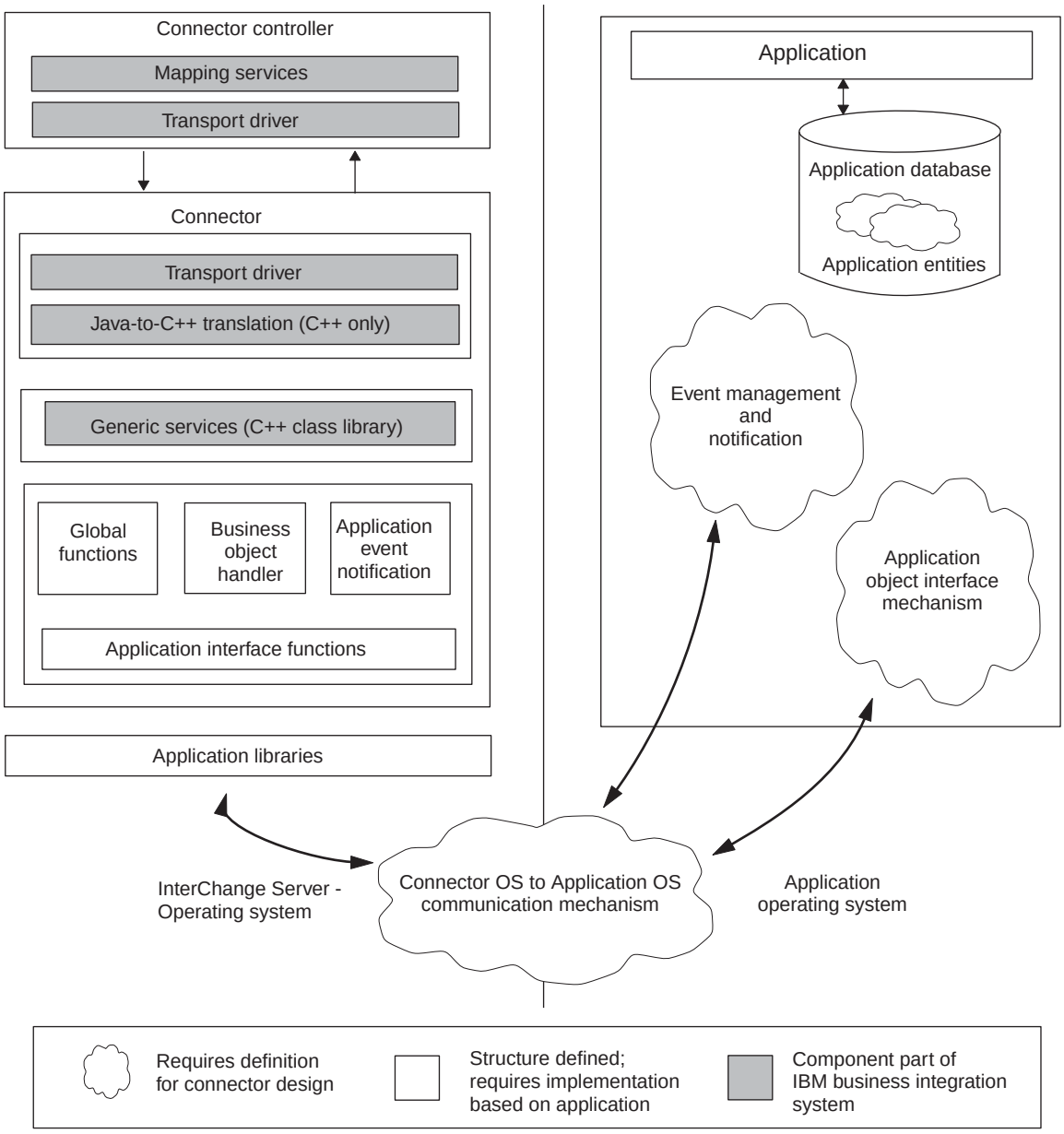

Figure 13. Areas of an application that affect connector design

Understanding the application environment

Understanding the application environment is the first step in assessing the feasibility of a connector development project. To obtain an understanding of the aspects of an application that affect connector development, consider these topics and questions:

Operating system

- What operating system does the application run on?

Programming languages

- What programming languages were used to create the application?

Application execution architecture

- What is the execution architecture of the application? For example, in a centralized architecture, the application and its database might both reside on a mainframe system. In this case, both application processing and database processing occur on this central system.

  Alternatively, in a client-server architecture, the database might reside on a server, and the application front-end program might be a client running on another machine, such as a personal computer. Other types of application execution architecture are online transaction processing and file server architecture.

Database type

- Is there a central database for application data? If application data is stored in a central database, what type of database is it? Example database types are RDMS and flat file.

Distributed application

- Is the application distributed across multiple servers?
- Is the application database distributed across multiple servers?

During project assessment, you may want to identify and work with an application expert. This person can also provide assistance during business object development and connector development.

Determining connector directionality

Early on in the project planning phase, you need to determine what roles the connector will perform for the application:

- Request processing—Update application data at the request of an integration broker. For more information, see "Request processing" on page 25..
- Event notification—Detect application events and send notification of events to the integration broker. For more information, see "Event notification" on page 22..

These roles determine the *directionality* that the connector supports:

- Unidirectional— some connectors might need to operate in only one direction, passing data from the application to the integration broker, or from the integration broker to the application.
  - To inform an integration broker that changes have occurred in the application, a connector must support event notification.
  - To receive data from an integration broker, a connector must support request processing, in which it interacts with the application to support Create, Retrieve, Update, or Delete operations as requested by the integration broker.

  For example, a connector might simply need to receive request business objects from an integration broker and pass them to an application. The connector for an application that serves only as the destination is a unidirectional connector – it implements request handling to pass data to the application, but it does not implement event notification. Knowing early in the development cycle that your connector will operate unidirectionally can save a significant amount of development time.
- Bidirectional—most connectors need to operate in *both* directions, passing data from the application to an integration broker *and* receiving data back from the integration broker.

To be bidirectional, your connector needs to support *both* event notification and request processing.

For information on how to provide event notification support in your connector, see Chapter 5, "Event notification," on page 109.

Getting data in and out of the application

An important aspect of the connector development project plan is to determine how the connector will get data into and out of the application. Ideally, an application provides an application programming interface (API) that includes all of the following features:

- Support for Create, Retrieve, Update, and Delete (CRUD) operations at the object level
- Encapsulation of all of the application business logic
- Support for delta and after-image operations
- An event-management strategy that allows external notification at the subobject level.

Typically, however, an application interface falls short of this ideal.

In your project plan, you need to establish whether a formal application API exists and evaluate its robustness, or, if an API does *not* exist, determine whether there is a suitable workaround. Keep in mind that an application CRUD interface can be anything from batch file imports and extracts to a COM/DCOM server, so be sure to explore all possible avenues. Refer to the application business object scope specified in Table 13 when exploring the application object CRUD interface.

Consider the following tasks:

- "Examining previous integration efforts"—Have there been any other efforts to integrate with this application?
- "Determining whether application data is shared with other applications" on page 42—Is the application data shared by other applications?
- "Examining an application API" on page 42—Is there an existing mechanism that the connector can use to communicate with the application?
- "Application use of batch clean-up or merge programs" on page 43—Does the application use batch clean-up or merge programs?

These questions are discussed in more detail in the following sections.

Examining previous integration efforts

If you have access to previous efforts to integrate other applications with your application, you might be able to find ways of getting data into and out of the application. Even if you decide to implement another approach to application integration, the previous integration effort may provide useful design information.

When examining previous integration efforts, consider these questions:

- What was the purpose of the integration?
- Does the integration use interfaces that modify or retrieve information from the application? If so, describe the mechanism used to modify or retrieve information.
- If the integration can process an event generated in the application, what is the mechanism used to trigger event processing?
- What is the mode of the existing integration? (batch, asynchronous, and so on)

- Will your connector replace the pre-existing integration? If not, will previous integrations work with the data entities that your connector will be working with?

In your answers, include information on all previous integration efforts that interact with the application in different ways.

Determining whether application data is shared with other applications

Your application might be one of several applications creating or updating data in a single database. In this case, your connector might have to consider an application data entity based on work that other applications are also doing. If you determine that your connector will be sharing application data with other applications, consider these questions:

- What is the mechanism used by the other applications to gain access to the application data?
- Do other applications create, retrieve, update, or delete application data? If so, what mechanism do other applications use for each verb?
- Is there object-specific business logic used by other applications? Is the logic consistent throughout all of the applications?

Provide answers to these questions for all applications that share the application data.

Examining an application API

If the application provides an API or other mechanism that the connector can use to communicate with the application, examine the API and review any available documentation. Keep in mind the following questions about the API:

- Does the API allow access for Create, Retrieve, Update, and Delete operations?
- Does the API provide access to all attributes of a data entity?
- Are there inconsistencies in the API implementation? Is the navigation to Create/Retrieve/Update/Delete the same regardless of the entity?
- Describe the transaction behavior of the API. For example, an API might simply enable the connector to run a report, which the connector can then read and use for processing. Or the API might be more robust, providing ways of performing asynchronous or synchronous Create and Update operations.
- Does the API allow access to the application for event detection? For example, if an application event-notification mechanism uses a database table as an event store, does the API allow access to this table?
- Is the API suited for metadata design? APIs that are forms-based, table-based, or object-based are good candidates. For information on metadata design, see "Assessing support for metadata-driven design" on page 47.
- Does the API enforce application business rules? In other words, is it an API that interacts at the table level, form level, or object level?

The recommended approach to connector development is to use whatever API the application provides. The use of an API helps ensure that connector interactions with the application abide by application business logic. In particular, a high-level API is usually designed to include support for the business logic in the application, whereas a low-level API might bypass application business logic.

As an example, a high-level API call to create a new record in a database table might evaluate the input data against a range of values, or it might update several

associated tables as well as the specified table. Using SQL statements to write directly to the database may bypass the data evaluation and related table updates performed by an API.

If no API is provided, the application might allow its clients to access its database directly using SQL statements. If you use SQL statements to update application data, work closely with someone who knows the application well so that you can be sure that your connector will not bypass application business logic.

This aspect of the application has a major impact on connector design because it affects the amount of coding that the connector requires. The easiest application for connector development is one that interacts with its database through a high-level API. If the application provides a low-level API or has no API, the connector will probably require more coding.

Application use of batch clean-up or merge programs

A final aspect of the application business object interface that you need to investigate is whether the application uses any batch clean-up or merge programs to purge redundant or invalid data. For example, an application may run a batch program once a day to standardize site names that operators may have typed in incorrectly or incompletely. This program might, for example, change all sites named IBM WebSphere to IBM WebSphere Software.

When this type of batch program runs, all changes to the database may also need to flow through an InterChange Server customer synchronization system. A program like this may result in hidden requirements for your connector. For example, even if it appears initially that your connector does not need to provide Delete functionality, you may need to provide Delete functionality to support a batch clean-up program that deletes all sites named IBM WebSphere.

You may decide that you want to handle batch clean-up tasks periodically, such as once a month, rather than synchronously. In any case, an important planning task is to gather information about any programs that result in unanticipated requirements for your connector.

Designing application-specific business objects

Application-specific business objects are the units of work that are triggered within the application, created and processed by the connector, and sent to the integration broker. A connector uses these business objects to export data from its application to other applications and to import data from other applications.

The connector exposes all the information about an application entity that is necessary to allow other applications to share the data. Once the connector makes the entity available to other applications, the integration broker can route the data to any number of other applications through their connectors.

Designing the relationship between the connector and its supported application-specific business objects is one of the tasks in connector development. Application-specific business object design can generate requirements for connector programming logic that must be integrated into the connector development process. Therefore, business object and connector developers must work together to develop specifications for the connector and its business objects.

Consider the following design guidelines when you design your application-specific business objects:

1. Determine what application entities the connector will work with.
2. Determine the scope of business object development.
3. Determine support for a metadata-driven design.

Note: For more information about the design of application-specific business objects, see the *Business Object Development Guide*.

Determining the application entities

The complexity of business objects can have a significant impact on the amount of work that is necessary to build a connector. A first step in identifying application-specific business objects is to determine what application entities the connector will work with.

You can identify application entities that the connector will work with in two ways:

- Focus on existing InterChange Server collaborations whose business processes correspond to those of your application.
- Focus on other applications that you want to integrate with your application.

Design focus on InterChange Server collaborations

If you are using InterChange Server as your integration broker, one way to begin identifying application-specific business objects is to list the InterChange Server collaborations that you want the application to work with. Consider the features of each collaboration, and note which generic business objects each collaboration references. Using this list, you can decide what kinds of business objects allow your application to work with the collaboration.

For example, you may decide that you want to use your application with the Customer Manager collaboration. In this case, the connector must handle customer entities. The connector might extract customer data from the application to forward to the collaboration or receive customer data from the collaboration to pass back to the application.

Design focus on other applications

Alternatively, you might start the connector development task by looking at other applications with which you want to integrate. As you examine your application and other applications, you can determine what business processes you want to share across applications and identify what data you want to exchange. The goal is to determine what entities in your application make sense to implement as business objects to enable integration with other applications.

For example, if your application stores customer data, you may want to keep the customer database consistent with the customer database in another application. To synchronize customer data, you need to know about the customer entity that each application publishes. Figure 14 illustrates a design approach that focuses on integrating with other applications.

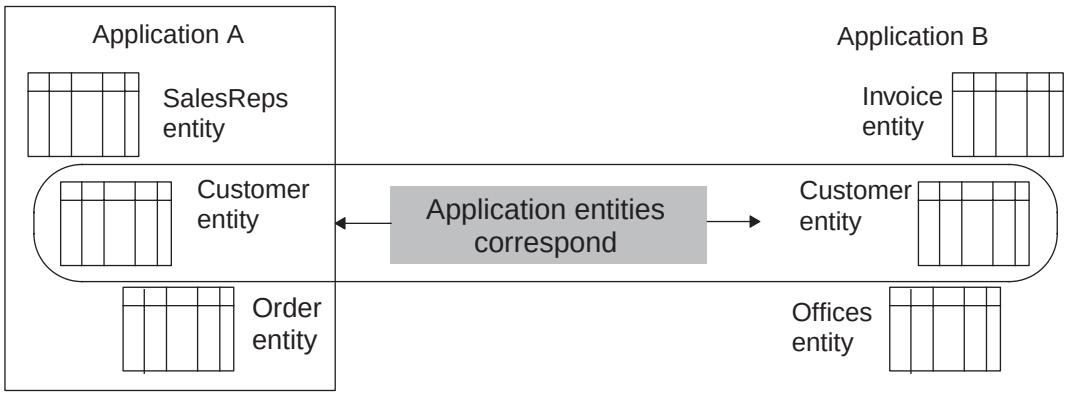

Figure 14. Design focus: identify applications with which to integrate

Design focus on the application

Use the following topics and questions to gather more information about application entities and business objects:

- "Contained entities"
- "Database representation of entities"
- "Denormalization of application entities" on page 46
- "Batch processing of application entities" on page 46

Contained entities:

- Do the application entities have contained entities?

  For example, in many applications a contract entity has one to many line items. The IBM WebSphere Business Integration Contract business object contains child line items as business objects. Determine whether the entities your connector will work with have related entities that will be defined as child business objects.

Database representation of entities:

- Are there application business entities that are the same type but that have different physical representations in the application?

  For example, an application may have two types of contracts: hardware contracts and software contracts. Both are of type Contract, but they are stored in different tables in the application database. In addition, the attributes for each Contract type differ.

  Because a single set of maps can convert between only one generic business object and one application-specific business object, developers for this application must design business objects to account for the different entities in the application. For example, they may need to redesign the IBM WebSphere Business Integration generic business object, create new generic child business objects, and create new maps.

  Figure 15 shows the business objects that may result from multiple application entities of the same type. It illustrates the creation of two generic child business objects, one that contains data specific to hardware contracts and one that contains data specific to software contracts.

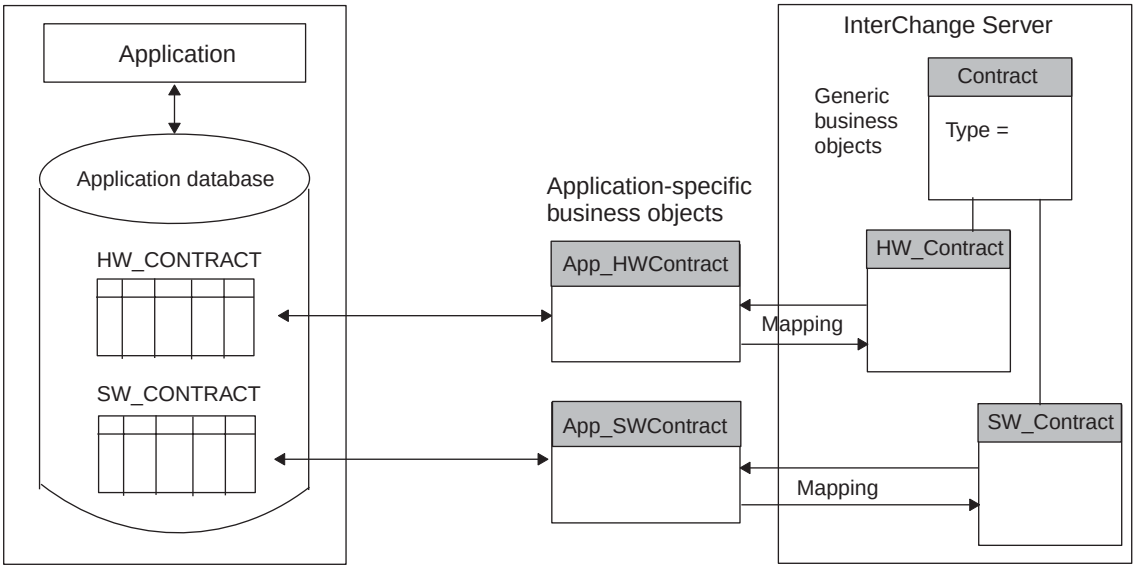

Figure 15. Database representation of application entities

Denormalization of application entities: Are there application entities that reside in more than one location in the database but that correspond to the same logical entity?

For example, Contract, Customer, and Contact entities might each have Customer address fields as part of the physical table definition for each entity. If the Customer address field changes in one entity, it must be updated in all entities.

However, the address fields might be consolidated into an Address business object that needs to be updated for the Contact, Customer, and Contract business objects if the address changes for any of the entities. In this case, the Address business object would be referenced rather than contained by the top-level business objects that use the data.

Batch processing of application entities: Are there batch processes associated with the creation of application entities?

In some applications, batch processing may add data to entities. As an example, a data entry operator may enter a new customer into the application database at 11:00 AM, but the customer record will not be complete until a 7:00 PM batch job runs to fill in some remaining values.

If a batch process is associated with application entities and the process adds important or required data, you need to determine when the business object is generated. For example:

- If the batch process generates the event notification, the event will trigger the connector to send a complete business object into the IBM WebSphere business integration system.
- If the operator's Save operation generates the event notification, the event may trigger the connector to send an incomplete business object.

If there is a need for real-time data synchronization, but there are batch processes running in the background, your connector development plans must account for this.

Determining the scope of business object development

When you have determined at a high level what business objects you need to define, you then need to determine the verb support for the business object development, as follows:

1. Use Table 13 to create a verb-scope summary for each business object and verb combination that your connector will support.
2. Use the completed scope summary to assemble information about each business object.

Table 13. Business Object Verb-Scoping Summary

Business object name	Required request Verbs (request processing)	Required delivery verbs (application event notification)
Object 1	O Create O Update O Delete	O Create O Update O Delete
Object 2	O Create O Update O Delete	O Create O Update O Delete
Object *n*	O Create O Update O Delete	O Create O Update O Delete

Important: Most connectors *must* support the Retrieve verb for each business object; therefore, it is not included in Table 13..

Assessing support for metadata-driven design

In addition to its structure and attributes, a business object definition can contain application-specific information, which can provide processing instructions or information on how the business object is represented in the application. Such information is called *metadata*.

Metadata can include any information that the connector needs in its interactions with the application. For example, if a business object definition for a table-based application includes metadata that provides the application table and column names, the connector can locate requested data using this information, and the application column names do not need to be encoded in the connector. Because the connector has access to its supported business object definitions at runtime, it can use the metadata in the business object definition to dynamically determine how to process a particular business object.

Depending on the application and its programming interface (API), a connector and its business objects might be designed based on the ability to support the use of metadata, as Table 14 shows.

Table 14. Connector support for metadata

Connector's use of metadata	Business object handlers required	For more information
Entirely driven by the processing instructions in the metadata of its business object definitions	One generic metadata-drive business object handler	"Metadata-driven connectors" on page 48
Partially driven by the metadata in its business object definitions	One partially metadata-driven business object handler	"Partially metadata-driven connectors" on page 49
Cannot use metadata	Separate business object handler for each business object that does not use metadata	"Connectors that do not use metadata" on page 50

While some application interfaces have constraints that restrict the use of metadata in connector and business object design, a worthwhile goal for connector

development is to make the connector as metadata driven as possible. Advantages and disadvantages of the approaches in Table 14 are discussed below.

Metadata-driven connectors

To be able to support metadata-driven design, the application API must be able to specify what objects in the application are to be acted upon. In general, this means that you can use the business object metadata to provide information about the application entity to be acted upon and the attribute data as the values for that object. A *metadata-driven connector* can then use the business object values and the metadata (the application-specific information that the business object definition contains) to build the appropriate application function calls or SQL statements to access the entity. The function calls perform the required changes in the application for the business object and verb the connector is processing.

Applications based on forms, tables, or objects are well suited for metadata-driven connectors. For example, applications that are forms-based consist of named forms. Programmatic interaction with a forms-based application consists of opening a form, reading or writing fields on the form, and then saving or dismissing the form. The connector for such an application can be driven directly by the business object definitions that the connector supports.

The main benefit to a metadata-driven connector is that the connector can use one generic business object handler for *all* business objects. In this approach, the business object definition contains *all* the information that the connector needs to process the business object. Because the business object itself contains the application-specific information, the connector can handle new or modified business objects without requiring modifications to the connector source code. The connector can be written in a generic manner, with a single *metadata-driven business object handler*, which does *not* contain hard-coded logic for processing specific business objects.

Note: Business object names should not have semantic value to the connector. The connector should process identically two business objects with the same structure, data, and application-specific information with different names.

WebSphere InterChange Server

Figure 16 shows an application-specific business object and a connector with a meta-data-driven business object handler. The processing instructions in the application-specific information of the `App_Order` business object tell the connector how to process the business object.

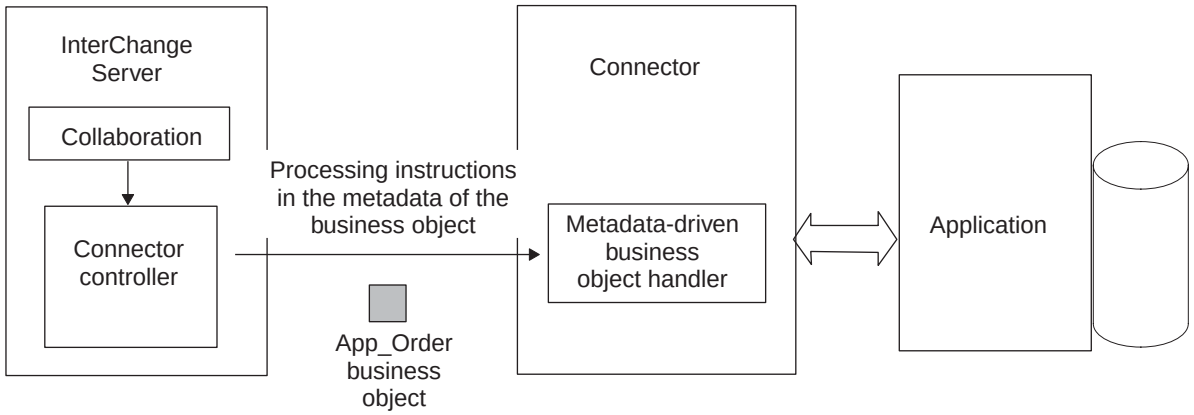

Figure 16. Using metadata in the business object for processing instructions

Because a metadata-driven connector derives its processing instructions from its application-specific business objects, the business objects must be designed with this type of processing in mind. This approach to connector and business object design provides flexibility and easy extensibility, but it requires more planning in the design phase. When connectors are designed to work with business object metadata, the business object itself can be changed without requiring corresponding changes in the connector.

For more information on designing a metadata-driven business object handler, see "Implementing metadata-driven business object handlers" on page 74.

Partially metadata-driven connectors

IBM encourages the metadata approach for designing connectors and application-specific business object definitions. However, some applications might not be suited for this approach. Application APIs that are specific for each entity in an application make it more difficult to write a metadata-driven connector. Often the issue is that the call itself differs between objects in some structural way, rather than just in the name of the method or the data that is passed.

Sometimes you can still drive a connector with metadata, though this metadata does not contain the actual processing instructions. This *partially metadata-driven connector* can use the metadata in the business object definition or attributes to help determine what processing to perform. For example, an application that has a large amount of business logic embedded in its user interface might have restrictions on how an external program, such as a connector, can get information into and out of its database. In some cases, it may be necessary to provide an extension to the application using the application environment and application programming interface. You may need to add object-specific modules to the application to handle the processing for each business object. The application may require the use of its application environment and interface to ensure that application business logic is enforced and not bypassed.

In this case, the business object and attribute application-specific information can still contain metadata for the connector. This metadata specifies the name of the module or API call needed to perform operations for the business object in the application. The connector can still be implemented with a single business object handler, but it is a *partially metadata-driven business object handler* because this metadata does not contain the processing instructions.

Figure 17 illustrates an application extension that is responsible for handling requests from the connector. The extension contains separate modules for each business object supported by the connector.

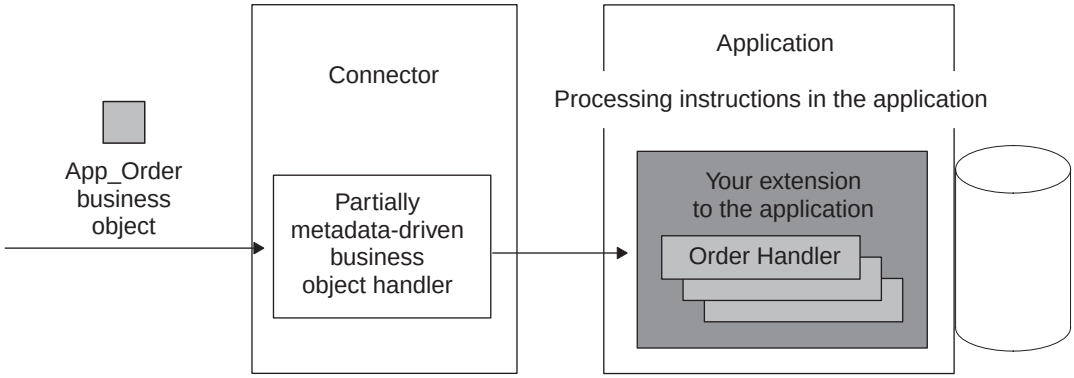

Figure 17. Application-specific processing in the application

The benefit to the partially metadata-driven connector is that it still uses just one business object handler. However, unlike with a metadata-driven connector, there is coding to do when new business objects are created for the connector. In this case, new object functions must be written and added to the application, but the connector does not need to be recoded or recompiled.

Connectors that do not use metadata

If the application API does *not* provide the ability to specify what entities in the application are to be acted upon, the connector cannot use metadata to support a single business object handler. Instead, it must provide *multiple business object handlers*, one for each business object the connector supports. In this approach, each business object handler contains specific logic and code to process a particular business object.

In Figure 18,, the connector has multiple, object-specific business object handlers. When the connector receives a business object, it calls the appropriate business object handler for that business object.

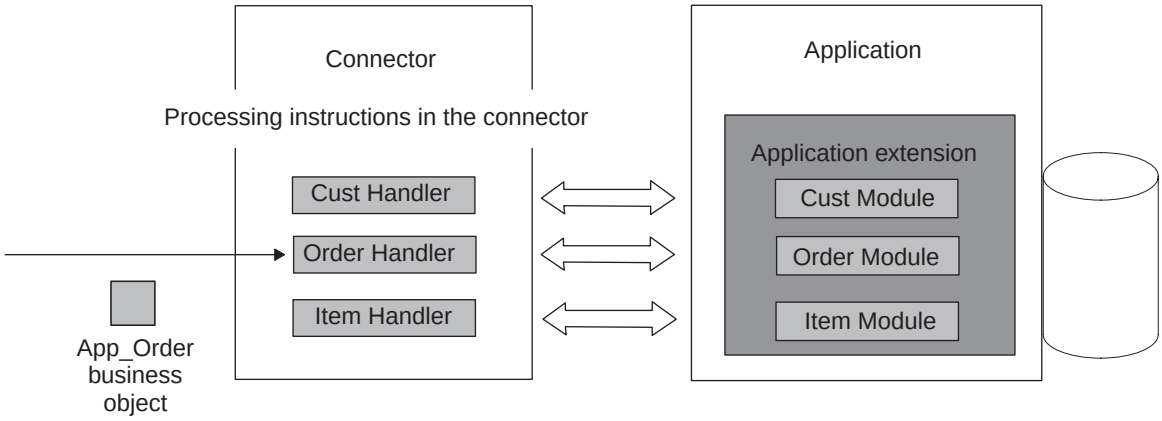

Figure 18. Application-specific processing in the connector

The drawback of this non-metadata approach is that when a business object is changed or a new business object is added, this type of connector must be recoded to handle the new or changed business object.

Event notification

The IBM WebSphere business integration system is an event-driven system, and connectors need some way to detect and record events that occur in the application. When you examine the application, determine whether it provides an event-notification mechanism that can notify the connector of changes to application data.

Event notification typically consists of a collection of processes that allows a connector to be notified of internal application events. The event record should include the type of the event, the business object name and verb, such as Customer and Create, and the data key required for the connector to retrieve associated data.

In addition, an event-notification strategy must incorporate the necessary mechanisms to ensure the data integrity between event records and the corresponding event data. In other words, an event notification should not occur until *all* the required data transactions for the event have completed successfully.

The design of an event notification mechanism varies depending on the extent to which the application reports application events and enables clients to retrieve event data. If the application provides an event notification interface such as an API, IBM recommends that you use this to implement the event-notification mechanism. The use of an API helps ensure that connector interactions with the application abide by application business logic. If the application provides an event-notification mechanism, use the following topics and questions to gather more information.

Event notification level of detail

- Does the application's event-notification mechanism provide enough detail about the event to establish the discrete business object and verb? If not, can the event notification component be configured to provide this level of detail?

  For example, if a new record is added or an existing customer is updated, determine whether the event-notification mechanism can provide information on the type of operation, such as Create or Update operations. If the connector supports delta operations, determine whether the event mechanism can provide information on exactly which subobjects or attributes changed.

Event notification support for business logic

- Does event notification occur at a level that adequately supports business requirements? In other words, an event-notification mechanism would ideally include support for application business logic.

In your project plan, describe the event-notification mechanism. If there is no existing event mechanism, determine what alternatives are available to detect changes to application data. For example, you might be able to provide event notification by setting up database triggers on tables in a relational database. Or the application might provide a batch-export capability that exports all database modifications to a file from which the connector can extract information about application events.

Note: For more information on the stages of implementing an event-notification mechanism, see "Overview of an event-notification mechanism" on page 109.

Communication across operating systems

Communication between the application and the connector is a major component in the overall connector design. If the application runs on a different operating system from InterChange Server and the connector, you must ensure that a mechanism is in place to allow the connector access to the application.

If the application provides an API, determine whether the API handles the communication between the operating system of application and that of the connector. For example, if the application runs on UNIX and the connector and InterChange Server run on Windows 2000, the application API might enable the connector and application to communicate across operating systems.

Figure 19 shows an example communication mechanism between an ODBC connector running on Windows 2000 and an ODBC-based application running on UNIX. The connector builds dynamic SQL statements and executes them using the ODBC API. The ODBC driver enables the connector to establish a connection with the application database and to access the database using ODBC SQL statements.

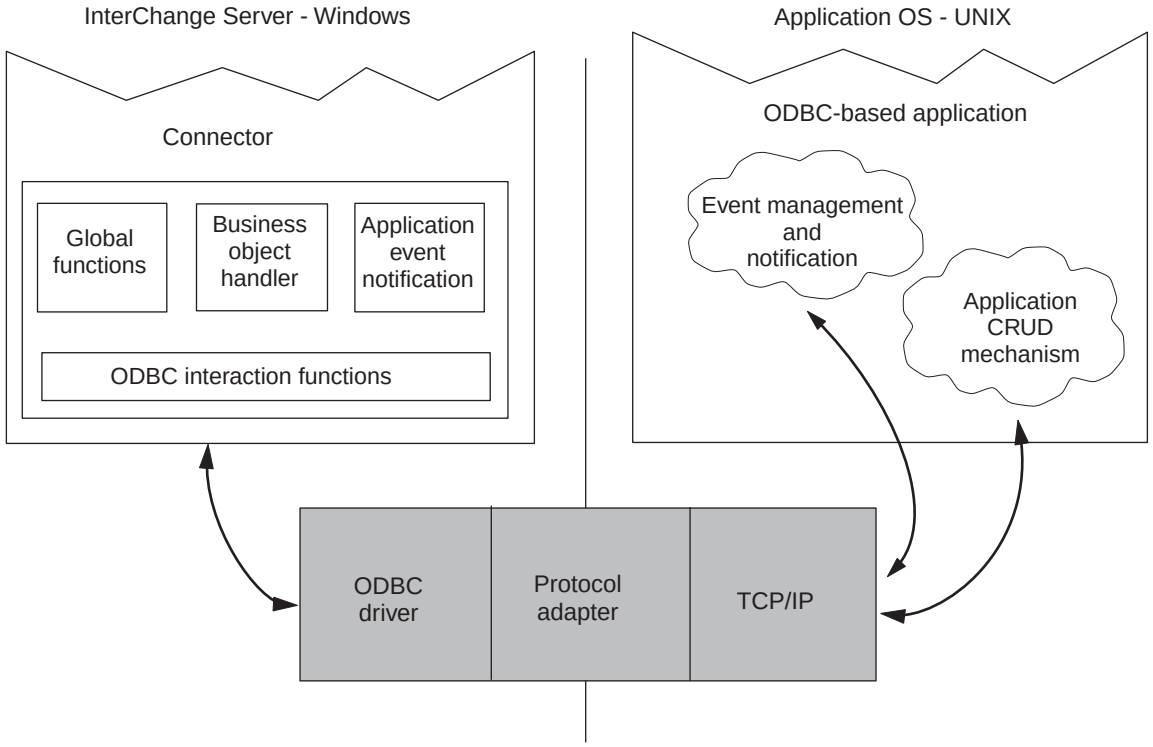

Figure 19. Sample Windows-to-UNIX communication

Summary set of planning questions

The following table lists the set of planning questions provided in this chapter. You can use this table as a worksheet for gathering information about your application. As you gather information, get copies of any documentation that can help in the planning, design, or development phases of the project.

1. Understanding the Application
   - What is the application operating system?
   - What programming languages were used to create the application?
   - What is the execution architecture of the application?
   - Is there a central database for application data? What type of database is it?
   - Is the application or its database distributed across multiple servers?
2. Identifying the Directionality of the Connector
   - Does the connector need to send data, receive data, or both?
3. Identifying the Application-Specific Business Objects
   - Do application entities have contained entities?
   - Are there application business entities that are the same type but have different physical representations in the application?
   - Are there application entities that reside in more than one location in the database but correspond to the same logical entity?
   - Are there batch processes associated with the creation of application entities?
4. Investigating the Application Data Interaction Interface
   - Have there been any other efforts to integrate with this application?
     - What was the purpose of the integration?
     - Does the integration use interfaces that modify or retrieve information?
     - If the integration is able to process an event generated in the application, what is the mechanism used to trigger event processing?
     - Will your connector replace the pre-existing integration?
   - Is application data shared by other applications?
     - Do other applications create, retrieve, update, or delete this application's data?
     - What is the mechanism used by other applications to gain access to the data?
     - Is there object-specific business logic used by other applications?
   - Is there a mechanism that the connector can use to communicate with the application?
     - Does the API allow access for create, retrieve, update, and delete operations?
     - Does the API provide access to all data entity attributes?
     - Does the API allow access to the application for event detection?
     - Are there inconsistencies in the API implementation?
     - Describe the transaction behavior of the API.
     - Is the API suited for meta-data design?
     - Does the API enforce application business rules?
   - Are there batch clean-up or merge programs used to purge redundant or invalid data?
5. Investigating the Event Management and Notification Mechanism
   - Describe the event management mechanism.
   - Does it provide the necessary granularity to establish the distinct object and verb?
   - Does event notification occur at a level that can support application business logic?
6. Investigating Communication Across Operating Systems
   - Does the API handle the communication mechanism between the application operating system and the connector operating system?
   - If not, is there a mechanism available to handle communication across operating systems?

Figure 20. Summary set of planning question

Evaluating the findings

As you assemble the answers to the questions presented in this chapter, you acquire essential information about application data entities, business object processing, and event management. These findings become the basis for a high-level architecture for the connector.

When you have determined what entities your connector will support and have examined the application functionality for database interaction and event notification, you should have a clear understanding of the scope of the connector development project. At this point, you can continue with the next phases of connector development—defining application-specific business objects and coding the connector.

Figure 21 shows a partial write-up of information about a sample connector. Figure 22 illustrates a high-level architecture diagram for an ODBC-based connector.

1. Understanding the Application
   - Application is running on UNIX.
   - Programming language used is Visual C++ with the Microsoft MFC libraries.
   - Application is client-server.
   - Application has a central database. Type is RDMS.
   - Application is not distributed.
2. Identifying the Directionality of the Connector
   - Connector will be bidirectional.
3. Identifying the Application-Specific Business Objects
   - Business objects have contained objects. Contained business objects are:
     - Customer "Address "Site Use and Site Profile
     - Item "Status
     - Contact "*n* Phones and *n* Roles
   - Application business entities do not have different physical representations in the application.
   - Application entities do not reside in more than one location in the database.
   - No batch processes are associated with the creation of these objects.
4. Examining the Application Data Interaction Interface
   - No previous efforts to integrate with this application.
   - Application data is not shared by other applications.
   - The application provides the OpenProduct API.
     - OpenProduct allows for Creates and Updates but not Retrieves and Deletes.
     - The API provide access to all data entity attributes.
     - The API allows access to the application for event detection. We can create an event table and poll for events at a specified interval.
     - There are no inconsistencies in the API.
     - The API has a batch interface.
     - The application is table-based, and the API is suited for meta-data design.
     - ...

Figure 21. Sample results write-up

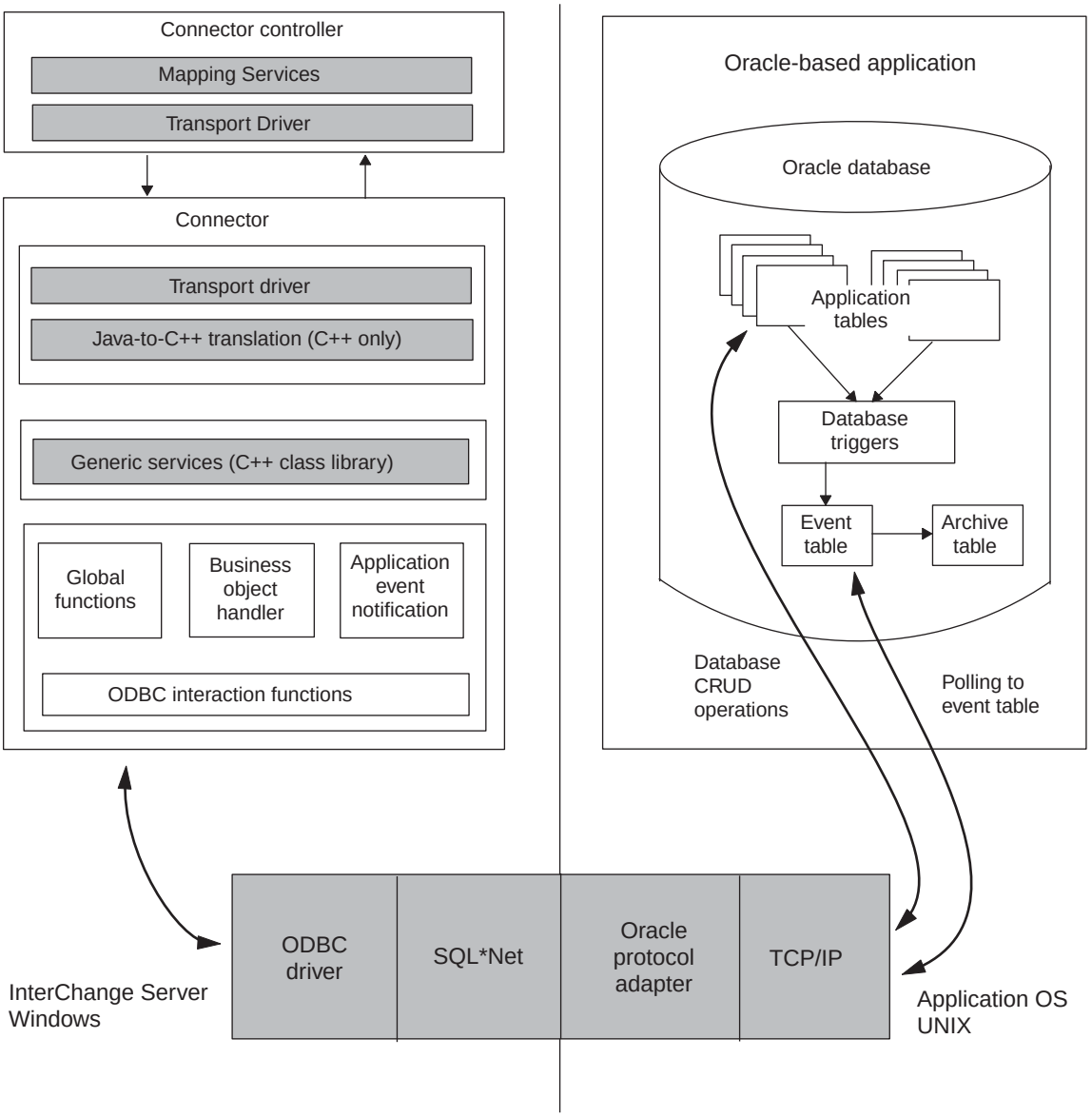

Figure 22. Sample ODBC-based connector architecture

An internationalized connector

An *internationalized connector* is a connector that has been written so that it can be customized for a particular locale. A *locale* is the part of a user's environment that brings together information about how to handle data that is specific to the end user's particular country, language, or territory. The locale is typically installed as part of the operating system. Creating a connector that handles locale-sensitive data is called the *internationalization* (I18N) of the connector. Preparing an internationalized connector for a particular locale is called the *localization* (L10N) of the connector.

This section provides the following information on an internationalized connector:

- "What is a locale?" on page 56
- "Design considerations for an internationalized connector" on page 56

What is a locale?

A *locale* provides the following information for the user environment:

- Cultural conventions according to the language and country (or territory):
  - Data formats:
    - Dates: define full and abbreviated names for weekdays and months, as well as the structure of the date (including date separator).
    - Numbers: define symbols for the thousands separator and decimal point, as well as where these symbols are placed within the number.
    - Times: define indicators for 12-hour time (such AM and PM indicators) as well as the structure of the time.
    - Monetary values: define numeric and currency symbols, as well as where these symbols are placed within the monetary value.
  - Collation order: how to sort data for the particular character code set and language.
  - String handling includes tasks such as letter "case" (upper case and lower case) comparison, substrings, and concatenation.
- A *character encoding* — the mapping from a character (a letter of the alphabet) to a numeric value in a character code set. For example, the ASCII character code set encodes the letter "A" as 65, while the EBCIDIC character set encodes this letter as 43. The *character code set* contains encodings for all characters in one or more language alphabets.

A locale name has the following format:

ll_TT.codeset

where *ll* is a two-character language code (usually in lower case), *TT* is a two-letter country and territory code (usually in upper case), and *codeset* is the name of the associated character code set. The *codeset* portion of the name is often optional. The locale is typically installed as part of the installation of the operating system.

Design considerations for an internationalized connector

This section provides the following categories of design considerations for internationalizing a connector:

- "Locale-sensitive design principles"
- "Character-encoding design principles" on page 60

Locale-sensitive design principles

To be internationalized, a connector must be coded to be locale-sensitive; that is, its behavior must take the locale setting into consideration and perform the task appropriate to that locale. For example, for locales that use English, the connector should obtain its error messages from an English-language message file. The WebSphere Business Integration Adapters product provides you with an internationalized version of the connector framework. To complete the internationalization (I18N) of a connector you develop, you must ensure that your application-specific component is internationalized.

Table 15 lists the locale-sensitive design principles that an internationalized application-specific component must follow.

Table 15. Locale-sensitive design principles for application-specific components

Design principle	For more information
The text of all error, status, and trace messages should be isolated from the application-specific component in a message file and translated into the language of the locale.	"Text strings"
The locale of a business object must be preserved during execution of the connector.	"Business object locales" on page 58
Properties of connector configuration properties must be handled to include possible inclusion of multibyte characters.	"Connector configuration properties" on page 59
Other locale-specific tasks must be considered.	"Other locale-sensitive tasks" on page 59

Text strings: It is good programming practice to design a connector so that it refers to an external message file when it needs to obtain text strings rather than hardcoding text strings in the connector code. When a connector needs to generate a text message, it retrieves the appropriate message by its message number from the message file. Once all messages are gathered in a single message file, this file can be localized by having the text translated into the appropriate language or languages.

This section provides the following information on how to internationalize text strings:

- "Handling logging and tracing"
- "Handling miscellaneous strings" on page 58

Handling logging and tracing: To internationalize the logging and tracing, make sure that all these operations use message files to generate text messages. By putting message strings in a message file, you assign a unique identifier to each message. Table 16 lists the types of operations that use a message file and the associated C++ connector library methods in the `GenGlobals` class that the application-specific component uses to retrieve their messages from a message file.

Table 16. Methods to log and trace messages from a message file

Message-file operation	Connector library method
Logging	`generateAndLogMsg()`
Tracing	`generateAndTraceMsg()` or `traceWrite()`

Log messages should display in the language of the customer's locale. Therefore, log messages should always be isolated into a connector message file and retrieved with the `generateAndLogMsg()` method.

Because trace messages are intended for the product debugging process, they often do not need to display in the language of the customer's locale. Therefore, whether trace messages are contained in a message file is left at the discretion of the developer:

- If non-English-speaking users need to view trace messages, you need to internationalize these messages. Therefore, you must put the trace messages in a message file and extract them with the `generateMsg()` method. This message file should be the connector message file, which contains message specific to your

connector. The `generateMsg()` method generates the message string for `traceWrite()`. It retrieves a predefined trace message from a message file, formats the text, and returns a generated message string.

- If only English-speaking users need to view trace messages, you do not need to internationalize these messages. Therefore, you can include the trace message (in English) directly in the call to `traceWrite()`. You do *not* need to use the `generateMsg()` method.

  However, storing trace messages in the message file makes it easy to locate and maintain them.

Handling miscellaneous strings: In addition to handling the message-file operations in Table 16,, an internationalized connector must *not* contain any miscellaneous hardcoded strings. You must isolate these strings into the message file as well. Table 17 shows the method that the application-specific component can use to retrieve a message from a message file.

Table 17. Method to retrieve a message from the message file

Connector library class	Method
`GenGlobals`	`generateMsg()`

To internationalize hardcoded strings, take the following steps:

- Generate a uniquely numbered message in the connector message file for the hardcoded string.

  Note: In the message file, you can also include an optional explanation to the isolated string. In this explanation, you can put the method name where the string is used. This information can help to track the position of the source and make changes when needed.

- In the application-specific component, use the `generateMsg()` method to specify the isolated string by its message number.

For example, suppose your application-specific component contains the following hardcoded string in a line of code:

```
********Before updating the event status********
```

To isolate this hardcoded string from the connector code, create a message in the message file and assign it a unique message number (100):

```
100
********Before updating the event status********
[EXPL]
Hardcoded message in pollForEvents()
```

The application-specific component retrieves the isolated string (message 100) from the message file and replaces the hardcoded string with this retrieved string:

```
char * msg;
//retrieve the message numbered '100'
msg = generateMsg(100, CxMsgFormat::XRD_INFO, NULL, 0, NULL);
MyClassObject::formatMsg(msg); // send retrieved message to a custom method
```

For more information on the use of message files, see Chapter 6, “Message logging,” on page 135.

Business object locales: The connector might need to perform locale-sensitive processing (such as data format conversions) when it converts from application

data to the application-specific business object. During processing of a business object in a connector, there are two different locale settings:

- The connector inherits a locale, called the *connector-framework locale*, from the connector framework with which it runs. The connector-framework locale determines the locale of text messages that the connector uses for logging and exceptions.
- The connector also can access the locale that is associated with a business object it is processing. This *business-object locale* identifies the locale associated with the data in the business object.

Table 18 shows the method that the connector can use to retrieve the locale associated with the connector framework.

Table 18. Method to retrieve the connector framework's locale

Connector library class	Method
`GenGlobals`	`getLocale()`

When a business object is created, it can have a locale associated with its data. Your connector can access this business-object locale in either of the following ways:

- To obtain the name of the business-object locale, use the `getLocale()` method, which is defined in the `BusinessObject` class.
- To associate a locale with the business object, use the `BusinessObject()` constructor, which is also defined in the `BusinessObject` class.

Connector configuration properties: As discussed in "Using connector configuration property values" on page 69,, an application-specific component can use two types of configuration properties to customize its execution:

- Standard configuration properties are available to all connectors.
- Connector-specific configuration properties are unique to the particular connector in which they are defined.

The names of all connector configuration properties must use *only* characters defined in the code set associated with the U.S English (`en_US`) locale. However, the values of these configuration properties can contain characters from the code set associated with the connector framework locale.

The application-specific component obtains the values of configuration properties with the methods described in "Retrieving connector configuration properties" on page 71.. These methods correctly handle characters from multibyte code sets. However, to ensure that your connector is internationalized, its code must correctly handle these configuration-property values once it retrieves them. The application-specific component must *not* assume that configuration-property values contain only single-byte characters.

Other locale-sensitive tasks: An internationalized connector must also handle the following locale-sensitive tasks:

- Sorting or collation of data: the collaboration must use a collation order appropriate for the language and country of the locale.
- String processing (such as comparison, substrings, and letter case): the collaboration must ensure that any processing it performs is appropriate for characters in the locale's language.

- Formats of dates, numbers, and times: the collaboration must ensure that any formatting it performs is appropriate for the locale.

Character-encoding design principles

If data transfers from a location that uses one code set to a location that uses a different code set, some form of character conversion needs to be performed for the data to retain its meaning. The Java runtime environment within the Java Virtual Machine (JVM) represents data in Unicode. The Unicode character set is a universal character set that contains encodings for characters in most known character code sets (both single-byte and multibyte). There are several encoding formats of Unicode. The following encodings are used most frequently within the integration business system:

- Universal multiple octet Coded Character Set: UCS-2

  The UCS-2 encoding is the Unicode character set encoded in 2 bytes (octets).
- UCS Transformation Format, 8-bit form: UTF-8

  The UTF-8 encoding is designed to address the use of Unicode character data in UNIX environments. It supports all ASCII code values (0...127) so that they are never interpreted as anything except a true ASCII code. Each code value is usually represented as a 1-, 2-, or 3- byte value.

Most components in the WebSphere business integration system, including the connector framework, are written in Java. Therefore, when data is transferred between most system components, it is encoded in the Unicode code set and there is no need for character conversion.

However, a C++ connector works with a C++ application (or technology). This application (or technology) might not have data already in the Unicode code set. Therefore, the application-specific component of the connector (written in C++, and converted by the connector framework to Java) might need to perform character conversion on application data for the application-specific business object. Figure 23 shows the character encoding for a C++ connector.

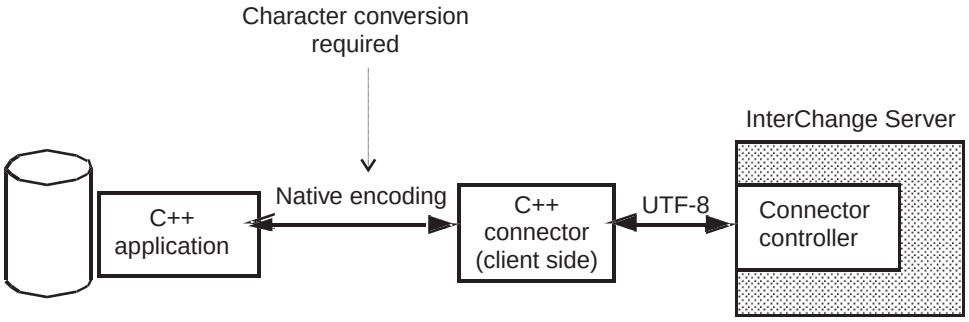

Figure 23. Character encoding with a C++ connector

Note: A connector obtains the character encoding of its application from the `CharacterEncoding` connector configuration property. If your connector performs character conversion, make sure you instruct the connector end user to set this connector property to the correct value.

To obtain the character encoding at runtime, Table 19 shows the method in the C++ connector library that the connector can use.

Table 19. Method to retrieve the connector framework's character encoding

Connector library class	Method
`GenGlobals`	`getEncoding()`

Note: Connector configuration properties with String values do not require character conversion because they originate from the InterChange Server repository and are therefore in the UCS-2 encoding.

Chapter 3. Providing general connector functionality

This chapter presents information on how to implement a *connector class*, which performs the initialization and setup for the application-specific component of a connector. It also discusses some basic functionality that your connector might need.

Note: Writing code for the application-specific component is only one part of the overall task for developing a connector. Before you begin to write your application-specific component, you should clearly understand the connector design issues as well as the design of any application-specific business objects. A thorough understanding of the design issues can help you complete the coding task successfully. For information on connector design, refer to Chapter 2, "Designing a connector," on page 37.

This chapter contains the following sections:

Running a connector

When the connector runs, it performs the tasks summarized in Table 20..

Table 20. Steps for executing a connector

	Execution step	For more information
1.	Start the connector with the startup script to initialize the connector framework and application-specific component of the connector.	"Starting up a connector" on page 63
2.	If polling is turned on, the connector framework calls `pollForEvents()` at the interval defined by the connector's `PollFrequency` connector configuration property.	"Polling for events" on page 67
3.	If the connector implements request processing, call the business-object handler associated with the request business object that the connector receives.	Request processing is implemented by the `doVerbFor()` method in the connector's business object handler. For more information, see Chapter 4, "Request processing," on page 73.
4.	When the connector is shut down, the connector framework calls `terminate()`.	"Shutting down the connector" on page 68

The following sections provide more information about each of the execution steps Table 20..

Starting up a connector

Each connector has a connector startup script to begin its execution. This startup script invokes the connector framework.

Note: For more information on how to create a connector startup script, see "Creating startup scripts" on page 206..

Once the connector framework is executing, it performs the appropriate steps to invoke the application-specific component of the connector, based on the integration broker.

Starting connectors with InterChange Server

When InterChange Server is the integration broker, the connector framework takes the following steps to invoke the application-specific component:

1. Use the Object Request Broker (ORB) to establish contact with InterChange Server.
2. From the repository, load the following connector-definition information into memory for the connector's process:
   - The connector configuration properties
   - A list of the connector's supported business object definitions
3. Begin execution of the connector's application-specific component by instantiating the connector base class and calling methods of this base class that initialize the application-specific component.

   When the connector is started, the connector framework instantiates the connector base class and then calls the connector-base-class methods in Table 21..

Table 21. Beginning execution of the connector

Initialization task		For more information
1.	Initialize the connector to perform any necessary initialization for the application-specific component, such as opening a connection to the application.	"Initializing the connector" on page 64
2.	For each business object that the connector supports, obtain the business object handler.	"Obtaining the business object handler" on page 66

   Once these methods have been called, the connector is operational.
4. Contact the connector controller to obtain the subscription list for business objects to which collaborations have subscribed. For more information, see "Business object subscription and publishing" on page 13..

Starting connectors with other integration brokers

When a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server is the integration broker, the connector framework takes the following steps to invoke the application-specific component:

1. From the local repository, load the following connector-definition information into memory for the connector's process:
   - The connector configuration properties
   - A list of the connector's supported business object definitions
2. Begin execution of the connector's application-specific component by instantiating the connector base class and calling methods of this base class that initialize the application-specific component.

   When the connector is started, the connector framework instantiates the connector base class and then calls the connector-base-class methods in Table 21.. Once these methods have been called, the connector is operational.

Initializing the connector

To begin connector initialization, the connector framework calls the initialization method of the connector base class. Table 22 shows the initialization method for the connector.

Table 22. Connector base class methods to initialize the connector

Class	Method
GenGlobals	init

As part of the implementation of the connector class, you *must* implement an initialization method for your connector. The main tasks of the initialization method include:

- "Establishing a connection"
- "Checking the connector version"
- "Recovering In-Progress events"

Important: During execution of the initialization method, business object definitions and the connector framework's subscription list are *not* yet available.

Establishing a connection: The main task of the initialization method is to establish a connection to the application. To establish the connection, the initialization method can perform the following tasks:

- Read from the repository the connector's configuration properties that provide connector information (such as ApplicationUserID and ApplicationPassword) and use them to send login information to the application. If a required connector property is empty, your initialization method can provide a default value.

  Use the getConfigProp() method to obtain the value of a connector configuration property. For more information, see "Using connector configuration property values" on page 69.
- Obtain any required connections or files. For example, the initialization method usually opens a connection with the application. It returns "success" if the connector succeeds in opening a connection. If the connector *cannot* open a connection, the initialization method must return the appropriate failure status to indicate the cause of the failure.

  In a C++ connector, typical return codes used in init() are BON_SUCCESS, BON_FAIL, and BON_UNABLETOLOGIN. For information on these and other return codes, see "C++ return codes" on page 195.

Checking the connector version: The getVersion() method returns the version of the connector. It is called in both of the following contexts:

- The initialization method should call getVersion() to check the connector version.
- The connector framework calls the getVersion() method when it needs to get a version for the connector.

Note: A connector should keep track of which application versions it supports. It should check the application version when it logs on to the application.

Recovering In-Progress events: Processing an event during event notification includes performing a retrieve on the application entity, creating a new business object for the event, and sending the business object to the connector framework. If the connector terminates while processing an event and before updating the event status to indicate that the event was either sent or failed, the In-Progress event will remain in the event store. When a connector is restarted, it should check the event store for events that have an In-Progress status.

If the connector finds events with the In-Progress status, it can choose to do one of the tasks outlined in Table 23.. This behavior should be configurable. Several connectors use the `InDoubtEvents` connector configuration property for this purpose. Its settings are also shown in Table 23..

Table 23. Actions to take to recover In-Progress events

Event-recovery action taken	Value of InDoubtEvents
Change the status of the In-Progress events to Ready-for-Poll so they can be submitted to the connector framework in subsequent poll calls. **Note:** If events are resubmitted, duplicate events might be generated. If you want to ensure that duplicate events are not generated during recovery, use another recovery response.	`Reprocess`
Log a fatal error, shutting down the connector. If `LogAtInterchangeEnd` is set to True, this triggers an email notification about the error.	`FailOnStartup`
Log an error without shutting down the connector.	`LogError`
Ignore the In-Progress event records in the event store.	`Ignore`

For a C++ connector, you must use the application-specific interface to obtain event records with an In-Progress status from the event store and take the appropriate recovery action.

Note: For more information on event notification, the event store, and In-Progress events, see Chapter 5, "Event notification," on page 109.

Obtaining the business object handler

As the final step in connector initialization, the connector framework obtains the business object handler for each business object definition that the connector supports. A *business object handler* receives request business objects from the connector framework and performs the verb operations defined in these business objects. Each connector must have a `getBOHandlerforBO()` method defined in its connector base class to retrieve the business object handler. This method returns a reference to the business object handler for a specified business object definition.

Important: As part of the implementation of the connector base class, you *must* implement the `getBOHandlerforBO()` to obtain business object handlers for your connector.

To instantiate the business object handler (or business object handlers), the connector framework takes the following steps:

1. During initialization, the connector framework receives a list of business object definitions that the connector supports. For more information, see "Starting up a connector" on page 63.
2. The connector framework then calls the `getBOHandlerforBO()` method, once for every supported business object.
3. The `getBOHandlerforBO()` method instantiates the appropriate business object handler for that business object, based on the name of the business object definition it receives as an argument. It returns the business object handler to the connector framework.

   The number of business object handlers that are instantiated depends on the overall design of your connector's business object handling:

- If the business object definitions for application-specific business objects contain metadata that follows consistent rules, the connector is metadata-driven. It can be designed to use a *metadata-driven business object handler*.

  A metadata-driven connector handles *all* business objects in a single, generic business object handler, called a metadata-driven business object handler. Therefore, the `getBOHandlerforBO()` method can simply instantiate one business object handler, regardless of the number of business objects the connector supports. It can create a business object handler the first time it is called and return a pointer to the same handler for each subsequent call.
- If some or all application-specific business objects require special processing, then you must set up *multiple business object handlers* for those objects.

  If your connector requires a separate business object handler for each business object, the `getBOHandlerforBO()` method can instantiate the appropriate business object handler, based on the name of the business object being passed in. In this case, `getBOHandlerforBO()` instantiates multiple business object handlers, one for each business object definition that requires a separate business object handler. Each time the business-object-handler retrieval method is called, it instantiates a separate business object handler.

4. The connector framework stores the reference to this business object handler in the associated business object definition (which resides in the memory of the connector's process).

Important: Before you implement the `getBOHandlerforBO()` method, you want to complete the design for business object handling for your connector. For information on designing application-specific business object, see "Assessing support for metadata-driven design" on page 47..

For more information on how to implement the `getBOHandlerforBO()`method, see "Obtaining the C++ business object handler" on page 148.. For information on how to implement business object handlers, see Chapter 4, "Request processing," on page 73.

Polling for events

If a connector is to implement event notification, it must implement an event notification mechanism. Event notification contains methods that interact with an application to detect changes to application business entities. These changes are represented as events, which the connector sends to the connector framework for routing to a destination (such as InterChange Server).

If the connector uses a polling mechanism for event notification, the connector must implement the `pollForEvents()` method to periodically to retrieve event information from the event store, which holds events that the application generates until the connector can process them. When polling is turned on, the connector framework calls the poll method `pollForEvents()`. The `pollForEvents()` method returns an integer indicating the status of the polling operation.

In the C++ connector library, the `pollForEvents()` method is defined in the `GenGlobals` class. Typical return codes used in `pollForEvents()` are `BON_SUCCESS`, `BON_FAIL`, and `BON_APPRESPONSETIMEOUT`. For more information on return codes, see "C++ return codes" on page 195.

Important: The developer must provide an implementation of the `pollForEvents()` method. If the connector supports *only* request processing, you do not

need to fully implement `pollForEvents()`. However, because the poll method is a required method, you must implement a stub for the method.

For a more thorough discussion of event notification and the implementation of `pollForEvents()`, see Chapter 5, "Event notification," on page 109.

Shutting down the connector

The administrator shuts down a connector with by terminating the connector startup script. When the connector is shut down, the connector framework calls the `terminate()` method of the connector base class. The main task of the `terminate()` method is to close the connection with the application and to free any allocated resources.

Extending the connector base class

To create a connector, you extend the *connector base class*, available in the connector library. The base class for the connector includes methods for initialization and setup of the connector's application-specific component. Your derived *connector class* contains the code for the application-specific component of the connector.

Note: For information on naming conventions for a connector, see *Naming IBM WebSphere InterChange Server Components* in the IBM WebSphere InterChange Server documentation set.

The connector base class includes the methods shown in Table 24.. You must implement these methods in your connector.

Table 24. Methods to implement in the connector base class

Description	Connector base class method	For more information
Initializes the connector's application-specific component	`init()`	"Initializing the connector" on page 64
Returns the version of the connector	`getVersion()`	"Checking the connector version" on page 65
Sets up one or more business object handlers	`getBOHandlerforBO()`	"Obtaining the business object handler" on page 66
Polls for application events	`pollForEvents()`	"Polling for events" on page 67
Performs cleanup tasks upon connector termination	`terminate()`	"Shutting down the connector" on page 68

Figure 24 illustrates the complete set of methods that the connector framework calls, and shows which methods are called at startup and which are called at runtime. All but one of the methods that the connector framework calls are in the connector base class. The remaining method, `doVerbFor()`, is in the business object handler class; for information on implementing the `doVerbFor()` method, see Chapter 4, "Request processing," on page 73.

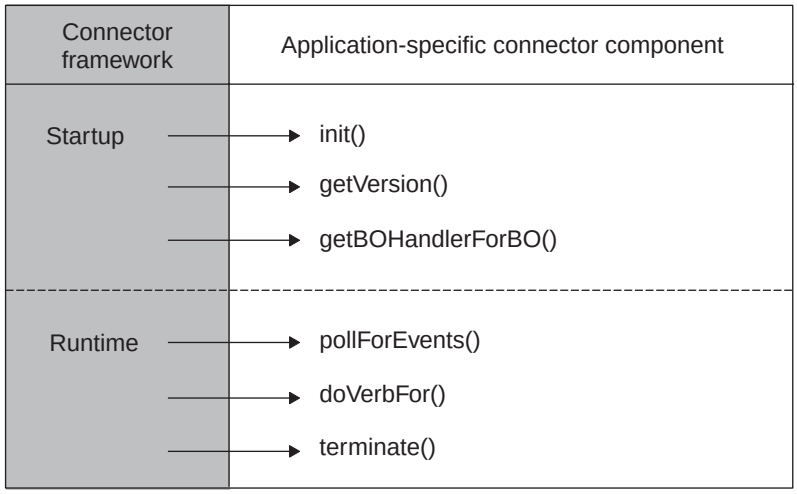

Figure 24. Summary of methods called by the connector framework

For more information on extending the connector base class, see "Extending the C++ connector base class" on page 145.

Handling errors

The methods of the connector class library indicate error conditions in the following ways:

- Return codes—The connector class library includes a set of defined outcome-status values that your virtual methods can use to return information on the success or failure of a method. The return codes are defined as integer values and outcome-status constants. In your code, IBM recommends use of the predefined constants to prevent a problem if the IBM changes the values of the constants.

  For information on C++ return codes, see "C++ return codes" on page 195.
- Return-status descriptor—during request processing, the connector framework sends status information back to the integration broker in a return-status descriptor. The business object handler can save a message and status code in this descriptor to provide the integration broker about the status of the verb processing. For more information, see "Return-status descriptor" on page 197.
- Error and message logging—The connector class library also provides the following features to assist in providing notification of errors and noteworthy conditions:
  - Logging allows you to send an informational or error message to a log destination.
  - Tracing allows you to include statements in your code that generate trace messages at different trace levels.

  For more information on how to implement logging and tracing, see Chapter 6, "Message logging," on page 135.

Using connector configuration property values

This section provides the following information about connector configuration properties:

• "Retrieving connector configuration properties" on page 71

What is a connector configuration property?

A *connector configuration property* (sometimes called just a *connector property*) allows you to create named place holders (similar to variables) that the connector can use to access information it needs. Connectors have two categories of configuration properties:

- Standard configuration properties
- Connector-specific configuration properties

Standard connector configuration properties

Standard configuration properties provide information that is typically used by the connector framework. These properties are usually common to *all* connectors and usually represent well-defined behavior that is the WebSphere business integration system enforces.

Connector-specific configuration properties

Connector-specific configuration properties provide information needed by a particular connector at runtime. These configuration properties provide a way of changing static information or logic within the connector's application-specific component without having to recode and rebuild it. For example, configuration properties can be used to:

- Hold the value of constants, such as the name of the application server or database, the name of the event table, or the name of files the connector needs to read.
- Set behavior for the connector in a particular situation. For example, a configuration property can indicate that the connector should not fail a business object Retrieve operation for a hierarchical business object if a child object is missing. As another example, a configuration property can determine whether the application or the connector should create an ID for a new object on a Create operation.

You can create any number of connector-specific configuration properties for your connector. When you have identified needed connector-specific properties, you define them as part of the connector configuration process. Use Connector Configurator to specify connector configuration properties as part of the information stored in the local repository.

You can also add configuration properties later on as needed. In general, your connector code needs only to query for the values of the connector-specific properties such as `ApplicationUserID` and `ApplicationPassword`.

Defining and setting connector configuration properties

The Connector Configurator tool provides you with the ability to perform the following tasks on connector configuration properties:

- Assign a value to a standard configuration property.
- Define and assign a value to a connector-specific configuration property.

  You invoke Connector Configurator from the System Manager tool.

WebSphere InterChange Server

If WebSphere InterChange Server is your integration broker, refer to the *Implementation Guide for WebSphere InterChange Server* for information about the Connector Configurator tool.

Other integration brokers

If a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) is your integration broker, refer the *Implementation Guide for WebSphere Message Brokers* for information about Connector Configurator. If WebSphere Application Server is your integration broker, refer to the *Implementation Guide for WebSphere Application Server* for information about Connector Configurator.

Retrieving connector configuration properties

Connector configuration properties are downloaded to the connector as part of the connector initialization (For more information, see "Starting up a connector" on page 63). Your connector application-specific component retrieves the values of any configuration properties that it needs for initialization based on the type of the connector property.

A connector can use a connector configuration property that has one of the following types:

- A *simple* connector configuration property contains only string values. It does *not* contain any other properties. A single-valued simple property contains only one string value.
- A *hierarchical* connector configuration property contains other properties and their values. A given connector property can contain multiple values.

Note: For the IBMWebSphere Business Integration Adapters product, single-valued simple connector configuration properties are the only kind of connector properties that a C++ connector supports. C++ connectors do *not* support hierarchical properties.

To retrieve a single-valued simple connector configuration property, you can use the `getConfigProp()` method. In the C++ connector library, the `getConfigProp()` method takes as input a character string for the name of the property value, which is case-sensitive. It also takes a pointer to a buffer to which the method can write the property value and the number of bytes in the buffer. The `getConfigProp()` method is defined in the `GenGlobals` class. The code fragment below shows the use of `getConfigProp()` to retrieve the value of a `Hostname` configuration property:

```
char * hostnameBuffer = new char[512];
if (getConfigProp("Hostname", hostnameBuffer, 511) == 0)
     logMsg("Invalid or empty property name.");
else
     hostName = hostnameBuffer;
```

Handling loss of connection to an application

A good design practice is to code the connector application-specific code so that it shuts down whenever the connection to the application is lost. To respond to a lost connection, the connector's application-specific component should take the following steps:

- Log a fatal error message so that email notification is triggered if the `LogAtInterchangeEnd` connector configuration property is set to True.
- Return the `BON_APPRESPONSETIMEOUT` outcome status inform the connector controller that the application is not responding. When this occurs, the process in which the connector runs is stopped. A system administrator must fix the problem with the application and restart the connector to continue processing events and business object requests.

The following user-implemented virtual methods should check for a loss of connection to the application:

- For event notification, the `pollForEvents()` method should verify the connection before it accesses the event store. For more information, see "Verifying the connection before accessing the event store" on page 179.
- For request processing, the `doVerbFor()` method should verify the connection before it begins verb processing. For more information, see "Verifying the connection before processing the verb" on page 152.

Chapter 4. Request processing

This chapter presents information on how to provide request processing in a connector. *Request processing* implements a mechanism to receive requests, in the form of request business objects, from an integration broker and to initiate the appropriate changes in the application business entities. The mechanism for implementing request processing is a *business object handler*, which contains methods that interact with an application to transform request business objects into requests for application operations. This chapter contains the following sections:

Note: For an introduction to request processing, see "Request processing" on page 25..

Designing business object handlers

The business object handler implements request processing for the connector. Therefore, the defining and coding of business object handlers is one of the primary tasks in connector development. A business object handler is an instance of a subclass of the `BOHandlerCPP` class. Each business object definition refers to a business object handler, which contains a set of methods to perform the tasks for the verbs that the business object definition supports. You need to code one or more business object handlers to process the business objects that the connector supports.

The way to implement a business object handler depends on the application programming interface (API) that you are using and how this interface exposes application entities. To determine how many business object handlers your connector requires, you need to take a look at the application that the connector will interact with:

- If the application is form-based, table-based, or object-based and has a standard access method across entities, you might be able to design business objects that store information about application entities. The business object handler can process the application entities in a *metadata-driven business object handler*.

  You can derive one generic business-object-handler class to implement a metadata-driven business object handler, which handles processing of *all* business objects. For more information, see "Implementing metadata-driven business object handlers" on page 74.

- If the application has different access methods for different kinds of entities, some or all of the application entities might require individual business object handlers.

  You can:

  – Derive a generic business-object-handler class to implement a metadata-driven business object handler for some business objects, and separate business-object-handler classes to implement business object handlers for other business objects.
  – Derive multiple business-object-handler classes, one for each business object definition that the connector supports.

  For more information, see "Implementing multiple business object handlers" on page 75.

Implementing metadata-driven business object handlers

If the application API is suitable for a metadata-driven connector, and if you design business object definitions to include metadata, you can implement a metadata-driven business object handler. This business object handler uses the metadata to process all requests. A business object handler can be completely metadata-driven if the application is consistent in its design, and the metadata follows a consistent syntax for each supported business object.

Note: For an introduction to metadata and metadata-driven design, see "Assessing support for metadata-driven design" on page 47..

This section provides the following information about metadata-driven design for a business object handler:

- "Metadata in business objects"
- "Benefits of metadata design" on page 75

Metadata in business objects

Business object definitions have specific locations for different types of application-specific data. For example, business object attributes have a set of properties, such as Key, Foreign Key, Required, Type, and so on, that provide the business object handler with information that it can use to drive business object processing. In addition, the `AppSpecificInfo` property can provide the business object handler with application-specific information, which can specify how to access data in the application and how to process application entities.

The `AppSpecificInfo` property is available for the business object definition, attributes, and verbs. Table 25 shows some typical schemes for encoding application-specific information in business objects.

Table 25. Example schemes for storage of application information in business objects

Scope of application-specific information	Table-based application	Form-based application
The whole business object	Table name	Form name
An individual attribute	Column name	Field name
The business object verb	SQL statement or other verb-processing instructions	Action to be performed

Using application-specific information, a metadata-driven business object handler might simply:

- Examine the verb of an incoming business object to identify the operation to perform.
- Examine the contents of the business object metadata to identify the name of the associated application entity (such as an application table or form).
- Examine the contents of the attribute metadata to identify fields, columns, or other information about the attributes.

If a business object definition contains the table name and column names, you do not have to explicitly code those names in the business object handler.

Benefits of metadata design

Encoding application information in a business object accomplishes two things:

- One business object handler class can perform *all* operations for *all* business objects supported by the connector. You do not have to code a separate business object handler for each supported business object.
- Changes to a business object definition do not require recoding the connector as long as the changes conform to existing metadata syntax. This benefit means that you can add attributes to a business object definition, remove attributes, or reorder attributes without recompiling or recoding the connector.

If information about application entities is encoded consistently in the business object definition, all request business objects can be handled by a single business-object-handler class in the connector. Also, you need to implement only a single `getBOHandlerforBO()` method to return the single business object handler and a single `doVerbFor()` method to implement this business object handler. This approach is recommended for connector development because it provides flexibility and automatic support for new business object attributes.

Implementing multiple business object handlers

For each business object definition that does *not* encapsulate all the metadata and business logic for an application entity, you need a separate business-object-handler class. You can derive separate handler classes directly from the business-object-handler base class, or you can derive a single utility class and derive subclasses as needed. You must then implement the `getBOHandlerforBO()` method to return business object handler that corresponds to particular business object definitions.

Each business object handler must contain a `doVerbFor()` method. If you implement multiple business object handlers, you must implement a `doVerbFor()` method for each business-object-handler class. In each `doVerbFor()` method, include code to handle any parts of the application entity or operations on the application entity that the business object definition does not describe.

This approach results in higher maintenance requirements and longer development time than designing a single business object handler for a metadata-driven connector. For this reason, this approach should be avoided if possible. However, if the application has different access methods for different kinds of entities, coding multiple, entity-specific business object handlers might be unavoidable.

Extending the business-object-handler base class

The C++ connector library provides the business-object-handler base class, `BOHandlerCPP`. This base class includes methods for handling request processing, including the `doVerbFor()` method. To create a business object handler, you must extend this business-object-handler base class and implement its virtual method `doVerbFor()`. For information specific to the C++ connector library, see "Extending the C++ business-object-handler base class" on page 149.

Handling the request

Once you have derived your business-object-handler class, you must implement the business-object-handler method, `doVerbFor()`. It is the `doVerbFor()` method that provides request processing for the business objects that the connector supports. At startup, the connector framework calls `getBOHandlerforBO()` to obtain the business object handler implemented for each of the business object definitions that the connector supports.

Important: All connectors *must* implement a business-object-handler method, `doVerbFor()`, that implements the Retrieve verb. This method and verb *must* be implemented even if your connector will *not* perform request processing.

This section provides the following information on how to implement the `doVerbFor()` method:

- "Basic logic for doVerbFor()"
- "General recommendations on verb implementations" on page 78

Basic logic for doVerbFor()

For a C++ connector, the `BOHandlerCPP` class defines the `doVerbFor()` method, which is a virtual method defined. The `doVerbFor()` method typically follows a basic logic for request processing.

Figure 25 shows a flow chart of the method's basic logic.

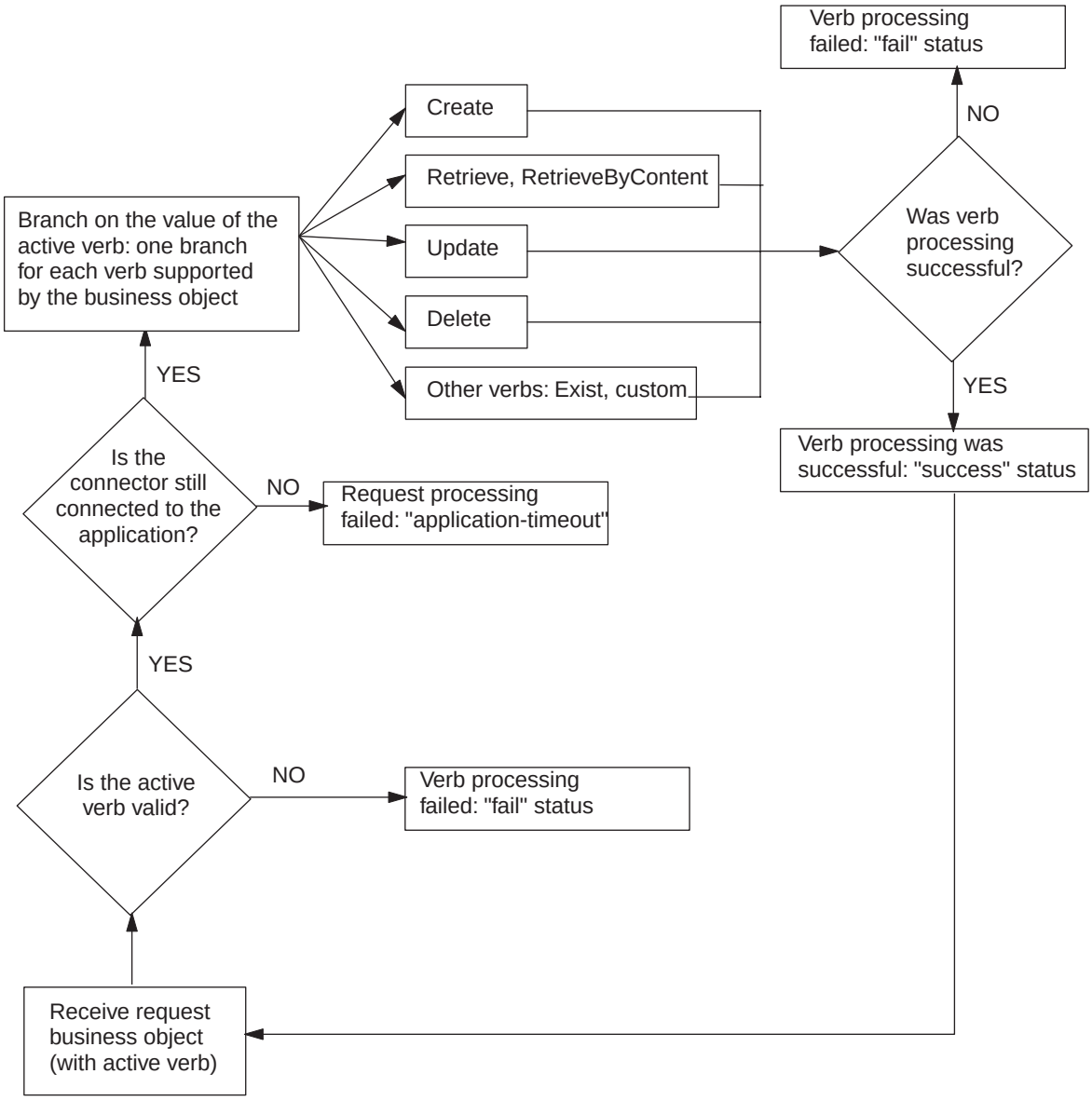

Figure 25. Flow chart for basic logic of doVerbFor()

For an implementation of this basic `doVerbFor()` logic, see "Implementing the doVerbFor() method" on page 150.

When the connector framework receives a request, it calls the `doVerbFor()` method for the business-object-handler class associated with the business object definition of the request business object. To this `doVerbFor()` method, the connector framework passes the request business object. Table 26 summarizes the tasks that the `doVerbFor()` method performs once it has received a request business object from the connector framework.

Table 26. Tasks of the doVerbFor() method

Task of business object handler	For more information
1. Determine the verb processing to perform, based on the active verb in the request business object.	"Performing the verb action" on page 79

Table 26. Tasks of the doVerbFor() method (continued)

Task of business object handler	For more information
2. Obtain information from the request business object to build and send requests for operations to the application.	"Processing business objects" on page 100

General recommendations on verb implementations

This section provides the following general recommendations for implementing your `doVerbFor()` method:

- "Verb stability"
- "Transaction support"
- "ObjectEventId attribute"

Verb stability

Verbs in a business object should remain stable throughout the request and response cycle. When a connector receives a request, the hierarchical business object that is returned to InterChange Server should have the same verbs as the original request business object, with the exception of verbs in child business objects that were *not* set in the original request.

Verbs in child business objects might or might not be set in request business objects:

- When a verb is set in a child business object, the connector should perform the operation that the child verb indicates, regardless of the verb on the top-level business object.
- If a verb in a child business object request is *not* set, the connector can either leave the child verb as `NULL`, set the child verb to the verb in the top-level business object, or set the value of the verb to the operation that the connector needs to perform.

Transaction support

An entire business object request must be wrapped in a single transaction. In other words, all Create, Update, and Delete transactions for a top-level business object and all of its children must be wrapped in a single transaction. If any failure is detected during the life of the transaction, the whole transaction should be rolled back.

For example, if a Create operation on a top-level business object succeeds, but the transaction for one of the child business objects fails, the connector application-specific component should roll back the entire Create transaction to the previous state. In this case, the connector's application-specific component should return failure from the verb method.

ObjectEventId attribute

The `ObjectEventId` attribute is used in the IBM WebSphere business integration system to identify an event-trigger flow in the system. In addition, it is used to keep track of child business objects across requests and responses, as the position of child business objects in a hierarchical business object request might be different from the position of the child business objects in the response business object.

Connectors are *not* required to populate `ObjectEventId` attributes for either a parent business object or its children. If business objects do not have values for `ObjectEventId` attributes, the IBM WebSphere business integration system

generates values for them. When connectors generate ObjectEventId values, this is done by the source connector as part of the event-notification mechanism.

When processing request business objects, connectors should preserve ObjectEventId values in *all* levels of a hierarchical business object between the request business object and the response business object. If a connector method changes the values of child business object ObjectEventIds, the IBM WebSphere business integration system may not be able to correctly track the child business objects.

For information on generating ObjectEventIds in the event notification mechanism, see "Event identifier" on page 111..

Performing the verb action

The standard verbs that IBM WebSphere business integration system expect connectors to handle are Create, Retrieve, Update, and Delete. IBM recommends that you implement these verbs according to standard behaviors documented in the sections listed in the For More Information column of Table 27.. These sections provide information about the standard behavior, implementation notes, and the appropriate outcome-status values.

Table 27 lists the standard verbs that IBM WebSphere business integration system defines. Your doVerbFor() method should implement those verbs appropriate for its application.

Table 27. Verbs implemented by the doVerbFor() method

Verb	Description	For more information
Create	Make a new entity in the application.	"Handling the Create verb" on page 80
Retrieve	Using key values, return a complete business object.	"Handling the Retrieve verb" on page 83
RetrieveByContent	Using non-key values, return a complete business object.	"Handling the RetrieveByContent verb" on page 89
Update	Change the value in one or more fields in the application.	"Handling the Update verb" on page 91
Delete	Remove the entity from the application. This operation must be a true physical delete.	"Handling the Delete verb" on page 98
Exists	Check whether the entity exists in the application.	"Handling the Exists verb" on page 99
Custom verbs	Perform some application-specific operation.	None

Note: Although the sections listed in the "For more information" column of Table 27 present suggested behavior for verb methods, your connector might need to implement some aspects of verb processing differently to support a particular application. Once the connector framework passes a request business object to your connector's doVerbFor() method, the doVerbFor() method can implement verb processing in any way that is necessary. Your verb processing code is not limited to the suggestions presented in this chapter.

InterChange Server

When InterChange Server is the integration broker and you design your own collaborations, you can implement any custom verbs that you need. Your

collaborations and connectors are not limited to the standard list of verbs.

End of InterChange Server

This basic verb-processing logic consists of the following steps:

1. Get the verb from the request business object.

   The `doVerbFor()` method must first retrieve the active verb from the business object with the `getVerb()` method. For a C++ connector, `getVerb()` is defined in the `BusinessObject` class.

2. Perform the verb operation.

   In the business object handler, you can design the `doVerbFor()` method in either of the following ways:

   - Implement verb processing for each supported verb directly within the `doVerbFor()` method. You can modularize the verb processing so that each verb operation is implemented in a separate verb method called from `doVerbFor()`. The method should also take appropriate action if the verb is not a supported verb by returning a message in the return-status descriptor and a "fail" status.
   - Handle all verb processing in the same method using a metadata-driven `doVerbFor()` method.

Handling the Create verb

When the business object handler obtains a Create verb from the request business object, it must ensure that a new application entity, whose type is indicated by the business object definition, is created, as follows:

- For a flat business object, the Create verb indicates that the specified entity must be created.
- For a hierarchical business object, the Create verb indicates that one or more application entities (to match the entire business object) must be created.

The business object handler must set all the values in the new application entities to the attribute values in the request business object. To ensure that all required attributes in the request business object have values assigned, you can call the `initAndValidateAttributes()` method, which assigns the attribute's default value to each required attribute that does not have its value set (when the `UseDefaults` connector configuration property is set to `true`). The `initAndValidateAttributes()` method is defined in the `BusinessObject` class. Call `initAndValidateAttributes()` *before* performing the Create operation in the application.

Note: For a table-based application, the entire application entity must be created in the application database, usually as a new row to the database table associated with the business object definition of the request business object.

This section provides the following information to help process a Create verb:

- "Standard processing for a Create verb" on page 81
- "Implementation of a Create verb operation" on page 81
- "Outcome status for Create verb processing" on page 82

Note: You can modularize your business object handler so that each supported verb is handled in a separate C++ method. If you follow this structure, a Create method handles processing for the Create verb.

Standard processing for a Create verb

The following steps outline the standard processing for a Create verb:

1. Create the application entity corresponding to the top-level business object.
2. Handle the primary key or keys for the application entity:
   - If the application generates its own primary key (or keys), get these key values for insertion in the top-level business object.
   - If the application does *not* generate its own primary key (or keys), insert the key values from the request business object into the appropriate key column (or columns) of the application entity.
3. Set foreign key attributes in any first-level child business objects to the value of the top-level primary key.
4. Recursively create the application entities corresponding to the first-level child business objects, and continue recursively creating all child business objects at all subsequent levels in the business object hierarchy.

In Figure 26,, a verb method sets the foreign key attributes (FK) in child business objects A, B, and C to the value of the top-level primary key (PK1). The method then recursively sets the foreign key attributes in child business objects D and E to the value of the primary key (PK3) in their parent business object, object B.

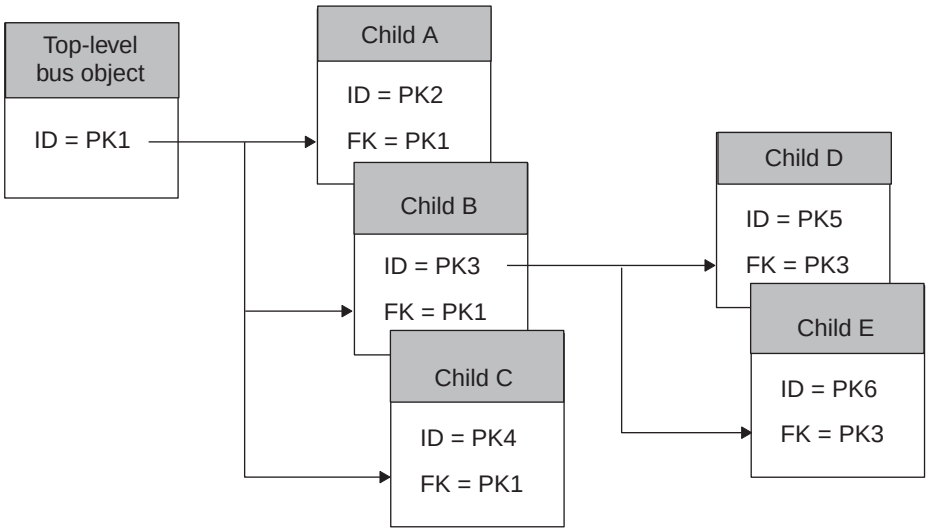

Figure 26. Creating parent/child relationships

Implementation of a Create verb operation

A typical implementation of a Create operation first traverses the top-level business object and processes the business object's simple attributes. It gets the values of the attributes from the business object and generates the application-specific action (such as an API call or SQL statement) that inserts an entity in the application to represent the top-level business object. Once this top-level entity is created, the verb operation takes the following steps:

1. Retrieve any primary keys for the entity from the application.
2. Use the keys to set the foreign key attributes in the first-level child business objects to the value of the parent primary keys.
3. Set the verb in each child business object to Create and recursively create application entities to represent the child business objects.

A recommended approach for creating child business objects is to design a submethod to recursively create child entities. The submethod might traverse the business object, looking for attributes of type `OBJECT`. If the submethod finds attributes that are objects, it calls the main Create method to create the child entities.

The way that the main method provides primary key values to the submethod can vary. For example, the main Create method might pass the parent business object to the submethod, and the submethod can then retrieve the primary key from the parent business object to set the foreign key in the child business object. Alternatively, the main method might traverse the parent object, find first-level children, set the foreign key attributes in the child business objects, and then call the submethod on each child.

In either case, the main Create method and its submethod interact to set the interdependencies between the parent business object and its first-level children. Once the foreign keys are set, the operation can:

- Insert new rows into the application.
- Set foreign keys for the next level of child business objects.
- Create the child entities.
- Descend the business object hierarchy, recursively creating child entities until there are no more child business objects to process.

Note: For an example Create verb method, see "Example: Create method for a flat business object" on page 171.

Note: For a table-based application, the order of the steps for setting the relationships between a top-level object and its children may vary, depending on the database schema for the application and on the design of the application-specific business objects. For example, if foreign keys for a hierarchical business object are located in the top-level business object, the verb operation might need to process all child business objects *before* processing the top-level business object. Only when the child entities are inserted into the application database and the primary keys for these entities are returned can the top-level business object be processed and inserted into the application database. Therefore, be sure to consider the structure of data in the application database when you implement connector verb methods.

Outcome status for Create verb processing

The Create operation should return one of the outcome-status values shown in Table 28.

Table 28. Possible outcome status for C++ Create verb processing

Create condition	C++ outcome status
If the Create operation is successful and the application generates new key values, the connector: • fills the business object with the new key values; this business object is returned to the connector framework through the request business object parameter. • returns the "Value Changed" outcome status to indicate that the connector has changed the business object	`BON_VALCHANGE`
If the Create operation is successful and the application does *not* generate new key values, the connector can simply return "Success".	`BON_SUCCESS`

Table 28. Possible outcome status for C++ Create verb processing (continued)

Create condition	C++ outcome status
If the application entity already exists, the connector can *either* of the following actions:	
• Fail the Create operation.	BON_FAIL
• Return an outcome status that indicates the application entity already exists.	BON_VALDUPES
If the Create operation fails, the verb method:	BON_FAIL
• fills a return-status descriptor with information on the failure	
• returns the "Fail" outcome status	

Note: When the connector framework receives the BON_VALCHANGE outcome status, it includes a business object in its response to InterChange Server. For more information, see "Sending the verb-processing response" on page 168.

Handling the Retrieve verb

When the business object handler obtains a Retrieve verb from the request business object, it must ensure that an existing application entity, whose type is indicated by the business object definition, is retrieved, as follows:

- For a flat business object, the Retrieve verb indicates that the specified entity is retrieved by its key values. The verb operation returns a business object that contains the current values for the application entity.
- For a hierarchical business object, the Retrieve verb indicates that one or more application entities (to match the entire business object) are retrieved by the key values of the top-level business object. The verb operation returns a business object in which all simple attributes of each business object in the hierarchy match the values of the corresponding entity attributes, and the number of individual business objects in each child business object array matches the number of child entities in the application.

Note: For a table-based application, the entire application entity must be retrieved from the application database.

For the Retrieve verb, the business object handler obtains the key value (or values) from the request business object. These key values uniquely identify an application entity. The business object handler then uses these key values to retrieve all the data associated with an application entity. The connector retrieves the entire hierarchical image of the entity, including all child objects. This type of retrieve operation might be referred to as an *after-image retrieve*.

Important: All connectors must implement a `doVerbFor()` method with verb processing for the Retrieve verb. This requirement holds even if your connector will *not* perform request processing.

An alternative way of retrieving data is to query using a subset of non-key attribute values, none of which uniquely define a particular application record. This type of retrieve processing is performed by the RetrieveByContent verb method. For information on retrieving by non-key values, see "Handling the RetrieveByContent verb" on page 89.

This section provides the following information to help process a Retrieve verb:

- "Standard processing for a Retrieve verb" on page 84

- "Implementation of a Retrieve verb operation"
- "Example: Retrieve operation"
- "Retrieving child objects" on page 86
- "Outcome status for Retrieve verb processing" on page 88

Note: You can modularize your business object handler so that each supported verb is handled in a separate C++ method. If you follow this structure, a Retrieve method handles processing for the Retrieve verb.

Standard processing for a Retrieve verb

The following steps outline the standard processing for a Retrieve verb:

1. Create a new business object of the same type as the request business object. This new business object is the *response business object*, which will hold the retrieved copy of the request business object.
2. Set the primary keys in the new top-level business object to the values of the top-level keys in the request business object.
3. Retrieve the application data for the top-level business object and fill the response top-level business object's simple attributes.
4. Retrieve *all* the application data associated with the top-level entity, and create and fill child business objects as needed.

Note: By default, the Retrieve method returns failure if it cannot retrieve application data for *all* the child objects in a hierarchical business object. This behavior can be made configurable; for information, see "Configuring a Retrieve to ignore missing child objects" on page 88.

Implementation of a Retrieve verb operation

A typical Retrieve operation can use one of the following methods:

- Create a new response business object from the business object definition for that object and sets the top-level primary keys in this new business object. Using the top-level primary keys, the verb operation can retrieve all data associated with the top-level entity.
- Start by pruning all child business objects from the top-level business object. Using the top-level keys in the pruned object, the verb operation can retrieve the top-level data and all associated data.

The goal of each of these approaches is the same: Start with the top-level business object and rebuild the complete business object hierarchy. This type of implementation ensures that *all* children in the request business object that are no longer in the database are removed and are not passed back in the response business object. This implementation also ensures that the hierarchical response business object exactly matches the database state of the application entity. At each level, the Retrieve operation rebuilds the request business object so that it accurately reflects the current database representation of the entity.

Example: Retrieve operation

In a Retrieve operation, an integration broker requests the complete set of data that is associated with an application entity. The request business object might contain any of the following:

- A top-level business object but no child objects, even though the business object definition includes children

- A business object that contains the top-level business object and some of its defined children
- A complete hierarchical business object containing all child business objects

Figure 27 shows a request business object for a Contact entity. The `Contact` business object includes a multiple cardinality array for the `ContactProfile` attribute. In this request business object, the `ContactProfile` business object array includes two child business objects.

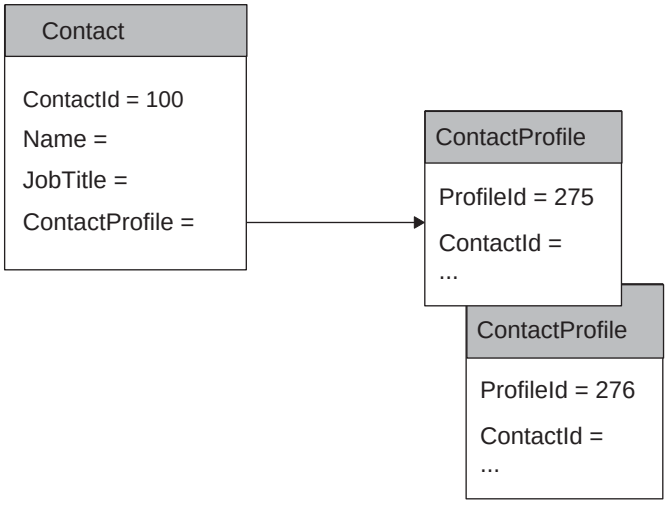

Figure 27. Example business object content for a Retrieve request

Application tables associated with the `Contact` and `ContactProfile` business objects might look like the tables in Figure 28.. This illustration also shows the foreign-key relationship between the tables. As you can see, the `contact_profile` table has a row for the `ContactId` of 100 that is *not* reflected in the `Contact` request business object in Figure 26..

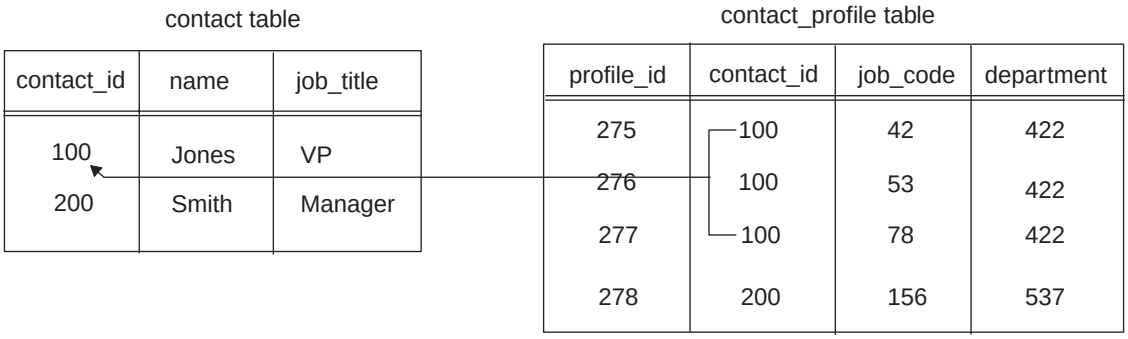

Figure 28. Foreign-key relationships between tables

The Retrieve operation uses the primary key in the `Contact` business object (100) to retrieve the data for the simple attributes in the response business object: values for the `Name` and `JobTitle` attributes. To be sure that it retrieves the correct number of child business objects, the verb operation must either create a new business object or prune child objects from the existing request business object. For the tables in Figure 28,, the Retrieve operation would need to create a new `ContactProfile` business object for the `contact_profile` row with a `profile_id`

value of 277. In this way, the Retrieve operation properly creates and populates *all* arrays based on the current state of the application entities.

Retrieving child objects

To retrieve entities associated with the top-level entity, the Retrieve operation might be able to use the application API:

- Ideally, the API will navigate the relationships between application entities and return all related data. The verb operation can then encapsulate the related data as child business objects.
- If the API does *not* provide information on associated entities, you might need to access the application (for example, with generated SQL statements) to retrieve related data. The SQL statements might use foreign keys to navigate through application tables.

If the attribute application-specific information in the business object definition contains information on foreign keys, the verb operation can use this information to generate command to access the application (such as SQL statements). For example, application-specific information for the foreign key attribute of the `ContactProfile` child business object might specify:

- The parent table: `contact`
- The child table's column for the foreign key: `contact_id`
- The attribute in the parent business object that contains the primary key value that serves as a foreign key in the child business object: `ContactId`

Figure 29 shows example application-specific information for the primary key attribute of the Contact business object and the primary and foreign key attributes of the `ContactProfile` child business object.

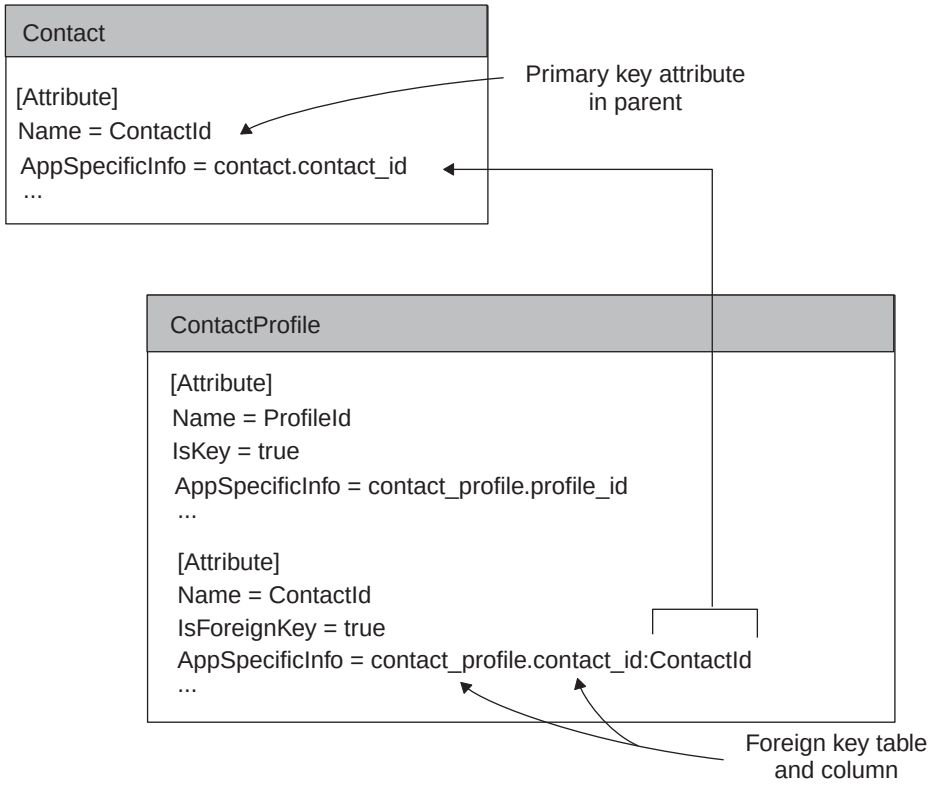

Figure 29. Foreign-key relationships in business objects

Using the application-specific information, the verb operation can find the name of the child table (`contact_profile`) and the column for the foreign key (`contact_id`) in the child table. The verb operation can also find the value of the foreign key for the child business object by obtaining the value of the primary key attribute (`ContactId`) in the parent business object (100). With this information, the verb operation can generate a SQL SELECT statement that retrieves all the records in the child table associated with the parent key. The SELECT statement to retrieve the data associated with the missing `contact_profile` row might be:

```
SELECT profile_id, job_code, department
FROM contact_profile
WHERE contact_id = 100
```

The SELECT statement returns three rows from the example `contact_profile` table, as shown in Figure 30..

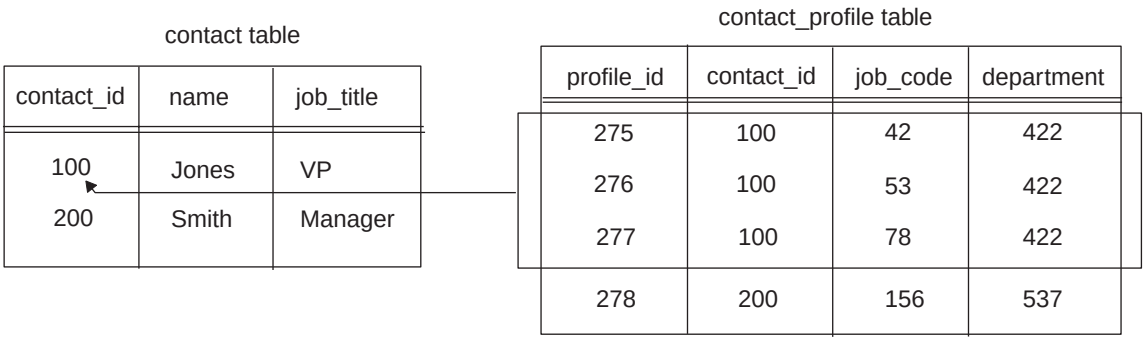

Figure 30. Results of SELECT statement for example Retrieve operation

If a Retrieve operation returns multiple rows, each row becomes a child business object. The verb operation might process retrieved rows as follows:

1. For each row, create a new child business object of the correct type.
2. Set attributes in the new child business object based on the values that a SELECT statement returns for the associated row.
3. Recursively retrieve all children of the child business object, creating the business object and setting the attributes for each one.
4. Insert the array of child business objects into the multiple-cardinality attribute in the parent business object.

The response business object for the Retrieve operation on the two example tables might look like Figure 31.. The verb operation has retrieved the current database entity and has added a child to the hierarchical business object.

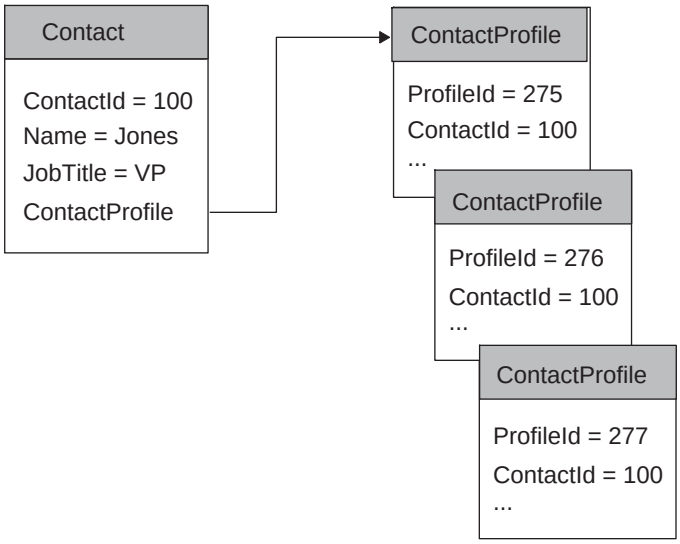

Figure 31. Business object response to example Retrieve request

Configuring a Retrieve to ignore missing child objects

By default, the Retrieve operation should return failure if it cannot retrieve application data for the complete set of child business objects in a hierarchical business object. However, you can implement the verb operation so that the behavior of the connector is configurable when one or more of the children in a business object are not found in the application.

To do this, define a connector-specific configuration property named `IgnoreMissingChildObject`, whose values are True and False. The Retrieve operation obtains the value of this property to determine how to handle missing child business objects. When the property is True, the Retrieve operation should simply move on to the next child in the array if it fails to find a child business object. In this case, the verb operation should return BON_VALCHANGE if it is successful in retrieving the top-level object, regardless of whether it is successful in retrieving its children.

Outcome status for Retrieve verb processing

The Retrieve operation should return one of the outcome-status values shown in Table 29.

Table 29. Possible outcome status for C++ Retrieve verb Processing

Retrieve condition	C++ outcome status
When the Retrieve operation is successful, it: • fills the entire business object hierarchy, including all child business objects; this business object is returned to the connector framework through the request business object parameter. • returns the "Value Changed" outcome status to indicate that the connector has changed the business object	BON_VALCHANGE
If the IgnoreMissingChildObject connector property is True, the Retrieve operation returns the "Value Changed" outcome status for the business object if it is successful in retrieving the top-level object, regardless of whether it is successful in retrieving its children.	BON_VALCHANGE
If the entity that the business object represents does *not* exist in the application, the connector returns a special outcome status rather than "Fail".	BON_BO_DOES_NOT_EXIST
If the request business object does *not* provide a key for the top-level business object, the Retrieve operation can take *either* of the following actions: • Fill a return-status descriptor with information about the cause of Request failure and return a "Fail" outcome status. • Call the RetrieveByContent method to retrieve using the content of the top-level business object.	BON_FAIL

Note: When the connector framework receives the BON_VALCHANGE outcome status, it includes a business object in its response to InterChange Server. For more information, see "Sending the verb-processing response" on page 168.

Handling the RetrieveByContent verb

An integration broker might need to retrieve a business object for which it has a set of attribute values without having the key attribute (or attributes) that uniquely identifies an application entity. Such a retrieve is called "retrieve by non-key values" or "retrieve by content." As an example, if a business object handler receives a Customer business object with the verb RetrieveByContent and with the non-key attributes Name and City set to `Smith` and `San Diego`, the RetrieveByContent operation can attempt to retrieve a customer entity that matches the values of the Name and City attributes.

When the business object handler obtains a RetrieveByContent verb from the request business object, it must ensure that an existing application entity, whose type is indicated by the business object definition, is retrieved, as follows:

- For a flat business object, the RetrieveByContent verb indicates that the specified entity is retrieved by its non-key values. The verb operation returns a business object that contains the current values for the application entity.
- For a hierarchical business object, the RetrieveByContent verb indicates that one or more application entities (to match the entire business object) are retrieved by the non-key values of the top-level business object. The verb operation returns a business object in which all simple attributes of each business object in the hierarchy match the values of the corresponding entity attributes, and the number of individual business objects in each child business object array matches the number of child entities in the application.

This section provides the following information to help process a RetrieveByContent verb:

- "Implementation for a RetrieveByContent verb operation"
- "Outcome status for RetrieveByContent processing"

Note: You can modularize your business object handler so that each supported verb is handled in a separate C++ method. If you follow this structure, a RetrieveByContent method handles processing for the RetrieveByContent verb.

Implementation for a RetrieveByContent verb operation

RetrieveByContent functions the same as the Retrieve verb except that it uses a subset of non-key values, instead of key values, to retrieve application data. In its most robust implementation, a top-level business object and its child business objects would independently support the RetrieveByContent verb. However, not all application APIs enable retrieve by non-key values for hierarchical business objects.

A more common implementation is to provide RetrieveByContent support only in the top-level business object. When a top-level business object supports retrieve by non-key values and this retrieve-by-content is successful, the RetrieveByContent operation can retrieve the keys for the entity matching the request business object. The verb operation can then perform a Retrieve operation to retrieve the complete business object.

You might want to specify which attributes are to be used in RetrieveByContent operations. To do this, you can implement attribute application-specific information to specify those attributes that will contain a value that is to be used in the RetrieveByContent operation or receive a value as a result of the operation.

Outcome status for RetrieveByContent processing

The RetrieveByContent operation should return one of the outcome-status values shown in Table 30.

Table 30. Possible outcome status for C++ RetrieveByContent verb processing

RetrieveByContent condition	C++ outcome status
If the RetrieveByContent operation finds a single entity that matches the query, it: • fills the entire business object hierarchy, including all child business objects; this business object is returned to the connector framework through the request business object parameter. • returns a "Value Changed" outcome status	BON_VALCHANGE
If the IgnoreMissingChildObject connector property is True, the RetrieveByContent operation returns the "Value Changed" outcome status for the business object if it is successful in retrieving the top-level object, regardless of whether it is successful in retrieving its children.	BON_VALCHANGE

Table 30. Possible outcome status for C++ RetrieveByContent verb processing (continued)

RetrieveByContent condition	C++ outcome status
If the RetrieveByContent operation finds multiple entries that match the query, it returns: • retrieves only the first occurrence of the match; this business object is returned to the connector framework through the request business object parameter. • fills a return-status descriptor with further information about the search • returns a status of "Multiple Hits" to notify the connector framework that there are additional records that match the specification	BON_MULTIPLE_HITS
If the RetrieveByContent operation does *not* find matches for retrieve by non-key values, it: • fills a return-status descriptor containing additional information about the cause of the RetrieveByContent error • returns a "RetrieveByContent Failed" outcome status	BON_FAIL_RETRIEVE_BY_CONTENT

Note: When the connector framework receives the BON_VALCHANGE outcome status, it includes a business object in its response to InterChange Server. For more information, see "Sending the verb-processing response" on page 168.

Handling the Update verb

When the business object handler obtains an Update verb from the request business object, it must ensure that an existing application entity, whose type is indicated by the business object definition, is updated, as follows:

- For a flat business object, the Update verb indicates that the data in the specified entity must be modified as necessary until the application entity exactly matches the request business object.
- For a hierarchical business object, the Update verb indicates that updates the application entity must be updated to match the entire business object hierarchy. To do this, the connector might need to create, update, and delete application entities:
  - If child entities exist in the application, they are modified as needed.
  - Any child business objects contained in the hierarchical business object that do *not* have corresponding entities in the application are added to the application.
  - Any child entities that exist in the application but are *not* contained in the business object are deleted from the application.

Note: For a table-based application, the entire application entity must be updated in the application database, usually as a new row to the database table associated with the business object definition of the request business object.

This section provides the following information to help process an Update verb:

Note: You can modularize your business object handler so that each supported verb is handled in a separate C++ method. If you follow this structure, an Update method handles processing for the Update verb.

Standard processing for an Update verb

The following steps outline the standard processing for an Update verb:

1. Create a new business object of the same type as the request business object. This new business object is the *response business object*, which will hold the retrieved copy of the request business object.
2. Retrieve a copy of the request business object from the application.

   Recursively retrieve the data for the entire entity from the application using the primary keys from the request business object:

   - For a flat business object, retrieve the single application entity.
   - For a hierarchical business object, use the Retrieve operation to descend into the application business object, expanding all paths in the business object hierarchy.
3. Place the retrieved data in the response business object. This response business object is now a representation of the current state of the entity in the application.

   The Update operation can now compare the two hierarchical business objects and update the application entity appropriately.
4. Update the simple attributes in the application entity to correspond to the top-level source business object.
5. Compare the response business object (created in step 2) with the request business object. Perform this comparison down to the lowest level of the business object hierarchy.

   Recursively update the children of the top-level business object following these rules:

   - If a child business object is present in *both* the response business object and the request business object, recursively update the child by performing the Update operation.
   - If a child business object is present in the request business object but *not* in the response business object, recursively create the child by performing the Create operation.
   - If a child business object is *not* present in the request business object but is present in the response business object, recursively delete the child using either the Delete operation (physical) or a logical delete, depending on the functionality of the connector and the application. For more information on logical deletes, see "Implications of business objects representing logical Delete events" on page 95.

   Note: Only the existence or non-existence of the child objects are compared, *not* the attributes of the child business objects.

If the connector's application supports logical delete, the connector recursively retrieves the complete business object hierarchy; then the Update operation sets status attributes and recursively updates the status of the children.

Note: The Update operation should fail if an application entity does *not* exist for any foreign key (Foreign Key is set to true) referenced in the request business object. The connector should verify that the foreign key is a valid key (it references an existing application entity). If the foreign key is invalid,

the Update operation should return BON_FAIL. A foreign key is assumed to be present in the application, and the connector should *never* try to create an application object marked as a foreign key.

Figure 32 shows a set of associated application entities that represent a customer in the application database. The entities contain customer, address, phone, and customer profile data. Note that the sample customer, Acme Construction, has no phone number in the database.

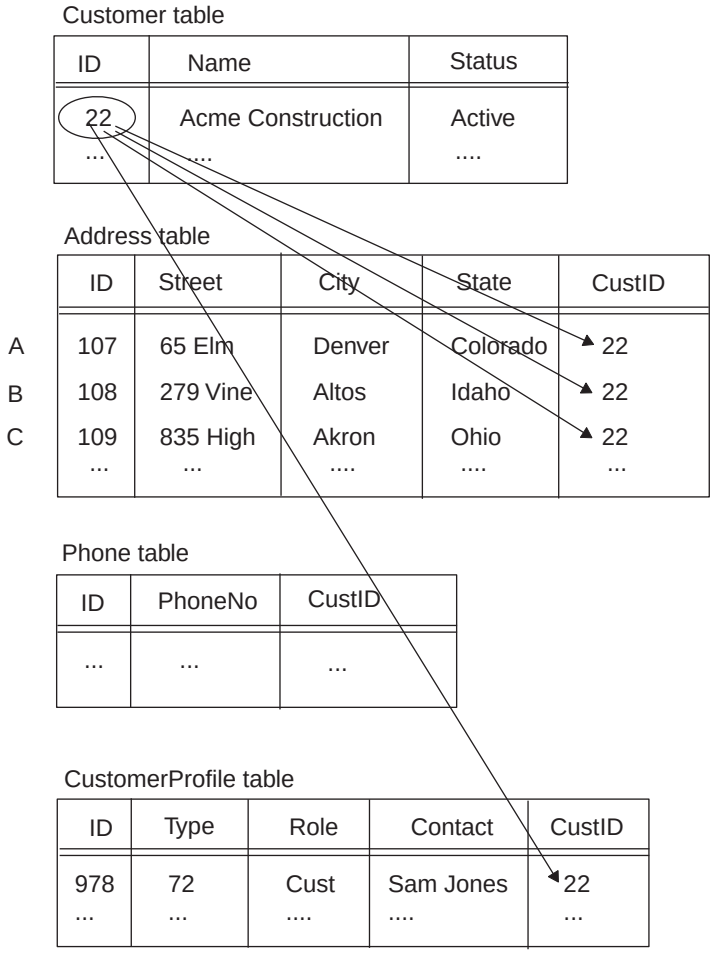

Figure 32. Customer entities before Update request

Assume that an integration broker sends an update request that consists of the request business object as shown in Figure 33..

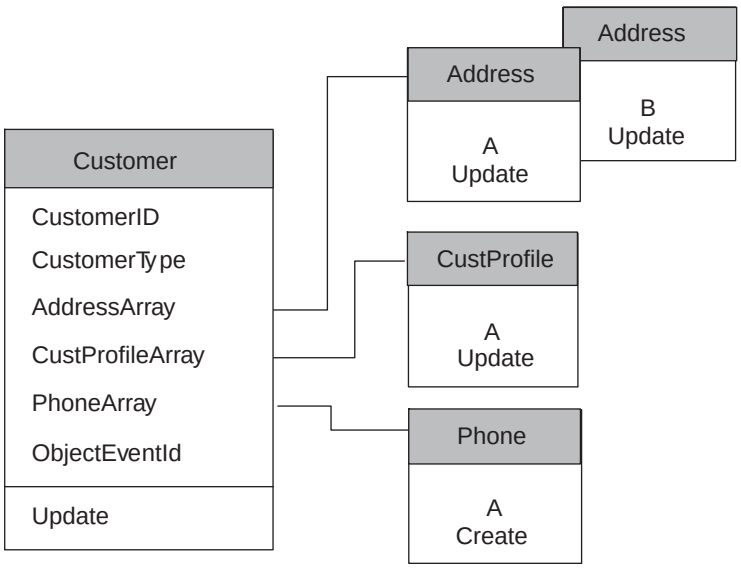

Figure 33. Customer request business object for an Update

This request business object indicates that the Acme Construction customer has undergone the changes listed in Table 31..

Table 31. Updates to Acme Construction in the Request business object

Update made to Acme Construction	Representation in request business object
Acquired a new phone number	The child business object for the PhoneArray attribute (Phone object A) has a Create verb.
Moved to new offices in Denver and Altos	Two child business objects (Address objects A and B) exist in the AddressArray attribute, each with an Update verb.
Closed the office in Akron	No child business object exists in the AddressArray attribute for the Akron address.
Changed the name of the contact person	The child business object for the CustProfileArray attribute (CustProfile object A) has an Update verb.

Your connector's task is to keep the application database for this destination application synchronized with the source application. Therefore, to respond to this request, the connector would need to perform the following tasks as part of its Update operation:

- Update any columns in `Customer` table that have updated values in the corresponding simple attributes of the Customer business object.
- Update the rows in the `Address` table that correspond to Address objects A and B. Update the columns in each of these rows with any new values from the corresponding simple attributes in the appropriate Address object. In this case, the `Street` column has changed for the Denver and Altos offices.
- Delete the row in the `Address` table that corresponds to the Akron address.
- Update the `Contact` column of the `CustomerProfile` table to the value of the corresponding simple attribute in the CustProfile object A business object.
- Create a row in the `Phone` table with column values from the simple attributes of the Phone object A business object. Make sure that the `CustID` column of this new row is created with the foreign-key value that identifies the appropriate `Customer` row (22).

Figure 34 shows the set of associated application entities that represent a customer after the Update operation has completed.

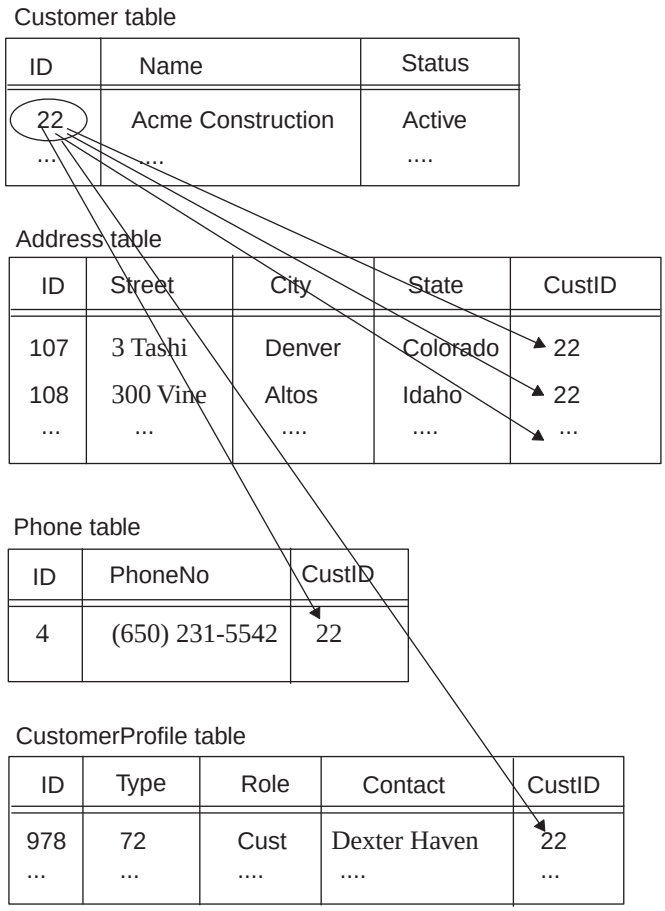

Figure 34. Customer entities after Update request

Implications of business objects representing logical Delete events

If your application supports physical delete, but an integration broker sends requests from a source application that supports only logical delete, you might need to handle a business object that represents a logical delete request. Connectors for applications that perform logical delete operations, where an entity is marked as deleted by updating a status value, should handle logical deletes in the Update method. A system view of this implementation is as follows:

- Events that represent the deletion of data in the source application should be sent as application-specific business objects with the Delete verb. Similarly, maps on the source application side should set the verb of generic business objects to Delete.
- On the destination side, maps for connectors supporting logical delete applications can transform Delete verbs in generic business objects to Update verbs in application-specific business objects. Business object attributes representing entity status values can be set to the inactive status.

In this way, a connector representing a logical delete application receives an application-specific business object with an Update verb and the status value marked appropriately.

For example, assume that a source application entity has been updated to look like the business object representation in Figure 35.. Components in the source application entity have been updated, created, and deleted.

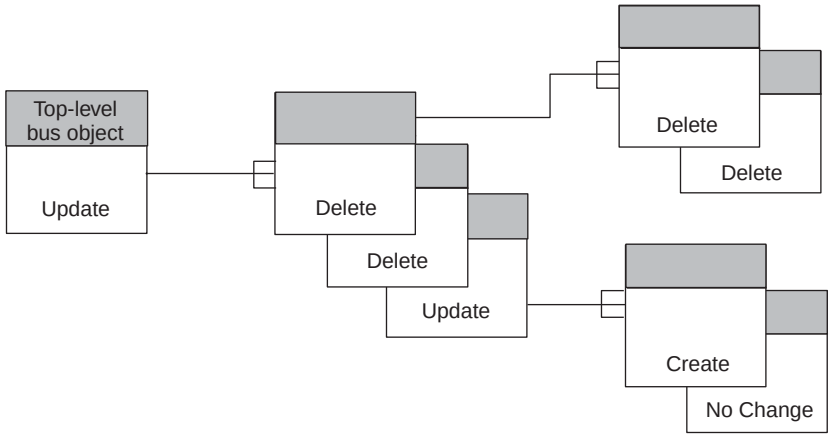

Figure 35. Updated entity in the source application

If the source application connector has implemented event notification as recommended in Chapter 5, "Event notification," on page 109, deleted child business objects are not present in the business object hierarchy, and the business object simply contains the updated and new child business objects.

An example of a business object representing an Update request might look like Figure 36.. In this figure, the parent object is set to update, and all entities that have been deleted are no longer present in the business object hierarchy.

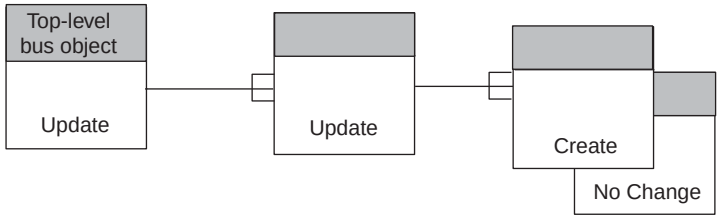

Figure 36. Update request business object from a physical-delete connector

In this case, the connector compares the source and destination business objects and deletes the entities that are not present in the source business object.

However, if the source application supports logical delete, the source connector might send a business object with deletes tagged as updates and status attribute values set to an inactive value. This business object might look like Figure 37,, where updates that are delete operations are identified by "[D]".

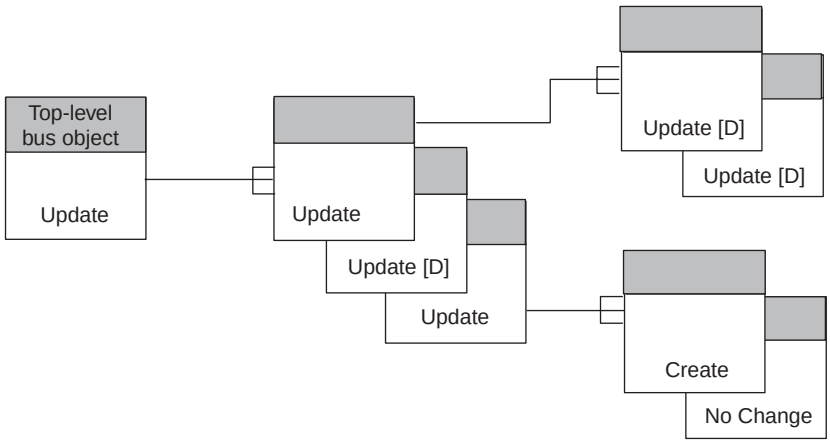

Figure 37. Update request business object from a logical-delete connector

There are several ways to handle a source business object that represents a logical delete request:

- Implement mapping to examine the status of child business objects. If the status of a particular child business object is inactive, the business object can be removed in mapping.
- Implement the Update operation to determine whether an update operation is actually a delete operation. In a logical delete source application, an entity may be marked as active or inactive by a status value. In the source's application-specific business objects, the status value is usually an attribute. Although entities in an application that supports physical delete might not include status information, you can extend your application-specific business objects to include status information.
- Extend a business object by adding an additional status attribute or by overloading an existing attribute with a status value. When the Update operation receives a request, it can check the status attribute. If it is set to the inactive value, the operation is really a delete. The Update operation can then set the business object verb to Delete and call the Delete operation to handle deleted child business objects.

Outcome status for Update verb processing

The Update operation should return one of the outcome-status values shown in Table 32.

Table 32. Possible outcome status for C++ Update verb processing

Update condition	**C++ outcome status**
If the application entity exists, the Update operation: • modifies the data in the application entity • returns a "Success" outcome status	`BON_SUCCESS`
If a row or entity does *not* exist, the Update operation: • creates the application entity • returns the "Value Changed" outcome status to indicate that the connector has changed the business object	`BON_VALCHANGE`

Table 32. Possible outcome status for C++ Update verb processing (continued)

Update condition	C++ outcome status
If the Update operation is unable to create the application entity, it: • fills a return-status descriptor with information about the cause of the update error • returns a "Fail" outcome status	BON_FAIL
If any object identified as a foreign key is missing from the application, the Update operation: • fills a return-status descriptor with information about the cause of the update error • returns a "Fail" outcome status	BON_FAIL

Note: When the connector framework receives the BON_VALCHANGE outcome status, it includes a business object in its response to InterChange Server. For more information, see "Sending the verb-processing response" on page 168.

Handling the Delete verb

For a delete, an application might support either of the implementations shown in Table 33..

Table 33. Delete Implementations

Delete implementation	Description	Verb-processing support
Physical delete	Physically removes the specified application entity.	Delete operation
Logical delete	Does not actually remove the entity; instead, it marks it with a special "deleted" status.	Update operation

Note: If the application does not allow *any* type of delete operation, the connector can return a "Fail" outcome status.

The Delete operation, discussed in this section, performs a true physical deletion of data in the application. Connectors for applications that perform logical delete operations should handle logical deletes in the Update operation. For more information, see "Implications of business objects representing logical Delete events" on page 95.

When the business object handler obtains a Delete verb from the request business object, it must ensure that a physical delete is performed; that is, the application deletes the application entity whose type is indicated by the business object definition, as follows:

- For a flat business object, the Delete verb indicates that the specified entity must be deleted.
- For a hierarchical business object, the Delete verb indicates that the top-level business object must be deleted. Depending on the application policies, the it might delete associated entities representing child business objects.

Note: For a table-based application, the entire application entity must be deleted from the application database, usually deleting a row in one or more database tables.

This section provides the following information to help process a Delete verb:

- "Standard processing for a Delete verb"
- "Outcome status for Delete verb processing"

Note: You can modularize your business object handler so that each supported verb is handled in a separate C++ method. If you follow this structure, a Delete method handles processing for the Delete verb.

Standard processing for a Delete verb

The following steps outline the standard processing for a Delete verb:

1. Perform a recursive retrieve on the request business object to get all data in the application that is associated with the top-level business object.
2. Perform a recursive delete on the entities represented by the request business object, starting from the lowest level entities and ascending to the top-level entity.

Note: Delete operations might be limited by application functionality. For example, cascading deletes might not always be the desired operation. If you are using an application API, it might automatically complete the delete operation appropriately. If you are *not* using an application API, you might need to determine whether the connector should delete child entities in the application. If a child entity is referenced by other entities, it might not be appropriate to delete it.

Outcome status for Delete verb processing

The Delete operation should return one of the outcome-status values shown in Table 34.

Table 34. Possible outcome status for C++ Delete verb processing

Delete condition	C++ outcome status
InterChange Server only: In most cases, the connector returns a "Value Changed" outcome status to enable the system to clean up the relationship tables after a delete operation.	BON_VALCHANGE
All integration brokers: If the Delete operation is unsuccessful, it: • fills a return-status descriptor with additional information about the cause of the delete error • returns a "Fail" outcome status	BON_FAIL

Note: When the connector framework receives the BON_VALCHANGE outcome status, it includes a business object in its response to InterChange Server. For more information, see "Sending the verb-processing response" on page 168.

Handling the Exists verb

When the business object handler obtains an Exists verb from the request business object, it must determine whether an application entity, whose type is indicated by the business object definition, exists. This operation enables an integration broker to verify that an entity exists before the integration broker performs an operation on the entity. As an example, assume that a customer site wants to synchronize Order, Customer, and Item entities in the source and destination applications. Before synchronizing an order, the user wants to ensure that the customer entity referenced by the Order business object already exists in the destination application

database. In addition, the user wants to ensure that each Item entity referenced by the OrderLineItem child business objects also exists in the destination application.

Note: For a table-based application, the Exists method checks for the existence of an entity in an application database, usually checking for a row in a database table.

The user can configure the integration broker to call the connector with a Customer business object that has the Exists verb and the primary keys set. In this way, the integration broker can verify that the customer already exists in the application. Similarly, the user can configure the integration broker to call the connector with referenced Item business objects that have the Exists verb and primary keys set. The user might decide to halt the synchronization of the Order if the verification of the existence of the application entities fails.

This section provides the following information to help implement an Exists verb:

- "Standard processing for an Exists verb"
- "Outcome status for Exists verb processing"

Note: You can modularize your business object handler so that each supported verb is handled in a separate C++ method. If you follow this structure, an Exists method handles processing for the Exists verb.

Standard processing for an Exists verb

The standard behavior of the Exists method is to query the application database for the existence of a top-level business object.

Outcome status for Exists verb processing

The Exists operation should return one of the outcome-status values shown in Table 35.

Table 35. Possible outcome status for C++ Exists verb processing

Exists condition	C++ outcome status
If the application entity exists, the Exists operation returns "Success".	BON_SUCCESS
If the Exists operation is unsuccessful in retrieving the top-level object, it: • fills a return-status descriptor with additional information about the cause of the "exist" error • returns a "Fail" outcome status	BON_FAIL

Processing business objects

A business object handler's role is to deconstruct a request business object, process the request, and perform the requested operation in the application. To do this, a business object handler extracts verb and attribute information from the request business object and generates an API call, SQL statement, or other type of application interaction to perform the operation.

Basic business object processing involves extracting metadata from the business object's application-specific information (if it exists) and accessing the attribute values. The actions to take on the attribute value depend on whether the business

object is flat or hierarchical. This section provides an overview on how a business object handler can process the following kinds of business objects:

- "Processing flat business objects"
- "Processing hierarchical business objects" on page 103

Processing flat business objects

This section provides the following information on how to process flat business objects:

- "Representing a flat business object"
- "Accessing simple attributes" on page 102

Representing a flat business object

If a business object does *not* contain any other business objects (called child business objects), it is called a *flat business object*. All the attributes in a flat business object are *simple attribute*; that is, each attribute contains an actual value, not a reference to another business object.

Suppose you have to perform verb processing on an example business object named `Customer`. This business object represents a single database table in a sample table-based application. The database table is named `customer`, and it contains customer data. Figure 38 shows the `Customer` business object definition and the corresponding `customer` table in the application.

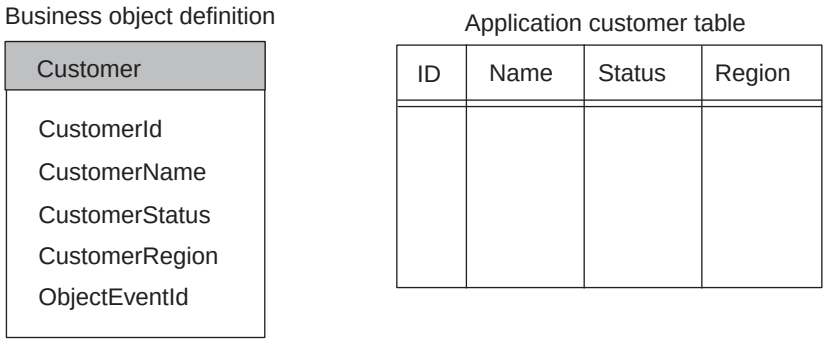

Figure 38. A Flat business object and corresponding application table

As Figure 38 shows, the example `Customer` business object has four simple attributes: `CustomerId`, `CustomerName`, `CustomerStatus`, and `CustomerRegion`. These attributes correspond to columns in the `customer` table. The business object also includes the required `ObjectEventId` attribute.

Note: The `ObjectEventId` attribute is used by the IBM WebSphere business integration system and does *not* correspond to a column in an application table. This attribute is automatically added to business objects by Business Object Designer.

Figure 39 shows an expanded business object definition and an instance of the business object. The business object definition contains the business object name, and the attribute name, properties, and application-specific information. The business object instance contains only the business object name, the active verb, and the attribute names and values.

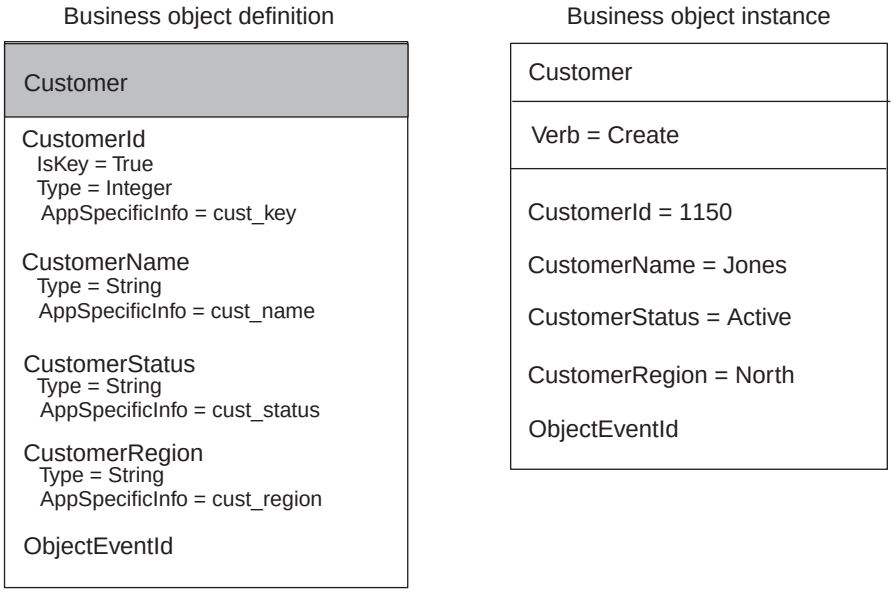

Figure 39. A flat business object with application-specific information

Accessing simple attributes

After the verb operation has accessed information it needs within the business object definition, it often needs to access information about attributes. Attribute properties include the cardinality, key or foreign key designation, and maximum length. For example, the example Create method needs to obtain the attribute's application-specific information. A connector business object handler typically uses the attribute properties to decide how to process the attribute value.

Figure 40 illustrates business object attribute properties of the `CustomerId` attribute from the business object in Figure 39..

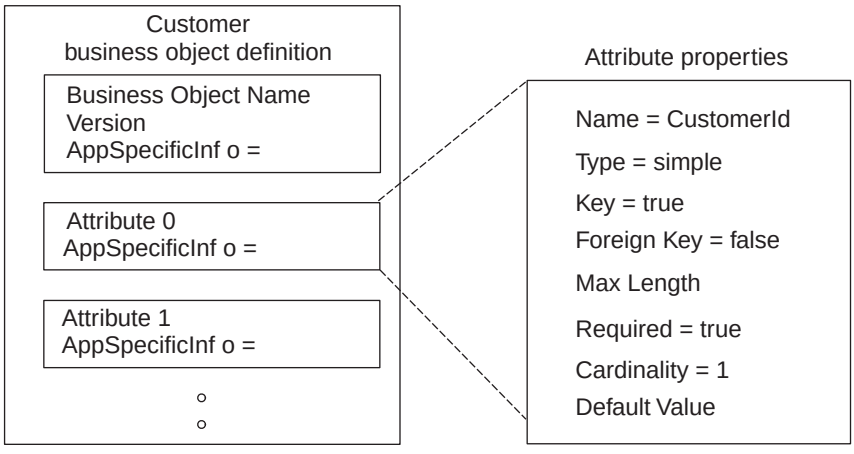

Figure 40. Business object attribute properties

Each attribute has a zero-based integer index (ordinal position) within the business object definition. For example, as Figure 40 shows, the `CustomerId` attribute would be accessed with an ordinal position of zero (`0`), the `CustomerName` attribute with an

ordinal position of one (1), and so on. The C++ connector library provides access to an attribute through its name or ordinal position.

For the business object handler that handles the flat `Customer` business object, deconstructing a business object includes the following steps:

1. Extract the table and column names from the application-specific information in the business object definition.
2. Extract the values of the attributes from the business object instance.

As Figure 39 shows, the `Customer` business object definition is designed for a metadata-driven connector. Its business object definition includes application-specific information that the verb operation uses to locate the application entity upon which to operation. The application-specific information is designed as shown in Table 36..

Table 36. Application-specific information for a table-based application

Application-specific information	Purpose
Business object definition	The name of application database table associated with this business object
Attribute	The name of the application table's column associated with this attribute

Note: Application-specific information is also used to store information on foreign keys and other kinds of relationships between entities in the application database. A metadata-driven connector can use this information to build a SQL statement or an application API call.

Processing hierarchical business objects

Business objects are hierarchical: parent business objects can contain child business objects, which can in turn contain child business objects, and so on. A hierarchical business object is composed of a *top-level business object*, which is the business object at the very top of the hierarchy, and *child business objects*, which are all business objects under the top-level business object. A child business object is contained in a parent object as an attribute.

This section provides the following information on how to process hierarchical business objects:

- "Representing Top-Level and Child Business Objects"
- "Accessing child business objects" on page 105

Representing Top-Level and Child Business Objects

If a top-level business object has child business objects, it is the parent of its children. Similarly, if a child business object has children, it is the parent of its children. The parent/child terminology describes the relationships between business objects, and it may also be used to describe the relationship between application entities.

There are two types of containment relationships between parent and child business objects:

- Cardinality 1 containment—the attribute contains a single child business object.
- Cardinality n containment—the attribute contains several child business objects in a structure called a *business object array*.

Figure 41 shows a typical hierarchical business object. The top-level business object has both cardinality 1 and cardinality n relationships with child business objects.

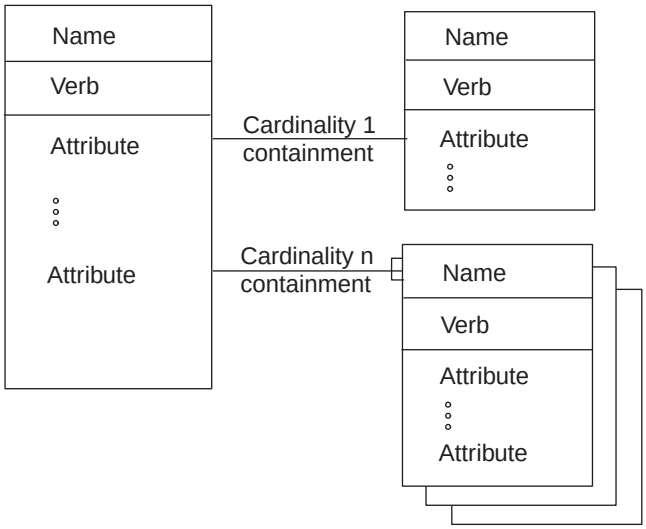

Figure 41. Hierarchical business object

In a typical table-based application, relationships between entities are represented by primary keys and foreign keys in the database, where the parent entity contains the primary keys and the child entity contains the foreign keys. An hierarchical business object can be organized in a similar way:

- In a cardinality 1 type (single cardinality) of relationship, each parent business object relates to a single child business object.

  The child business object typically contains one or more foreign keys whose values are the same as the corresponding primary keys in the parent business object. Although applications might structure the relationships between entities in different ways, a single cardinality relationship for an application that uses foreign keys might be represented as shown in Figure 42..
- In a cardinality n type (multiple cardinality) relationship, each parent business object can relate to zero or more child business objects in an array of child business objects.

  Each child business object within the array contains foreign key attributes whose values are the same as the corresponding values in the primary key attributes of the parent business object. A multiple cardinality relationship might be represented as shown in Figure 43..

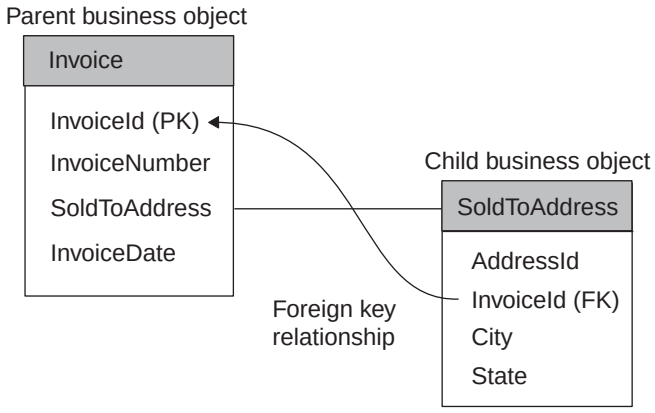

Figure 42. Business objects with single cardinality

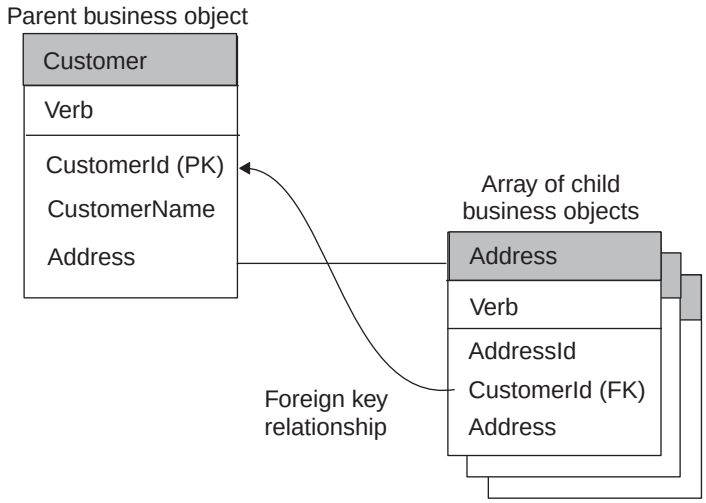

Figure 43. Business objects with multiple cardinality

Note: In Figure 42 and Figure 43,, the string "PK" appears next to an attribute that serves as a primary key in the business object. The string"FK" appears next to an attribute that serves as a foreign key.

Accessing child business objects

As part of its verb processing, the `doVerbFor()` method needs to handle any hierarchical business objects. To process a hierarchical business object, the `doVerbFor()` method takes the same basic steps as it does to process a flat business object: it obtains any application-specific information and then accesses the attribute. However, if the attribute contains a child business object, `doVerbFor()` must take the following steps to access the child business object:

1. Determine whether the attribute type is type `OBJECT` by calling the `isObjectType()` method.

   The `OBJECT` type indicates that the attribute is a complex attribute; that is, it contains a business object rather than a simple value. The `OBJECT` attribute-type constant is defined in the BOAttrType class. The `isObjectType()` method returns True if an attribute is complex; that is, if it contains a business object.
2. When the `doVerbFor()` method finds an attribute contains a business object, it checks the cardinality of the attribute using `isMultipleCard()`.

If the attribute has single cardinality (cardinality 1), the method can perform the requested operation on the child. One way to perform an operation on a child business object is to recursively call `doVerbFor()` or a verb method on the child object. However, such a recursive call assumes that the child business object is set as follows:

- If the verb on a child business object is set, the method should perform the specified operation.
- If the verb on the child business object is *not* set, the verb method should set the verb in the child business object to the verb in the top-level business object before calling another method on the child.

If an attribute has multiple cardinality (cardinality n), the attribute contains an array of child business objects. In this case, the connector must access the contents of the array before it can process individual child business objects. From the array, the `doVerbFor()` method can access individual business objects:

- To access individual business objects, the method can get the number of child business objects in the array with the `getObjectCount()` method and then iterate through the objects.
- To get an individual child business object, the method can obtain the business object at one element of the array.

Once the `doVerbFor()` method has access to a child business object, it can recursively process the child as needed.

Note: A connector should never create arrays for child business objects. An array is always associated with a business object definition when cardinality is n.

When a connector a request business object, the business object includes all its arrays even though some or all of the arrays might be empty. If an array contains no child business objects, it is an array of size 0.

You might want to modularize your verb operation so that the main verb method calls a submethod to process child objects. For a business object such as the one shown in Figure 44,, a Create method might first create the application entity for the parent Customer business object, and then call the submethod to traverse the parent business object to find attributes referring to contained business objects.

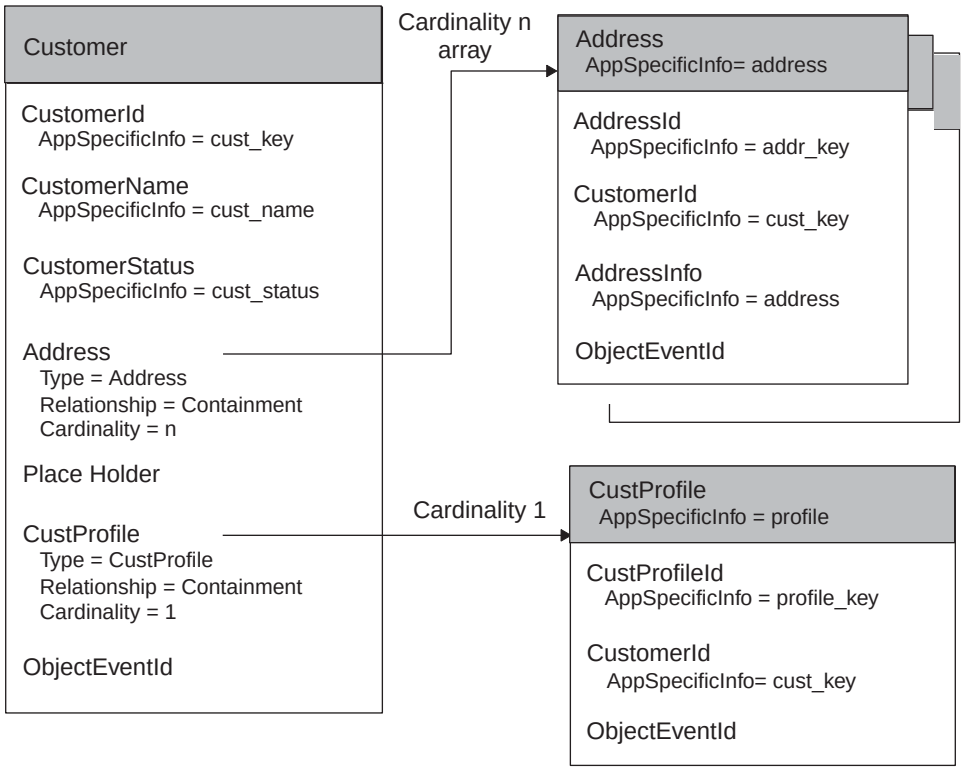

Figure 44. Example of a hierarchical business object definition

When the submethod finds an attribute that is an `OBJECT` type, it can process the attribute as needed. For example, the submethod processes the `Address` attribute by retrieving each child business object in the `Address` array and recursively calling `doCreate()`. One by one, the main method creates each `address` entity in the database until all `Address` children in the array are processed. Finally, the submethod processes the single cardinality `CustProfile` business object.

For more information about how to access a child business object, see "Accessing child business objects" on page 175.

Indicating the connector response

Before the `doVerbFor()` method exits, it must prepare the response it sends back to the connector framework. This response indicates the success (or lack thereof) of the verb processing. The connector framework, which has invoked `doVerbFor()`, uses this information to determine its next action and to build the response it returns to the integration broker.

The `doVerbFor()` method can provide the response information in Table 37 to the connector framework.

Table 37. Response information from the doVerbFor() method

Response information	How the response is returned
Outcome status	Integer return code of `doVerbFor()`

Table 37. Response information from the doVerbFor() method (continued)

Response information	How the response is returned
Return-status descriptor	Return-status descriptor that was passed in as an argument—Connector framework passes in an empty return-status descriptor as an argument to `doVerbFor()`. The method can update this descriptor with a message and status value to provide informational, warning, or error status.
Response business object	Request business object that was passed in as an argument—Connector framework passes in the request business object as an argument to `doVerbFor()`. The method can update this request business object with attribute values to provide a response business object.

For information on how to send this response information for a C++ connector, see "Sending the verb-processing response" on page 168.

Handling loss of connection to the application

Each time the connector framework calls the connector application-specific component, the application-specific code validates that the connection with the application is still open. For a business object handler, this check should be done in either the `doVerbFor()` method or in each verb method.

If the connection has been lost, the `doVerbFor()` method should log a fatal error message so that email notification is triggered if the `LogAtInterchangeEnd` connector configuration property is set to True. The method should also return a BON_APPRESPONSETIMEOUT outcome status to inform the connector controller that the application is not responding. When this occurs, the process in which the connector runs is stopped. A system administrator must fix the problem with the application and restart the connector to continue processing of business object requests.

For more information, see "Verifying the connection before processing the verb" on page 152.

Chapter 5. Event notification

This chapter presents information on how to provide event notification in a connector. *Event notification* implements a mechanism to interact with an application to detect changes made to application business entities. This chapter provides the following information about how to implement an event-notification mechanism:

- "Overview of an event-notification mechanism"
- "Implementing an event store for the application" on page 110
- "Implementing event detection" on page 115
- "Implementing event retrieval" on page 120
- "Implementing the poll method" on page 122
- "Special considerations for event processing" on page 126

Note: For an introduction to event notification, see "Event notification" on page 22..

Overview of an event-notification mechanism

An *event-notification mechanism* enables a connector to determine when an entity within an application changes. Implementation of an event-notification mechanism is a three-stage process, as Table 38 shows.

Table 38. Stages of an event-notification mechanism

Stage of event-notification mechanism	For more information
Create an *event store* that the application uses to hold notifications of events that have changed application business entities.	"Implementing an event store for the application" on page 110
Implement an **event detection** mechanism within the application. Event detection notices a change in an application entity and writes an **event record** containing information about the change to an event store in the application.	"Implementing event detection" on page 115
Implement an **event retrieval** mechanism (such as a polling mechanism) within the connector to retrieve events from the event store and take the appropriate action to notify other applications.	"Implementing an event store for the application" on page 110

Note: For design considerations for an event-notification mechanism, see "Event notification" on page 22..

In many cases, an application must be configured or modified before the connector can use the event-notification mechanism. Typically, this application configuration occurs as part of the installation of the connector's application-specific component. Modifications to the application might include setting up a user account in the application, creating an event store and event table in the application database, inserting stored procedures in the database, or setting up an inbox. If the application generates event records, it might be necessary to configure the text of the event records.

The connector might also need to be configured to use the event-notification mechanism. For example, a system administrator might need to set connector-specific configuration properties to the names of the event store and event table.

Implementing an event store for the application

An *event store* is a persistent cache in the application where event records are saved until the connector can process them. The event store might be a database table, application event queue, email inbox, or any type of persistent store. If the connector is not operational, a persistent event store enables the application to detect and save event records until the connector becomes operational.

This section provides the following information about an event store:

- "Standard contents of an event record"
- "Possible implementations of an event store" on page 112

Standard contents of an event record

Event records must encapsulate everything a connector needs to process an event. Each event record should include enough information that the connector poll method can retrieve the event data and build a business object that represents the event.

Note: Although different event retrieval mechanisms might exist, this section describes event records in the context of the most common mechanism, polling.

If the application provides an event detection mechanism that writes event records to an event store, the event record should provide discrete detail on the object and verb. If the application does not provide sufficient detail, it might be possible to configure it to provide this level of detail.

Table 39 lists the standard elements for event records. The sections that follow include more information on certain fields.

Table 39. Standard elements of an event record

Element	Description	For more information
Event identifier (ID)	A unique identifier for the event.	"Event identifier" on page 111
Business object name	The name of the business object definition as it appears in the repository.	"Business object name" on page 111
Verb	The name of the verb, such as Create, Update, or Delete.	"Event verb" on page 111
Object key	The primary key for the application entity.	"Object key" on page 111
Priority	The priority of the event in the range 0 - n, where 0 is the highest priority.	"Processing events by event priority" on page 125
Timestamp	The time at which the application generated the event.	None.
Status	The status of the event. This is used for archiving events.	"Event status" on page 112
Description	A text string describing the event.	None
Connector identifier (ID)	An identifier for the connector that will process the event.	"Event distribution" on page 126

Note: A minimal set of information in an event record includes the event ID, business object name, verb, and object key. You may also want to set a priority for an event so that if large numbers of events are queued in the event store, the connector can select events in order of priority.

Business object name

You can use the name of the business object definition to check for event subscriptions. Note that the event record should specify the *exact* name of the business object definition, such as `SAP_Customer` rather than `Customer`.

Event verb

The verb represents the kind of event that occurred in the application, such as Create, Update, or Delete. You can use the verb to check for event subscriptions.

Note: Events that represent deletion of application data should generate event records with the Delete verb. This is true even for logical delete operations, where the delete is an update of a status value to inactive. For more information, see "Processing Delete events" on page 126.

The verb that the connector sets in the business object should be same verb that was specified in the event record.

Object key

The entity's object key enables the connector to retrieve the full set of entity data if the object has subscribing events.

Note: The only data from the application entity that event records should include are the business object name, active verb, and object key. Storing additional entity data in the event store requires memory and processing time that might be unneeded if no subscriptions exist for the event.

The object key column must use name/value pairs to set data in the event record. For example, if `ContractId` is the name of an attribute in the business object, the object key field in the event record would be:

```
ContractId=45381
```

Depending on the application, the object key may be a concatenation of several fields. Therefore, the connector should support multiple name/value pairs that are separated by a delimiter, for example ContractId=45381:HeaderId=321. The delimiter should be configurable as set by the `PollAttributeDelimiter` connector configuration property. The default value for the delimiter is a colon (:).

Event identifier

Each event must have a unique identifier. This identifier can be an number generated by the application or a number generated by a scheme that your connector uses. As an example of an event ID numbering scheme, the event may generate a sequential identifier, such as 00123, to which the connector adds its name. The resulting object event ID is `ConnectorName`_00123. Another technique might be to generate a timestamp, resulting in an identifier such as *`ConnectorName`*_06139833001001.

Your connector can optionally store the event ID in the `ObjectEventId` attribute in a business object. The `ObjectEventId` attribute is a unique value that identifies each event in the IBM WebSphere business integration system. Because this attribute is required, the connector framework generates a value for it if the application-specific connector does not provide a value. If no values for

ObjectEventIds are provided for hierarchical business objects, the connector framework generates values for the parent business object and for each child. If the connector generates ObjectEventId values for hierarchical business objects, each value must be unique across all business objects in the hierarchy regardless of level.

Event status

IBM recommends that a C++ connector use the event-status values that Table 40 lists. To improve readability of your C++ code, you might want to define constants for each of these event-status values.

Table 40. Suggested event-status values for a C++ connector

Event-status value	Description
0	Ready for poll
1	Sent to the integration broker
2	No subscriptions for event
3	Event is in progress
-1	Error in processing the event. A description of the error can be appended to the event description in the event record.
-2	Error in sending the event to the integration broker. A description of the error can be appended to the event description in the event record.

Possible implementations of an event store

The application might use any of the following as the event store:

- "Event inbox"
- "Event table" on page 113
- "Email" on page 114
- "Flat files" on page 115

Note: Some applications might provide multiple ways of keeping track of changes to application entities. For example, an application might provide workflow for some database tables and user exits for other tables. If this is the case, you may have to piece together an event notification mechanism that handles events in one way for some business objects and another way for other business objects.

Event inbox

Some applications have a built-in inbox mechanism. This inbox mechanism can be used to transfer information about application events to the connector, as follows:

- Event detection—you might need to identify the entities and events that trigger entries in the inbox.
- Event retrieval—the connector's application-specific component can retrieve the entries. If an API is available that provides interfaces to access the inbox, the application-specific component can use this API.

Figure 45 illustrates this interaction.

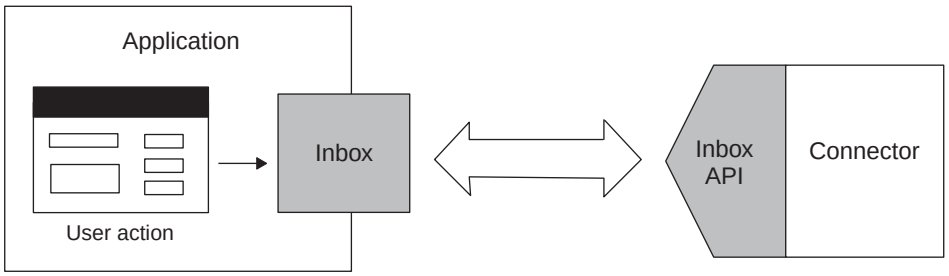

Figure 45. An event inbox as an event store

Event table

An application can use its application database to store event information. It can create a special *event table* in this database to use as the event store for event records. This table is created during the installation of the connector. With an event table as an event store:

- Event detection—when an event of interest to the connector occurs, the application places an event record in the event table.
- Event retrieval—the connector application-specific component polls the event table periodically and processes any events. Applications often provide database (DB) APIs that enable the connector to gain access to the contents of the event table.

Figure 46 illustrates this interaction.

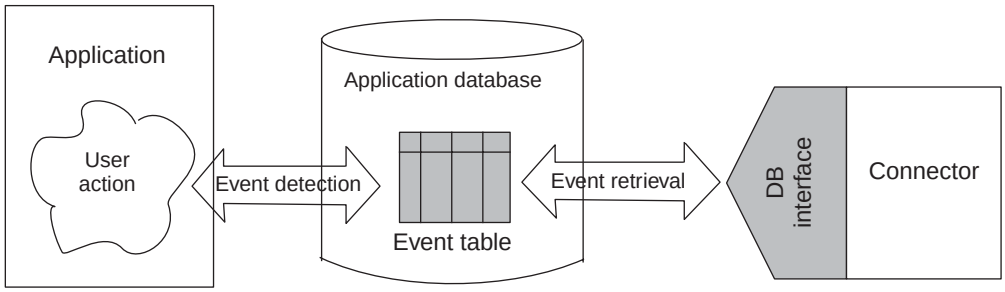

Figure 46. An event table as an event store

Note: Avoid full table scans of existing application tables as a way of determining whether application tables have changed. The recommended approach is to populate an event table with event information and poll the event table.

If your connector supports archiving of events, you can also create an archive table in the application database to hold the archived events. Table 41 shows a recommended schema for event and archive tables. You can extend this schema as needed for your application.

Table 41. Recommended schema for event and archive tables

Column name	Type	Description
`event_id`	Use appropriate type for database	The unique key for the event. System constraints determine format.
`object_name`	Char 80	Complete name of the business object.
`object_verb`	Char 80	Event verb.

Table 41. Recommended schema for event and archive tables (continued)

Column name	Type	Description
`object_key`	Char 80	The primary key of the object.
`event_priority`	Integer	The priority of the event, where 0 is the highest priority.
`event_time`	DateTime	The timestamp for the event (time at which the event occurred).
`event_processed`	DateTime	For the archive table only. The time at which the event was handed to the connector framework.
`event_status`	Integer	For possible status values, see "Event status" on page 112.
`event_description`	Char 255	Event description or error string
`connector_id`	Integer	Id for the connector (if applicable)

Email

You can use an email system as an event store:

- Event detection—the application sends an email message to a mailbox when an application event occurs.
- Event retrieval—the connector's application-specific component checks the mailbox and retrieves the event message.

Figure 47 illustrates this interaction.

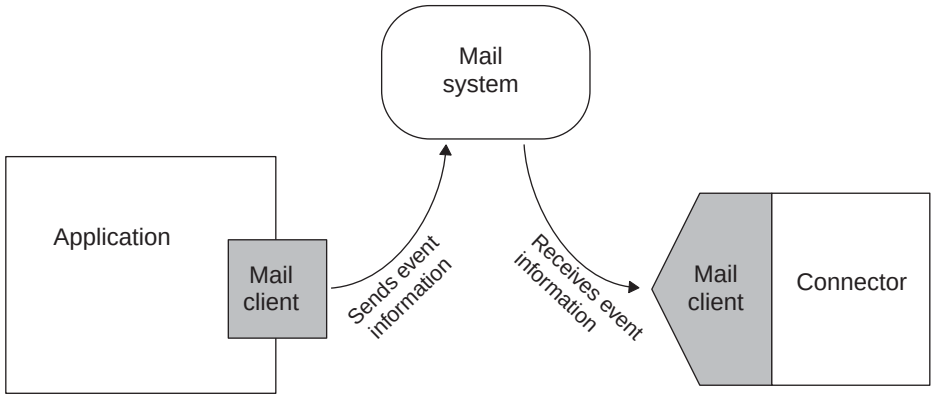

Figure 47. A mailbox as an event store

For an email-based event store, the mailbox used for a connector must be configurable, and the actual name of the inbox used should reflect its usage. The following list specifies the format and recommended names for fields in event messages.

- Message attributes – Email messages usually have certain attributes, such as a creation date and time, and a priority. You may be able to use these attributes in the event notification mechanism. For example, you may be able to use the date and time attributes to represent the date and time at which the event occurred.
- Subject – The subject of an event message might have the following format. In this example, fields are separated by spaces for human-readability, but connectors can use a different field delimiter.

  `object_name object_verb event_id`

  The *event_id* is the unique key for the event. Depending on the application, the *event_id* key may or may not be included in the mail message. The *event_id* can

be derived from a combination of the connector name, business object name, and either the message timestamp or the system time.

- Body – The body of an event message might contain a sequence of key/value pairs separated by delimiters. These key/value pairs are the elements of the object key. For example, if a particular customer and address are uniquely identified by the combination of CustomerId and AddrSeqNum, the body of the mail message might look like this:

  ```
  CustomerId 34225
  AddrSeqNum 2
  ```

  The body of the event message can be a list of attribute names for the business object, and the values that should be inserted into those attributes.

Flat files

If no other event detection mechanism is available, it might be possible to set up an event store using flat files. With this type of event store:

- Event detection—the event detection mechanism in the application writes event records to a file.
- Event retrieval—the connector's application-specific component locates the file and reads the event information.

If the file is not directly accessible by the connector (if, for example, it was generated on a mainframe system), the file must be transferred to a location that the connector can access. One way of transferring files is to use File Transfer Protocol (FTP). This can be done either internally in the connector or using an external tool to copy the file from one location to another. There are other ways to transfer information between files; the approach that you choose depends on your application and connector.

Figure 48 illustrates event detection and retrieval using flat files. In this example, FTP is used to transfer the event information to a location accessible by the connector.

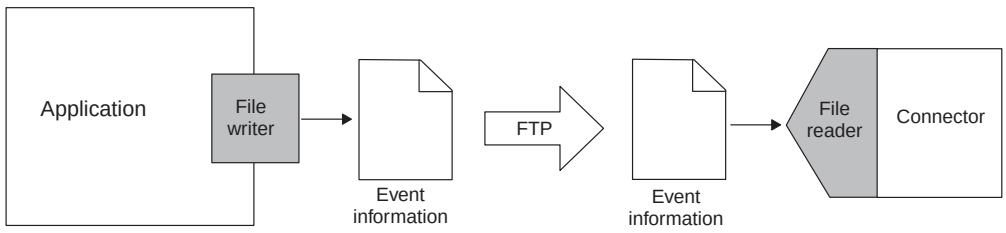

Figure 48. Retrieving event records from flat files

Implementing event detection

For most connectors, the application must be configured to implement the event detection mechanism. A system administrator does this as part of the connector installation. Once the application has been configured, it can detect entity changes and write event records to the event store. The information is then picked up by the connector and processed. In this way, an event notification mechanism is implemented in both the application and the connector.

This section provides the following information about event detection:

- "Event detection mechanisms" on page 116
- "Event detection: standard behavior" on page 119

Event detection mechanisms

Events can be triggered by user actions in the application, by batch processes that add or modify application data, or by database administrator actions. When an event detection mechanism is set up in an application and an application event associated with a business object occurs, the application must detect the event and write it to the event store.

Event detection mechanisms are application dependent. Some applications provide an event detection mechanism for use by clients such as connectors. The event detection mechanism may include an event store and a defined way of inserting information about application changes into the event store. For example, one type of implementation uses an event message box, where the application sends a message every time it processes an event in which the connector is interested. The connector's application-specific component periodically polls the message box for new event messages.

Other applications have no built-in event detection mechanism but have other ways of providing information on changes to application entities. If an application does not provide an event detection mechanism, you must use whatever mechanism is available to extract information on entity changes for the connector. For example, you may be able to implement database triggers, use user exits to call out to a program that writes to an event store, or extract information on application changes from flat files.

Note: Although the way in which events are generated can vary significantly from application to application, certain aspects of an event notification mechanism should be consistent across all types of applications. For example, all types of event detection mechanisms should create event records that have similar contents.

Three common ways in which events are detected and written to an event store are discussed in the following sections:

- "Form events"
- "Workflow" on page 117
- "Database triggers" on page 118

Form events

Some form-based applications provide form events that are executed when a special user action occurs. To set up event detection in this way, you must create a script that executes when a particular type of event occurs. When a user opens a form and performs an action that has an associated script, the script places event records in the event store.

In most cases, form events are integrated in application business processes and therefore support application business logic. However, only application events that are triggered by user actions are detected; if the application database is updated directly in other ways, such as by a batch process, these events are not detected.

Figure 49 shows a form-based event detection mechanism. When a user enters a new customer on the Customer form and clicks OK, a script generates an event record and places it in the event store.

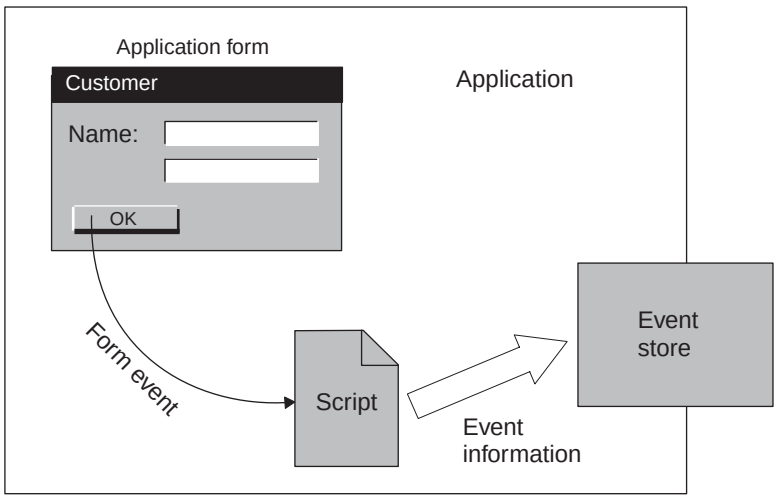

Figure 49. Form-based event detection

Workflow

Some applications use an internal workflow system to keep track of their business processes. You may be able to use the workflow system to generate events for event detection.

For example, you may be able to define a workflow process that inserts an entry in an event store when a particular operation occurs. Alternatively, the event detection mechanism might be able to intercept information from a workflow process and use the information to place an event record in the event store. In designing a workflow-based event detection mechanism, you need to determine at what point in the workflow an event record should be written to the event store and then use the available application mechanism to generate the event record.

Using a workflow system for event detection ensures that event detection is integrated into an application business process. The workflow system can also detect application events that are generated automatically without user involvement.

Figure 50 shows a workflow-based event detection mechanism. When a particular operation occurs, the workflow process is started. The event detection mechanism receives the information about the event and writes a record to the event store. The workflow process continues with other tasks.

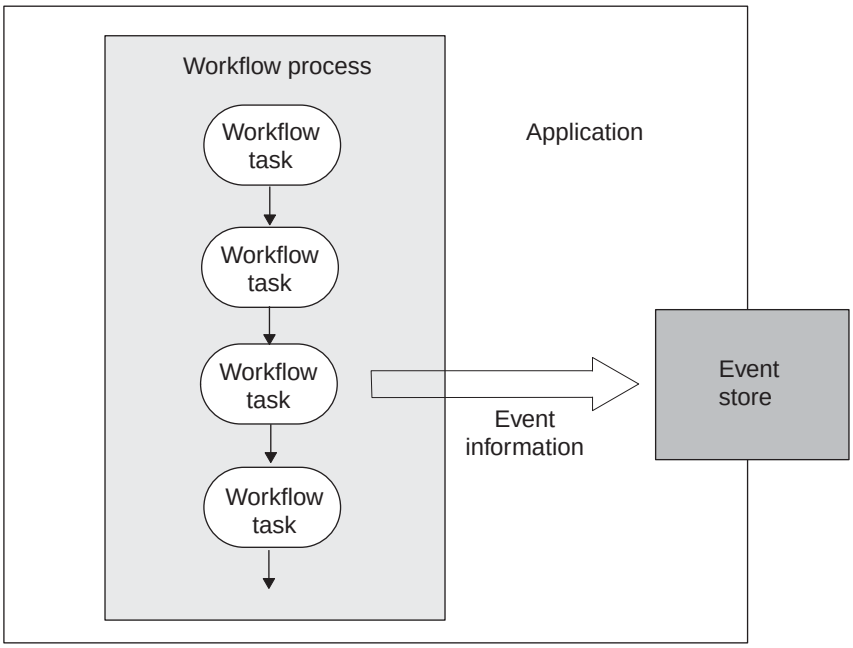

Figure 50. Workflow-based event detection

Database triggers

If the application has no built-in method for detecting events and the database that the application is running on provides database triggers, you may be able to implement row-level triggers to detect changes to application tables. The triggers are inserted in application tables that correspond to business object definitions supported by the connector.

With this mechanism, you also need to set up an event table in the application database to store the event records that the triggers generate. Whenever an application entity is created, updated, or deleted, a trigger inserts a row into the event table. Each row represents one event record, and the event table queues the events for processing by the connector.

Figure 51 shows a user action that updates an application Customer table. When the Customer table is updated, a trigger on the table executes and writes an event record to the event table in the application database.

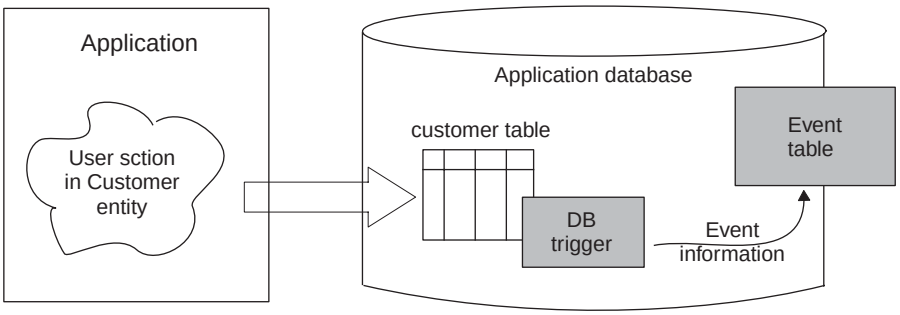

Figure 51. Event detection using database triggers

If you use database triggers, keep the following in mind:

- Make sure that any triggers you provide do not overwrite triggers already in use in the application.
- Make sure that the application is suitable for the use of triggers for event notification. For example, if an application has implemented complex business rules in its database, a simple trigger on a particular table might not accurately reflect the complete application event.
- A drawback to database triggers is that if table schemas change in the application database, you may need to modify the triggers that you have created. If table schemas change frequently and you have set up many database triggers, you may need to spend considerable time maintaining the triggers.

Event detection: standard behavior

An application event detection mechanism should take the following steps:

- Detect an event on an application entity for a business object supported by the connector.
- Create an event record. To create the record, the event detection mechanism should:
  - Set the name of the object to the complete name of the business object in the repository.
  - Set the verb to the action that occurred in the database.
  - Set the object key to the primary key of the application entity.
  - Generate a unique event identifier (ID).
  - Set the event priority.
  - Set the event timestamp.
  - Set the event status to Ready-for-Poll.
- Insert the completed event record into the event store.

  Note: An event detection mechanism can optionally query the event store for existing duplicate events before inserting a record for a new event. For more information, see "Filtering the event store for duplicate event records" on page 119.

Once event records are in the event store, the event store queues events for pickup by the connector's poll method. The event store should be internal to the application. If the application terminates unexpectedly, the event store can be restored to its preceding state when the application is restored, and the connector application-specific code can then pick up queued events.

The event detection mechanism should ensure data integrity between an application event and the event record written to the event store. For example, generation of an event record should *not* take place until *all* required data transactions for the event have completed successfully.

Subsequent sections provide the following information about issues to handle in the event detection mechanism:

- "Filtering the event store for duplicate event records"
- "Future event processing" on page 120

Filtering the event store for duplicate event records

The event detection mechanism can be implemented so that duplicate events are *not* saved in the event store. This behavior can minimize the amount of processing that the integration broker has to perform. As an example, if an application

updates a particular `Address` object several times between connector polls, all the events might be stored in the event store, and the connector will then create business objects for all events and send them to InterChange Server. To prevent this, the event detection mechanism can filter the events such that only a single Update event is stored.

Before storing a new event as a record in the event store, the event detection mechanism can query the event store for existing events that match the new event. The event detection mechanism should *not* generate a record for a new event in these cases:

Case 1	The business object name, verb, key, status, and `ConnectorId` (if applicable) in a new event match those of another unprocessed event in the event store.
Case 2	The business object name, key, and status for a new event match an unprocessed event in the event table; in addition, the verb for the new event is Update, and the verb for the unprocessed event is Create.
Case 3	The business object name, key, and status for a new event match an unprocessed event in the event table; in addition, the verb in the unprocessed event in the event table is Create, and the verb in the new event is Delete. In this case, remove the Create record from the event store.

Note: If event detection is implemented with stored procedures and triggers, the stored procedures can perform the query before inserting records for new events.

Future event processing

The event detection mechanism can be set up to specify a date and time in the future to process an event. To implement this feature, you may need to set up an additional event store for these events. Event records in the future event store should include a date that identifies when they will be processed.

This feature is required for applications with records that include effective dates. As an example, suppose that an existing employee will receive a promotion in a month and that, at that time, he will receive a raise. Because the paperwork for his increased compensation is completed prior to the date of his promotion, the change to his status generates an event with an effective date, which is stored in the future event table.

Implementing event retrieval

For most connectors, the application-specific component of the connector implements the event retrieval mechanism. The connector developer does this as part of the connector design and implementation. This mechanism works in conjunction with the event detection mechanism, which detects entity changes and writes event records to the event store. Event retrieval transfers information about application events from the event store to the connector's application-specific component.

This section provides the following information about event retrieval:

- "Event retrieval mechanisms"
- "Using a polling mechanism" on page 121

Event retrieval mechanisms

Two common mechanisms use to retrieve event records from an event store are:

- Event callback mechanism—connectors can be notified of application events through an event-callback mechanism; however, few applications currently provide event callback APIs for application events.
- Polling mechanism—the most common type of event retrieval mechanism is a polling mechanism.

Using a polling mechanism

In a polling mechanism, the application provides a persistent event store, such as an database table or inbox, where it writes event records when changes to application entities occur. The connector periodically checks, or polls, the event store for changes to entities that correspond to business object definitions that the connector supports. In general, the only information about the business object that is kept in the event store is the type of operation and the key values of the application entity. As the connector processes the event, it retrieves the remainder of the application entity data. After the connector has processed the event, it removes the event record from the event store and places it in an archive store.

To implement a polling mechanism to perform event retrieval, the connector's application-specific component uses a *poll method*, called the `pollForEvents()` method. The poll method checks the event store, retrieves new events, and processes each event before returning.

This section provides the following information about the poll method:

- "Polling interval"
- "Event polling: standard behavior"

Polling interval

The connector framework calls the poll method at a specified *polling interval* as defined by the `PollFrequency` connector configuration property. This property is initialized at connector installation time with Connector Configurator. Typically, the polling interval is about 10 seconds.

Note: If your connector does not need to poll to retrieve event information, polling can be turned off by setting the `PollFrequency` property to zero (0).

Therefore, the connector framework calls the `pollForEvents`() method in either of the following conditions:

- The `PollFrequency` is set to a value greater than zero.
- The connector startup script specifies a value for the `-fPollFreq` option.

Event polling: standard behavior

Figure 52 illustrates the basic behavior of a poll method:

1. The connector framework calls the application-specific component's `pollForEvents`() method to begin polling.
2. The `pollForEvents`() method checks the event store in the application for new events and retrieves the events.
3. The poll method then queries the connector framework to determine whether an event has subscribers.
4. If an event has subscribers, the poll method retrieves the complete set of data for the business object from the application.
5. The poll method sends the business object to the connector framework, which routes it to its destination (such as InterChange Server).

Each time the poll method is called, it checks for and retrieves new events, determines whether the event has subscribers, retrieves application data for events with subscribers, and sends business objects to InterChange Server.

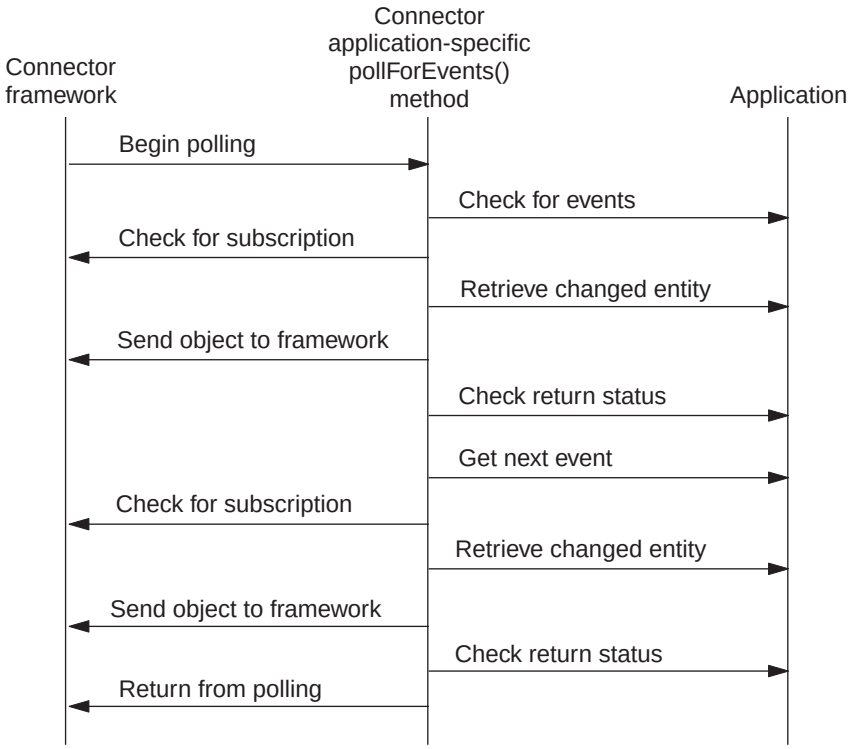

Figure 52. Basic behavior of pollForEvents() method

For information on how to implement the `pollForEvents()` method, see "Implementing the poll method" on page 122.

Implementing the poll method

Regardless of whether the application provides is an event store in a table, inbox, or other location, the connector must poll periodically to retrieve event information. The connector's poll method, `pollForEvents()`, polls the event store, retrieves event records, and processes events. To process an event, the poll method determines whether the event has subscribers, creates a new business object containing application data that encapsulates the event, and sends the business object to the connector framework.

Note: If your connector will be implementing request processing but *not* event notification, you might not need to fully implement `pollForEvents()`. However, since the poll method is defined as a pure virtual method in the C++ connector library, you must at least implement a stub for this method.

This section provides the following information on how to implement the `pollForEvents()` method:

Basic logic for pollForEvents()

The `pollForEvents()` method typically uses a basic logic for event processing. Figure 53 shows a flow chart of the poll method's basic logic.

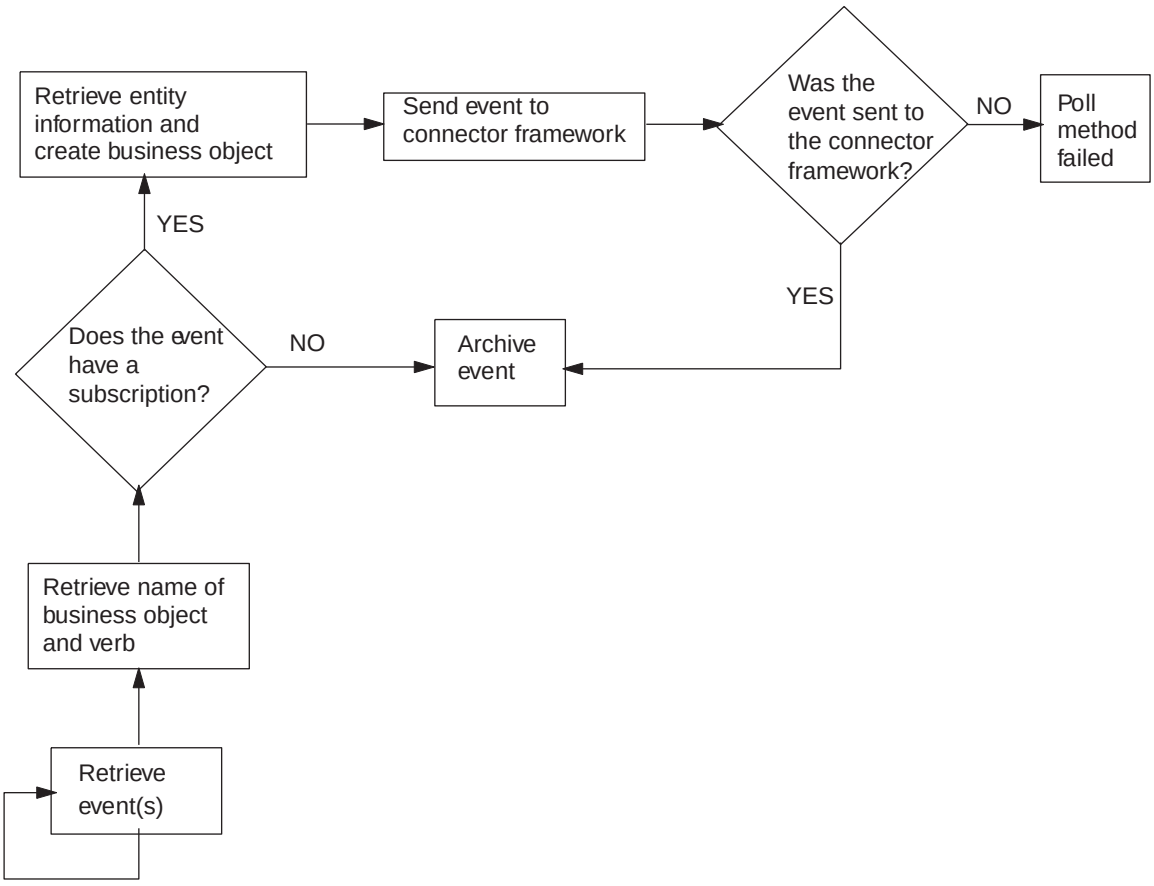

Figure 53. Flow chart for basic logic of pollForEvents()

For an implementation of this basic polling logic, see "Polling for events" on page 177."Implementing an event-notification mechanism" on page 189.

Other polling issues

This section provides information on the following polling issues:

- "Archiving events"
- "Threading issues" on page 125
- "Processing events by event priority" on page 125
- "Event distribution" on page 126

Archiving events

Once a connector has processed an event, it can archive the event. Archiving processed or unsubscribed events ensures that events are not lost. Archiving usually involves the following steps:

- Copy the event record from the event store to the archive store.

  The *archive store* serves the same basic purpose as an event store: it saves archive records in a persistent cache until the connector can process them. An *archive record* contains the same basic information as an event record.
- Update the event status of the event in the archive store.

The archive record should be updated with one of the event-status values in Table 42..

- Delete the event record from the event store.

Table 42. Event-status values in an archive record

Status	Description
Success	The event was detected, and the connector created a business object for the event and sent the business object to the connector framework.
Unsubscribed	The event was detected, but there were no subscriptions for the event, so the event was not sent to the connector framework and on to the integration broker.
Error	The event was detected, but the connector encountered an error when trying to process the event. The error occurred either in the process of building a business object for the event or in sending the business object to connector framework.

This section provides the following information about event archiving:

- "Creating an archive store"
- "Configuring a connector for archiving"
- "Accessing the archive store"

Creating an archive store: If the application provides archiving services, you can use those; otherwise, an archive store is usually implemented using the same mechanism as the event store:

- For an event-notification mechanism that uses database triggers, one way to set up event archiving is to install a delete trigger on the event table. When the connector's application-specific component deletes a processed or unsubscribed event from the event table, the delete trigger moves the event to the archive table. For information on event tables, see "Event table" on page 113.

  Note: If a connector uses an event table, an administrator might need to clean up the archive periodically.
- With an email event notification scheme, archiving might consist of moving a message to a different folder. A folder called `Archive` might be used for archiving event messages.

Configuring a connector for archiving: Archiving can have performance impact in the form of the archive store and moving the event records into this store. Therefore, you might want to design event archiving to be configurable at install time, so that a system administrator can control whether events are archived. To make archiving configurable, you can create a connector-specific configuration property that specifies whether the connector archives unsubscribed events. IBM suggests a name of `ArchiveProcessed` for this configuration property. If the configuration property specifies no archiving, the connector application-specific component can delete or ignore the event. If the connector is performance-constrained or the event volume is extremely high, archiving events is not required.

Accessing the archive store: A connector performs archiving as part of the event processing in its poll method, `pollForEvents()`. Once a connector has processed an event, the connector must move the event to an archive store whether or not the event was successfully delivered to the connector framework. Events that have no subscriptions are also moved to the archive. Archiving processed or unsubscribed events ensures that events are not lost.

Your poll method should consider archiving an event when any of the following conditions occur:

- When the poll method has processed the event and the connector framework has delivered the business object
- When no subscriptions exist for the event

Note: If a connector uses an event table, an administrator might need to clean up the archive periodically. For example, the administrator may need to truncate the archive to free disk space.

Threading issues

By default, C++ connectors are single-threaded. Therefore, the connector framework does not accept business objects from InterChange Server while the poll method is running. The connector framework calls the poll method only when it is not processing request business objects or other messages. Therefore, the poll method should not take a long time to complete.

Because request processing is blocked when the poll method is processing events, you need to be careful implementing the poll method to process multiple events per poll:

- Implementing the poll method to return after processing a single event minimizes the amount of time required for polling. However, if the application generates a large number of events, processing one event at a time may not provide adequate performance.
- If you implement polling to process multiple events per poll, make sure that the poll implementation balances the need to process large numbers of events with the need to handle business object requests from the integration broker.

Note: If a connector is configured to run in parallel-process mode (with ParallelProcessDegree greater than 1), it consists of several processes, each with a particular purpose, as shown in Table 108 on page 292.. Such a connector does not block request processing while it executes the poll method.

Processing events by event priority

Event priority enables the connector poll method to handle situations where the number of events in the event store exceeds the maximum number of events the connector retrieves in a single poll. In this type of polling implementation, the poll method polls and processes events in order of priority. Event priority is defined as an integer value in the range 0 - n, with 0 as the highest priority.

To process events by event priority, the following tasks must be implemented in the event notification mechanism:

- The event detection mechanism must assign a priority value to an event record when it saves it to the event store.
- The event retrieval mechanism (the polling mechanism) must specify the order in which it retrieves event records to process, based on the event priority.

Note: As events are picked up, event priority values are *not* decremented. In rare circumstances, this might lead to low priority events being not picked up.

The following example SQL SELECT statement shows how a connector might select event records based on event priority. The SELECT statement sorts the events by priority, and the connector processes each event in turn.

```
SELECT event_id, object_name, object_verb, object_key
FROM event_table
WHERE event_status = 0 ORDER BY event_priority
```

The logic for a poll method is then the same as discussed in "Basic logic for pollForEvents()" on page 123.

Event distribution

The event detection and retrieval mechanisms can be implemented so that multiple connectors can poll the same event store. Each connector can be configured to process certain events, create specific business objects and pass those business objects to InterChange Server. This can streamline the processing of certain types of events and increase the transfer of data out of an application.

To implement event distribution so that multiple connectors can poll the event store, do the following:

- Add a column to the event record for an integer connector identifier (ID), and design the event detection mechanism to specify which connector will pick up the event.

  This might be done per application entity. For example, the event detection mechanism might specify that all `Customer` events be picked up by the connector that has the `connectorId` field set to 4.
- Add an application-specific connector property named `ConnectorId`. Assign each connector a unique identifier and store this value in its `ConnectorId` property.
- Implement the poll method to query for the value of the `ConnectorId` property. If the property is not set, the poll method can retrieve *all* event records from the event store as usual. If the property is set to a connector identifier value, the poll method retrieves *only* those events that match the `ConnectorId`.

Special considerations for event processing

This section contains the following information about event processing:

- "Processing Delete events"
- "Using guaranteed event delivery" on page 128

Processing Delete events

An application can support one of the following types of delete operations:

- Physical delete—Data is physically deleted from the database.
- Logical delete—A status column in a database entity is set to an inactive or invalid status, but the data is not deleted from the database.

It may be tempting to implement delete event processing in a manner that is consistent with the application. For example, when an application entity is deleted, a connector poll method for an application that supports physical deletes might publish a business object with the Delete verb. A connector poll method for an application that supports logical deletes might publish a business object with the Update verb and the status value changed to inactive.

Problems can arise with this approach when a source application and a destination application support different delete models. Suppose that the source application supports logical delete and the destination application supports physical delete. Assume that an enterprise is synchronizing between the source and destination applications. If the source connector sends a change in status (in other words, a

delete event) as a business object with the Update verb, the destination connector might be unable to determine that the business object actually represents a delete event.

Therefore, event publishing must be designed so that source connectors for both types of applications can publish delete events in such a way that destination connectors can handle the events appropriately. The Delete verb in an event notification business object should represent an event where data was deleted, whether the delete operation was a physical or logical delete. This ensures that destination connectors will be correctly informed about a delete event.

This section provides the following information on how to implement event processing for delete events:

- "Setting the verb in the event record"
- "Setting the verb in the business object"
- "Setting the verb during mapping" on page 128

Setting the verb in the event record

The event detection mechanism for both logical and physical delete connectors should set the verb in the event record to Delete:

- For a physical delete connector, this is the standard implementation.
- For a connector whose application supports logical deletes, the event detection mechanism must be designed to determine when update events actually represent deletion of data.

  In other words, it must differentiate update events for modified entities from update events for logically deleted entities. For logically deleted entities, the event detection mechanism should set the verb in the event record to Delete even if the event in the application was an Update event that updated a status column.

Setting the verb in the business object

The poll method for both logical and physical delete connectors should generate a business object with the Delete verb:

- If the application supports logical deletes, the connector poll method retrieves the delete event from the event store, creates an empty business object, sets the key, sets the verb to Delete, and sends the business object to the connector framework.

  For hierarchical business objects, the connector should *not* send deleted children. The connector can constrain queries to not include entities with status of inactive, or child business objects with a status of inactive can be removed in mapping.
- If the application supports physical deletes, the connector might not be able to retrieve the application data. In this case, the connector poll method retrieves the delete event from the event store, creates an empty business object, sets the key values, sets the values of other attributes to the special Ignore value (`CxIgnore`), sets the verb in the business object to Delete, and sends the business object to the connector framework.

Setting the verb during mapping

WebSphere InterChange Server

Mapping between the application-specific business object and the generic business object should map the verb as Delete. This ensures that the correct information about an event is sent to the collaboration, which may perform special processing based on the verb.

Follow these recommendations for relationship tables:

- For delete events for a logical delete application, leave relationship entries in the relationship table.
- For delete events for a physical delete application, delete relationship entries from the relationship table.

Using guaranteed event delivery

The guaranteed-event-delivery feature enables the connector framework to guarantee that events are never sent twice between the connector's event store and the integration broker.

Important: This feature is available *only* for JMS-enabled connectors; that is, those connectors that use Java Messaging Service (JMS) to handle queues for their message transport. A JMS-enabled connector always has its `DeliveryTransport` connector property set to `JMS`. When the connector starts, it uses the JMS transport; all subsequent communication between the connector and the integration broker occurs through this transport. The JMS transport ensures that the messages are eventually delivered to their destination.

Without use of the guaranteed-event-delivery feature, a small window of possible failure exists between the time that the connector publishes an event (when the connector calls the `gotApplEvent()` method within its `pollForEvents()` method) and the time it updates the event store by deleting the event record (or perhaps updating it with an "event posted" status). If a failure occurs in this window, the event has been sent but its event record remains in the event store with a "ready for poll" status. When the connector restarts, it finds this event record still in the event store and sends it, resulting in the event being sent twice.

You can provide the guaranteed-event-delivery feature to a JMS-enabled connector in one of the following ways:

- With the *container-managed-events* feature: If the connector uses a JMS event store (implemented as a JMS source queue), the connector framework act as a container and manage the JMS event store. For more information, see "Guaranteed event delivery for connectors with JMS event stores."
- With the *duplicate-event-elimination* feature: The connector framework can use a JMS monitor queue to ensure that no duplicate events occur. This feature is usually used for a connector that uses a non-JMS event store (for example, implemented as a JDBC table, Email mailbox, or flat files). For more information, see "Guaranteed event delivery for connectors with non-JMS event stores" on page 131.

Guaranteed event delivery for connectors with JMS event stores

If the JMS-enabled connector uses JMS queues to implement its event store, the connector framework can act as a "container" and manage the JMS event store (the JMS source queue). One of the roles of JMS is to ensure that once a transactional

queue session starts, the messages are cached there until a commit is issued; if a failure occurs or a rollback is issued, the messages are discarded. Therefore, in a single JMS transaction, the connector framework can remove a message from a source queue and place it on the destination queue. This *container-managed-events* feature of guaranteed event delivery enables the connector framework to guarantee that events are never sent twice between the JMS event store and the destination's JMS queue.

This section provides the following information about use of the guaranteed-event-delivery feature for a JMS-enabled connector that has a JMS event store:

- "Enabling the feature for connectors with JMS event stores"
- "Effect on event polling" on page 130

Enabling the feature for connectors with JMS event stores: To enable the guaranteed-event-delivery feature for a JMS-enabled connector that has a JMS event store, set the connector configuration properties shown in Table 43..

Table 43. Guaranteed-event-delivery connector properties for a connector with a JMS event store

Connector property	Value
`DeliveryTransport`	`JMS`
`ContainerManagedEvents`	`JMS`
`PollQuantity`	The number of events to processing in a single poll of the event store
`SourceQueue`	Name of the JMS source queue (event store) which the connector framework polls and from which it retrieves events for processing **Note:** The source queue and other JMS queues should be part of the same queue manager. If the connector's application generates events that are stored in a different queue manager, you must define a remote queue definition on the remote queue manager. WebSphere MQ can then transfer the events from the remote queue to the queue manager that the JMS-enabled connector uses for transmission to the integration broker. For information on how to configure a remote queue definition, see your IBM WebSphere MQ documentation.

Note: A connector can use *only one* of these guaranteed-event-delivery features: container managed events or duplicate event elimination.Therefore, you cannot set the `ContainerManagedEvents` property to `JMS` *and* the `DuplicateEventElimination` property to `true`.

In addition to configuring the connector, you must also configure the data handler that converts between the event in the JMS store and a business object. This data-handler information consists of the connector configuration properties that Table 44 summarizes.

Table 44. Data-handler properties for guaranteed event delivery

Data-handler property	Value	Required?
`MimeType`	The MIME type that the data handler handles. This MIME type identifies which data handler to call.	Yes
`DHClass`	The full name of the Java class that implements the data handler	Yes
`DataHandlerConfigMOName`	The name of the top-level meta-object that associates MIME types and their data handlers	Optional

Note: The data-handler configuration properties reside in the connector configuration file with the other connector configuration properties.

End users that configure a connector that has a JMS event store to use guaranteed event delivery must be instructed to set the connector properties as described in Table 43 *and* Table 44.. To set these connector configuration properties, use the Connector Configurator tool. Connector Configurator displays the connector properties in Table 43 on its Standard Properties tab. It displays the connector properties in Table 44 on its Data Handler tab.

Note: Connector Configurator activates the fields on its Data Handler tab only when the `DeliveryTransport` connector configuration property is set to `JMS` and `ContainerManagedEvents` is set to `JMS`.

For information on Connector Configurator, see Appendix B, "Connector Configurator," on page 329.Appendix B, "Connector Configurator," on page 527.

Effect on event polling: If a connector uses guaranteed event delivery by setting `ContainedManagedEvents` to `JMS`, it behaves slightly differently from a connector that does not use this feature. To provide container-managed events, the connector framework takes the following steps to poll the event store:

1. Start a JMS transaction.
2. Read a JMS message from the event store.

   The event store is implemented as a JMS source queue. The JMS message contains an event record. The name of the JMS source queue is obtained from the `SourceQueue` connector configuration property.
3. Call the appropriate data handler to convert the event to a business object.

   The connector framework calls the data handler that has been configured with the properties in Table 44..
4. When a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, WebSphere Business Integration Message Broker) or WebSphere Application Server is the integration broker, convert the business object to a message based on the configured wire format (XML).
5. Send the resulting message to the JMS destination queue.

WebSphere InterChange Server

The message sent to the JMS destination queue is the business object.

Other integration brokers
The message sent to the JMS destination queue is an XML message.

6. Commit the JMS transaction.

   When the JMS transaction commits, the message is written to the JMS destination queue and removed from the JMS source queue in the same transaction.
7. Repeat step 1 through 6 in a loop. The `PollQuantity` connector property determines the number of repetitions in this loop.

Important: A connector that sets the `ContainerManagedEvents` property is set to `JMS` does *not* call the `pollForEvents()` method to perform event polling. If the connector's base class includes a `pollForEvents()` method, this method is *not* invoked.

Guaranteed event delivery for connectors with non-JMS event stores

The connector framework can use *duplicate event elimination* to ensure that duplicate events do not occur. This feature is usually enabled for JMS-enabled connectors that use a non-JMS solution to implement an event store (such as a JDBC event table, Email mailbox, or flat files). This duplicate-event-elimination feature of guaranteed event delivery enables the connector framework to guarantee that events are never sent twice between the event store and the destination's JMS queue.

Note: JMS-enabled connectors that use a JMS event store usually use the container-managed-events feature. However, they can use duplicate event elimination instead of container managed events.

This section provides the following information about use of the guaranteed-event-delivery feature with a JMS-enabled connector that has a non-JMS event store:

- "Enabling the feature for connectors with non-JMS event stores"
- "Effect on event polling" on page 130

Enabling the feature for connectors with non-JMS event stores: To enable the guaranteed-event-delivery feature for a JMS-enabled connector that has a non-JMS event store, you must set the connector configuration properties shown in Table 45..

Table 45. Guaranteed-event-delivery connector properties for a connector with a non-JMS event store

Connector property	Value
`DeliveryTransport`	`JMS`
`DuplicateEventElimination`	`true`
`MonitorQueue`	Name of the JMS monitor queue, in which the connector framework stores the ObjectEventId of processed business objects

Note: A connector can use *only one* of these guaranteed-event-delivery features: container managed events or duplicate event elimination.Therefore, you

cannot set the DuplicateEventElimination property to true *and* the ContainerManagedEvents property to JMS.

End users that configure a connector to use guaranteed event delivery must be instructed to set the connector properties as described in Table 45.. To set these connector configuration properties, use the Connector Configurator tool. It displays these connector properties on its Standard Properties tab. For information on Connector Configurator, see Appendix B, "Connector Configurator," on page 329.Appendix B, "Connector Configurator," on page 527.

Effect on event polling: If a connector uses guaranteed event delivery by setting DuplicateEventElimination to true, it behaves slightly differently from a connector that does not use this feature. To provide the duplicate event elimination, the connector framework uses a JMS monitor queue to track a business object. The name of the JMS monitor queue is obtained from the MonitorQueue connector configuration property.

After the connector framework receives the business object from the application-specific component (through a call to gotApplEvent() in the pollForEvents() method), it must determine if the current business object (received from gotApplEvents()) represents a duplicate event. To make this determination, the connector framework retrieves the business object from the JMS monitor queue and compares its ObjectEventId with the ObjectEventId of the current business object:

- If these two ObjectEventIds are the same, the current business object represents a duplicate event. In this case, the connector framework ignores the event that the current business object represents; it does *not* send this event to the integration broker.
- If these ObjectEventIds are *not* the same, the business object does *not* represent a duplicate event. In this case, the connector framework copies the current business object to the JMS monitor queue and then delivers it to the JMS delivery queue, all as part of the same JMS transaction. The name of the JMS delivery queue is obtained from the DeliveryQueue connector configuration property. Control returns to the connector's pollForEvents() method, after the call to the gotApplEvent() method.

For a JMS-enabled connector to support duplicate event elimination, you must make sure that the connector's pollForEvents() method includes the following steps:

- When you create a business object from an event record retrieved from the non-JMS event store, save the event record's unique event identifier as the business object's ObjectEventId attribute.

  The application generates this event identifier to uniquely identify the event record in the event store. If the connector goes down after the event has been sent to the integration broker but before this event record's status can be changed, this event record remains in the event store with an In-Progress status. When the connector comes back up, it should recover any In-Progress events. When the connector resumes polling, it generates a business object for the event record that still remains in the event store. However, because both the business object that was already sent and the new one have the same event record as their ObjectEventIds, the connector framework can recognize the new business object as a duplicate and not send it to the integration broker.

  A C++ connector can use the setAttrValue() method of the BusinessObject class to assign the event identifier to the ObjectEventId attribute, as follows:

```
pBusObj->setAttrValue("ObjectEventId", strngEventId, BOATTRTYPE::STRING);
```

- During connector recovery, make sure that you process In-Progress events *before* the connector begins polling for new events.

  Unless the connector changes any In-Progress events to Ready-for-Poll status when it starts up, the polling method does not pick up the event record for reprocessing.

Chapter 6. Message logging

This chapter presents information on message logging. A *message* is a string of information that the connector can send to an external connect log, where it can be reviewed by the system administrator or the developer to provide information about the runtime state of the connector. There are two different categories of messages that a connector can send to the connector log:

- Error or informational messages
- Trace messages

Messages can be generated within the connector code or obtained from a message file. This chapter contains the following sections:

Error and informational messages

A connector can send information about its state to a log destination. The following types of information are recommended for logging:

- Errors and fatal errors from your code to a log file.
- Warnings require a system administrator's attention, from your code to a log file.
- Informational messages such as:
  - Connector startup and termination messages
  - Important messages from the application

Although a connector can send informational or error messages, this logging process is referred to as *error logging*.

Note: These messages are independent of any trace messages defined for the connector.

Indicating a log destination

A connector sends its log messages into its log destination. The *log* is an external destination that is available for viewing by those needing to review the execution state of the connector. The log destination is defined at connector configuration time by the setting of the Logging field in the Trace/Log Files tab of Connector Configurator as one of the following:

- To File: The absolute pathname of an external file, which must reside on the same machine as the connector's process (with its connector framework and application-specific component)
- To console (STDOUT): The command prompt window generated when the connector startup script starts the connector

By default, the connector's log destination is set to the console, which indicates use of the startup script's command prompt window as the log destination. Set this log destination as appropriate for your connector.

WebSphere InterChange Server

You can also set the LogAtInterchangeEnd connector configuration property to indicate whether messages are also logged to the InterChange Server's log destination:

- Messages logged locally *only*: LogAtInterchangeEnd is false.
- Messages are logged both locally *and* sent to InterChange Server's log destination: LogAtInterchangeEnd is true.

By default, LogAtInterchangeEnd is set to false, so that messages are only logged locally. If messages are sent to InterChange Server, they are written to the destination specified for InterChange Server messages.

Note: Logging to InterChange Server's log destination also turns on email notification, which generates email messages for the MESSAGE_RECIPIENT parameter specified in the InterchangeSystem.cfg file when errors or fatal errors occur. As an example, when a connector loses its connection to its application, if LogAtInterchangeEnd is set to true, an email message is sent to the specified message recipient.

These connector properties are set with Connector Configurator. For more information on InterChange Server's message logging, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set.

Sending a message to the log destination

Table 46 shows the ways that a connector sends an error, warning, and information message to its log destination.

Table 46. Methods for sending a message to the log destination

Connector library method	Description
logMsg() and generateMsg()	Takes as input a text string or a string generated from a message in a message file. Optionally, it can take a message-type constant to indicate whether the message is an error, warning, or informational. To generate a character string from the message text in a message file, use the generateMsg() method.
generateAndLogMsg()	Combines the functionality of the logMsg() and generateMsg() methods into a single call.

For more information on how to generate a message, see "Generating a message string" on page 141.

In the C++ connector library, the logMsg(), generateMsg(), and generateAndLogMsg() methods are defined in two classes:

- In the GenGlobals for access to logging from within the connector base class
- In the BOHandlerCPP class for access to logging from within a business object handler.

Both the generateMsg() and generateAndLogMsg() methods require a message type as an argument. This argument indicates the severity of the message. For more information, see "Generating a message string" on page 141.

Trace messages

Tracing is an optional troubleshooting and debugging feature that can be turned on for connectors. When tracing is turned on, system administrators can follow events as they work their way through the IBM WebSphere business integration system.

WebSphere InterChange Server

When InterChange Server is the integration broker, you can also use tracing on connector controllers, and other components of the InterChange Server system.

Tracing in an application-specific component allows you and other users of your connector code to monitor the behavior of the connector. Tracing can also track when specific connector functions are called by the connector framework. Trace messages that you provide for the connector application-specific code augment the trace messages provided for the connector framework.

Enabling tracing

By default, tracing on a connector is turned off. Tracing is turned on for a connector when the connector configuration property `TraceLevel` is set to a non-zero value in Connector Configurator. You can set `TraceLevel` to a value from 1 to 5 to obtain the appropriate level of detail. Level 5 tracing logs the trace messages of *all* lower trace levels.

WebSphere InterChange Server

Tip: For information on turning on tracing for connector controllers or for other components of the InterChange Server system, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set.

Identifying a trace destination

A connector sends its trace messages into its trace destination, which is an external destination that is available for viewing by those needing to review the execution state of the connector. The trace destination is defined at connector configuration time by the setting of the Tracing field in the Trace/Log Files tab of Connector Configurator as one of the following:

- To File: The absolute pathname of an external file, which must reside on the same machine as the connector's process (with its connector framework and application-specific component)
- To console (STDOUT): The command prompt window generated when the connector startup script starts the connector

By default, the connector's trace destination is set to the console, which indicates use of the startup script's command prompt window as the trace destination. Set this trace destination as appropriate for your connector.

Sending a trace message to the trace destination

Table 47 shows the ways that a connector sends a trace message to its trace destination.

Table 47. Methods for sending a trace message to the trace destination

Connector library method	Description
`traceWrite()` and `generateMsg()`	Takes as input a text string or a string generated from a message in a message file and a trace-level constant to indicate the trace level. This method writes a trace message for the specified trace level or greater to the trace destination. To generate a character string from the message text in a message file, use the `generateMsg()` method with the message type set to `XRD_TRACE`.
`generateAndTraceMsg()`	Combines the functionality of the `traceWrite()` and `generateMsg()` methods into a single call.

For information on the `generateMsg()` method, see "Generating a message string" on page 141.

Note: It is *not* required that trace messages be localized in the message file. Whether trace messages are contained in a message file is left at the discretion of the developer. For more information, see "Locale-sensitive design principles" on page 56..

In the C++ connector library, the `traceWrite()`, `generateMsg()`, and `generateAndTraceMsg()` methods are defined in two classes:

- In the `GenGlobals` for access to tracing from within the connector base class
- In the `BOHandlerCPP` class for access to tracing from within a business object handler

The `traceWrite()` and `generateAndTraceMsg()` require a trace level as an argument. This argument specifies the trace level to use for a trace message. When you turn on tracing at runtime, you specify a trace level at which to run the tracing. All trace messages in your code with trace levels at or below the runtime trace level are sent to the trace destination. For more information, see "Recommended content for trace messages" on page 139.

To specify a trace level to associate with a trace message, use a trace-level constant of the form `LEVEL`*n* where *n* can be a trace level from 1 to 5. Trace-level constants are defined in the `Tracing` class.

The C++ code fragment below uses `traceWrite()` to write a level 4 trace message to log the number of records retrieved from the application.

```
sprintf(msg, "Fetched %d record(s).", rCount);
traceWrite(Tracing::LEVEL4, msg, NULL);
```

The trace message written to the trace destination contains the date, time, connector name, and message, as shown by the output of this code sample:

```
[1999/05/28 12.36:48.105] [ConnectorAgent MyConnector]
                        Trace: Fetched 2 record(s).
```

Both the `generateMsg()` and `generateAndTraceMsg()` methods require a message type as an argument. This argument indicates the severity of the message. Because trace messages do not have severity levels, you just use the `XRD_TRACE` message-type constant. Message-type constants are defined in the `CxMsgFormat` class.

Recommended content for trace messages

You are responsible for defining what kind of information your connector returns at each trace level. Table 48 shows the recommended content for application-specific connector trace messages.

Table 48. Content of application-specific connector trace messages

Level	Content
0	Trace message that identifies the connector version. No other tracing is done at this level.
1	Trace messages that: • Log status messages and identifying (key) information for each business object processed. • Record each time the `pollForEvents()` method is executed.
2	Trace messages that: • Identify the business object handlers used for each object the connector processes. • Log each time a business object is posted to InterChange Server, either from `gotApplEvent()` or `executeCollaboration()`. • Indicate each time a request business object is received.
3	Trace messages that: • Identify the foreign keys being processed (if applicable). These messages appear when the connector has encountered a foreign key in a business object or when the connector sets a foreign key in a business object. • Relate to business object processing. Examples of this include finding a match between business objects, or finding a business object in an array of child business objects.
4	Trace message that: • Identify application-specific information. Examples of this information include the values returned by the methods that process the application-specific information fields in business objects. • Identify when the connector enters or exits a function. These messages help trace the process flow of the connector. • Record any thread-specific processing. For example, if the connector spawns multiple threads, a message should log the creation of each new thread.
5	Trace message that: • Indicate connector initialization. Examples of this message include the value of each connector configuration property that has been retrieved from InterChange Server. • Detail the status of each thread the connector spawns while it is running. • Represent statements executed in the application. The connector log file contains all statements executed in the target application and the value of any variables that are substituted (where applicable). • Record business object dumps. The connector should output a text representation of a business object before it begins processing (showing the object the connector receives from the integration broker) and after it has processed the object (showing the object the connector returns to the integration broker).

Note: The connector should deliver all the trace messages at the specified trace level and lower.

For information on the content and level of detail for connector framework trace messages, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set.

Message file

You can provide message input to the connector error logging or tracing method be as text strings or as references to a *message file*. A message file is a text file containing message numbers and message text. The message text can contain positional parameters for passing runtime data out of your connector. You can provide a message file by creating a file and defining the messages that you need.

WebSphere InterChange Server

Important: Do *not* add your messages to the InterChange Server message file, `InterchangeSystem.txt`. Access *only* existing messages from this system message file.

This section provides the following information about a message file:

- "Message format"
- "Name and location of a message file"
- "Generating a message string" on page 141

Message format

Within a message file, messages have the following format:

message number message text[EXPL]*explanation text*

The *message number* is an integer that uniquely identifies the message. This message number must appear on one line. The *message text* can span multiple lines, with a carriage return terminating each line. The *explanation text* is a more detailed explanation of the condition that causes the message to occur.

As an example of message text, a connector can call the following message when it determines that its version differs from the version of the connector infrastructure.

```
20017
Connector Infrastructure version does not match.
```

Messages can contain parameters whose values are replaced at runtime by values from the program. The parameters are positional and are indicated in the message file by a number in braces. For example, the following message has two parameters to record an unsubscribed event.

```
20026
Warning: Unsubscribed event: Object Name:{1}, Verb: {2}.
```

For information on how to provide values to message parameters, see "Using parameter values" on page 143.

Note: For additional examples of messages, see the InterChange Server message file `InterchangeSystem.txt`.

Name and location of a message file

A connector can obtain its messages from one of two message files:

- A connector message file is named *Appname*`Connector.txt` and is stored in the following subdirectory of the product directory:

  ```
  connectors\messages
  ```

For example, the connector message file for the IBM WebSphere Business Integration Adapter for Clarify is named `ClarifyConnector.txt`.

- The InterChange Server message file is named `InterchangeSystem.txt` and is located in the product directory.

All methods that generate messages (see Table 49 on page 141) first search the connector message file for the specified message number.

WebSphere InterChange Server

If a connector message file does *not* exist, the InterChange Server message file `InterchangeSystem.txt` (located in the product directory) is used as the message file.

The connector message file should contain all text strings that the application-specific component uses. These strings include those for logging as well as hardcoded strings.

Note: Connector standards suggest that trace messages are *not* included in a connector message file because end users do not normally view them.

For an internationalized connector, it is important that text strings are isolated into the connector message file. This message file can be translated and the messages can then be easily available in different languages. The name of the translated connector message file should include the name of the associated locale, as follows:

Appname`Connector_`*ll*`_`*TT*`.txt`

In the preceding line, *ll* is the two-letter abbreviation for the locale (by convention in lowercase letters) and *TT* is the two-letter abbreviation for the territory (by convention in uppercase letters). For example, the version of the connector message file for the WBI Adapter for Clarify that contains U.S. English messages would have the following name:

`ClarifyConnector_en_US.txt`

At runtime, the connector framework locates the appropriate message file for the connector framework locale from the `connectors\messages` subdirectory. For example, if the connector framework's locale is U.S. English (`en_US`), the connector framework retrieves messages from the *Appname*`Connector_en_US.txt` file.

For more information on how to internationalize the text strings of a connector, see "An internationalized connector" on page 55..

Generating a message string

The methods in Table 49 retrieve a predefined message from a message file, format the text, and return a generated message string.

Table 49. Methods that generate a message string

Message method	Description
`generateMsg()`	Generates a message of the specified severity from a message file. You can use the message as input to the `logMsg()` or `traceWrite()` method.

Table 49. Methods that generate a message string (continued)

Message method	Description
`generateAndLogMsg()`	Generates a message of the specified severity from a message file and sends it to the log destination
`generateAndTraceMsg()`	Generates a trace message from a message file and sends it to the log destination

Tip: Before using `generateMsg()` for tracing, check that tracing is enabled with the `isTraceEnabled()` method. If tracing is *not* enabled, you need not generate the trace message.

In the C++ connector library, the `generateMsg()`, `generateAndLogMsg()`, and `generateAndTraceMsg()` methods are defined in two classes:

- In the `GenGlobals` for access to logging from within the connector base class
- In the `BOHandlerCPP` class for access to logging from within a business object handler.

The message-generation methods in Table 49 require the following information:

- "Specifying a message number"
- "Specifying a message type"
- "Using parameter values" on page 143 (optional)

Specifying a message number

The methods in Table 49 require a message number as an argument. This argument specifies the number of the message to obtain from the message file. As described in "Message format" on page 140, each message in a message file must have a unique integer message number (identifier) associated with it. The methods in Table 49 search the message file for the specified message number and extract the associated message text.

The IBM WebSphere business integration system generates the date and time and displays the following message:

```
[1999/05/28 11:54:15.990] [ConnectorAgent ConnectorName]
Error 1100: Failed to connect to application
```

Note: If the connector logs to its local log file, the connector infrastructure adds the timestamp.

WebSphere InterChange Server

If the connector logs to InterChange Server, InterChange Server adds the timestamp.

Specifying a message type

The methods in Table 49 also require a message type as an argument. This argument indicates the severity of the message. Table 50 lists the valid message types and their associated message-type constants.

Table 50. Message types

Message type	Severity level	Description
XRD_FATAL	Fatal Error	Indicates an error that stops program execution.
XRD_ERROR	Error	Indicates a error that should be investigated.
XRD_WARNING	Warning	Indicates a condition that might represent a problem but that can be ignored.
XRD_INFO	Informational	Information message only; no action required.
XRD_TRACE	--	Use for trace messages.

To specify a message type to associate with a message, use one of the message-type constants in Table 50 as an argument to the message-generation method, as follows:

- For a log message, use a message-type constant that indicates the message severity (in decreasing level of severity): XRD_FATAL, XRD_ERROR, XRD_WARNING, XRD_INFO.
- For a trace message, use the XRD_TRACE message-type constant.

Message-type constants are defined in the CxMsgFormat class.

The following C++ code fragment logs an error message when it cannot connect to an application. The error message is defined in a connector message file as message 1100, Error: Failed to connect to application. The code example includes a comment explaining the message number; this makes code more readable. The example also calls freeMemory() to release memory allocated by generateMsg().

```
char * msg;
ret_code = connect_to_app(userName, password);
// Message 1100 - Failed to connect to application
if (ret_code == -1) {
      msg = generateMsg(1100, CxMsgFormat::XRD_ERROR, NULL, 0, NULL);
      logMsg(msg);
      JToCPPVeneer::getTheHandlerStuff()->freeMemory(msg);
      return BON_FAIL;
}
```

Note: For a C++ connector, after generating a message with generateMsg() and logging the message, call void freeMemory(char * mem) to release memory allocated by generateMsg().

Using parameter values

With the message-generation methods in Table 49, you can specify an optional number of values for message-text parameters. The number of parameter values must match the number of parameters defined in the message text. For information on how to define parameters in a message, see "Message format" on page 140.

To specify parameter values, you must include the following arguments:

- An argument count to indicate the number of parameters within the message text; to determine the number, refer to the message in the message file.
- A comma-separated list of parameter values; each parameter is represented as a character string.

Suppose you have the following informational message in your connector message file that contains one parameter:

```
2887
Initializing connector {1}
```

Because this message contains a single parameter, a call to one of the message-generation methods must specify an argument count of 1 and then provide the name of the connector as a character string. In the code fragment below, `generateAndLogMsg()` is called to format a message that contains one parameter and send this message to the log:

```
char val[512];
getConfigProp("ConnectorName", val, 512);
// Message 2887 - Initializing connector
generateAndLogMsg(2887, CxMsgFormat::XRD_INFO, 1, val);
```

The parameter value of `val` is combined with the message in the message file. If `val` is set to `MyConnector`, the resulting message is:

```
[1999/05/28 11:54:15.990] [ConnectorAgent MyConnector]
       Info 2887: Initializing connector MyConnector
```

You can also locate trace messages in the connector message file.

Suppose you have the following trace message in your connector message file that contains one parameter:

```
3033
Opened main form for {1} object.
```

Because this message contains a single parameter, a call to one of the message-generation methods must specify an argument count of 1 and then provide the name of the form as a character string. In the code fragment below, `generateAndTraceMsg()` is called to format a message that contains one parameter and send this trace message to the log:

```
char * formName[512];
if(getFormName(theObj,formName)==0)
               return BON_FAIL;
if(tracePtr->getTraceLevel()>= Tracing::LEVEL3) {
              // Message 3033 - Opened main form for object
             generateAndTraceMsg(3033,CxMsgFormat::XRD_TRACE,
                    Tracing::LEVEL3, 1, formName);
}
```

This code fragment retrieves the application form name and calls Tracing::getTraceLevel() to retrieve the current tracing level as set in the repository. If the trace level is at least 3, the routine uses `generateAndTraceMsg()` to generate a message string and write the trace message to the log destination. The call to `generateMsg()` specifies that the value of `argCount` is 1 and val contains a character string for the form name.

For the Item object, the trace message displayed is:

```
[1999/05/28 12:00:00.000] [ConnectorAgent MyConnector]
                 Trace 3033: Opened main form for Item object
```

Chapter 7. Implementing a C++ connector

This chapter presents information on how to implement a connector's application-specific component in C++. It provides language-specific details for the general tasks discussed in earlier chapters of this guide.

This chapter contains the following sections:

Extending the C++ connector base class

In the C++ connector library, the connector base class is named `GenGlobals`. For a C++ connector, the base class methods are pure virtual methods. The `GenGlobals` class provides methods for connector startup and shut down, access to connector configuration properties, and utility methods for logging and tracing. To implement your own connector, you extend this connector base class to create your own *connector class*.

Note: For general information about the methods of the connector base class, see "Extending the connector base class" on page 68..

To derive a connector class for a C++ connector, follow these steps.

1. Create a connector class that extends the `GenGlobals` class, and include the header file `GenGlobals.hpp`. A suggested name for this class is:

   connectorName`Globals.cpp`

   where *connectorName* uniquely identifies the application or technology with which the connector communicates. For example, to create a connector that communicates with a Baan application, you could name the connector class `BaanGlobals.cpp`.

   Note: For information on naming conventions for a connector, see *Naming IBM WebSphere InterChange Server Components* in the IBM WebSphere InterChange Server documentation set.
2. Implement the `GenGlobals` pure virtual methods for the connector methods. For more information on how to create these virtual methods, see Table 51..
3. Provide the connector framework with a handle to your global connector class.

Table 51. Extending virtual methods of the GenGlobals class

Virtual GenGlobals method	Description	For more information
`init()`	Initializes the application-specific component of the connector.	"Initializing the connector" on page 146
`getVersion()`	Obtain the version of the connector's application-specific component.	"Checking the connector version" on page 147
`getBOHandlerforBO()`	Obtain the business-object handler for the business objects.	"Obtaining the C++ business object handler" on page 148

Table 51. Extending virtual methods of the GenGlobals class (continued)

Virtual GenGlobals method	Description	For more information
`doVerbFor()`	Process the request business object by performing its verb operation.	"Creating a business object handler" on page 149
`pollForEvents()`	Poll event store to obtain application events and send them to the connector framework.	"Polling for events" on page 177
`terminate()`	Perform cleanup operations for the connector shut down.	"Shutting down the connector" on page 195

Beginning execution of the connector

When the connector is started, the connector framework instantiates the associated connector class and then calls the connector class methods in Table 52..

Table 52. Beginning execution of the connector

Initialization task		For more information
1.	Initialize the connector's application-specific component to perform any necessary initialization for the connector, such as opening a connection to the application.	"Initializing the connector" on page 146
2.	For each business object that the connector supports, obtain the business object handler.	"Obtaining the C++ business object handler" on page 148

Initializing the connector

To begin connector initialization, the connector framework calls the initialization method, `init()`, in the connector class, (derived from `GenGlobals`). This method performs initialization steps for the connector's application-specific component.

Important: As part of the implementation of your connector class, you *must* implement an `init()` method for your connector.

As discussed in "Initializing the connector" on page 64,, the main tasks of the `init()` initialization method include:

- "Establishing a connection"
- "Checking the connector version" on page 147
- "Recovering In-Progress events" on page 147

In addition to the above topics, this section provides an example C++ `init()` method in "Example C++ initialization method" on page 147.

Important: During execution of the initialization method, business object definitions and the connector framework's subscription list are *not* yet available.

Establishing a connection

The main task of the `init()` method is to establish a connection to the application. It returns "success" if the connector succeeds in opening a connection. If the connector *cannot* open a connection, the `init()` method must return the appropriate failure status to indicate the cause of the failure. In a C++ connector, typical return codes used in `init()` are `BON_SUCCESS`, `BON_FAIL`, and `BON_UNABLETOLOGIN`. For information on these and other return codes, see "Handling errors and status" on page 195.

Note: For an overview of the steps in an initialization method, see "Establishing a connection" on page 65..

Checking the connector version

The `getVersion()` method returns the version of the connector's application-specific component.

Note: For a general description of `getVersion()`, see "Checking the connector version" on page 65..

In the C++ connector library, the `getVersion()` method is defined in the `GenGlobals` class. It returns a pointer to a character string indicating the version of the connector. When returning a version string for the connector, you can choose to return the default version as defined by the connector framework. To do this, return the string `CX_CONNECTOR_VERSIONSTRING`, as shown in Figure 54 below.

```
char * ExampleGenGlob::getVersion()
{
   return (char *) CX_CONNECTOR_VERSIONSTRING;
}
```

Figure 54. A getVersion() method for a C++ connector

Alternatively, you can override the default version by setting a version string in a file named `ConnectorVersion.h`. Include the following lines, or something similar, in `ConnectorVersion.h`.

```
#ifndef CX_CONNECTOR_VERSIONSTRING
#define CX_CONNECTOR_VERSIONSTRING    "3.0.0"
#endif
```

Recovering In-Progress events

To recover In-Progress events, a C++ connector must use whatever technique the application provides to access event records in the event store. For example, the connector could use the ODBC API and ODBC SQL commands. Once the In-Progress event record has been retrieved, the connector can take one of the actions in Table 23 to handle the In-Progress event.

Note: For a general discussion of how to recover In-Progress events, see "Recovering In-Progress events" on page 65..

Example C++ initialization method

For a C++ connector, the `init()` method provides the initialization for the connector's application-specific component. This method returns an integer that indicates the status of the initialization operation. Figure 55 shows an example of the initialization method for a C++ connector. In this example, the `init()` method calls `GenGlobals::getConfigProp()` to retrieve the properties of the connector configuration from the repository. The connector's application-specific component uses the configuration values to log on to the application and perform any other required initialization tasks.

```
int ExampleGenGlob::init(CxVersion *version)
{
   char val[512];
   char val1[512];

   int return_code = BON_SUCCESS;

   // get values for connector configuration properties
   getConfigProp("ConnectorName", val, 512);
   //Log trace message "Initializing {ConnectorName}"
   traceWrite(Tracing::LEVEL5, generateMsg(20050,
            CxMsgFormat::XRD_INFO, NULL, 1, val), NULL);

   . . .
   getConfigProp("Hostname",val1, 512);

   // use configuration values to log in to the application
   // If log in fails, log error message.
   logMsg(generateMsg(21000,CxMsgFormat::XRD_ERROR, NULL, 1, name));

   return return_code;
}
```

Figure 55. Initializing a C++ connector

Obtaining the C++ business object handler

In a C++ connector, the business-object-handler base class is `BOHandlerCPP`. To obtain an instance of a business object handler for a supported business object, the connector framework calls the `getBOHandlerforBO()` method, which is defined as part of the `GenGlobals` class.

Note: For general information about the `getBOHandlerforBO()` method, see "Obtaining the business object handler" on page 66.. For a general discussion of how to design business object handlers, see "Designing business object handlers" on page 73..

The number of business object handlers that the connector framework obtains through its calls to the `getBOHandlerforBO()` method depends on the overall design for business object handling in your connector:

- If the connector is metadata-driven, it can be designed to use a generic, metadata-driven business object handler.

  Figure 56 contains an implementation of the `getBOHandlerforBO()` method that returns a metadata-driven business object handler. It calls the constructor for the business-object-handler class (derived from the `GenBOHandler` class), which instantiates a single business-object-handler class that handles *all* the business objects supported by the connector.
- If some or all application-specific business objects require special processing, then you must set up multiple business object handlers for those objects.

  Figure 57 contains an implementation of the `getBOHandlerforBO()` method that creates unique business object handlers for `Invoice` and `Item` business objects, and creates a generic business object handler for all other business objects.

Important: During execution of the `getBOHandlerforBO()` method, the business-object class methods are not yet available.

Figure 56 calls the constructor for the GenBOHandler class to instantiate a single business object handler class that handles *all* the business objects supported by the connector.

```
BOHandlerCPP *ExampleGenGlob::getBOHandlerforBO(char * BOName)
{
   //   If a business object handler does not exist,
   //   create one; otherwise, return the existing pointer
   if(pHandlr == NULL) {
      pHandlr = new GenBOHandler();
   }
   return pHandlr;
}
```

Figure 56. A getBOHandlerforBO() method for a generic business object handler

Figure 57 calls the constructor for the appropriate business object handler, based on the business object name that is passed in:

- If the name of the business object is "App_Invoice", call the constructor for the Invoice_Handler business object handler class.
- If the name of the business object is "App_Item", call the constructor for the Item_Handler business object handler class.
- If the name of the business object is some other string, call the constructor for the Generic_Handler business object handler class.

```
BOHandlerCPP *ExampleGenGlob::getBOHandlerforBO(char * BOName)
{
   if (strcmp(BOName, App_Invoice)==0)
      return new Invoice_Handler();
   else if (strcmp(BOName, App_Item)==0)
      return new Item_Handler();
   else
      return new Generic_Handler();
   }
}
```

Figure 57. A getBOHandlerforBO() method for multiple business object handlers

Creating a business object handler

Creating a business object handler involves the following steps:

- "Extending the C++ business-object-handler base class"
- Implementing a Business-Object-Handler Retrieval Method—For more information, see "Obtaining the C++ business object handler" on page 148.
- "Implementing the doVerbFor() method" on page 150

Note: For an introduction to request processing, see "Request processing" on page 25.. For a discussion of request processing and the implementation of doVerbFor(), see Chapter 4, "Request processing," on page 73.

Extending the C++ business-object-handler base class

In the C++ connector library, the base class for a business object handler is named BOHandlerCPP. The BOHandlerCPP class provides methods for defining and accessing a business object handler. To implement your own business object handler, you extend this business-object-handler base class to create your own *business-object-handler class*.

Note: For general information about the methods of the business-object-handler base class, see "Extending the business-object-handler base class" on page 76..

To derive a business-object-handler class for a C++ connector, follow these steps:

1. Create a class that extends the `BOHandlerCPP` class. Name this class:

   connectorName`BOHandler.cpp`

   where connectorName uniquely identifies the application or technology with which the connector communicates. For example, to create a business object handler for a Baan application, you can create a business-object-handler class called `BaanBOHandler`. If your connector design implements multiple business object handlers, include the name of the handled business objects in the name of the business-object-handler class.
2. Implement the virtual method, `doVerbFor()`, to define the behavior of the business object handler. Other methods in the `BOHandlerCPP` class are already implemented. For more information on how to implement this virtual method, see "Implementing the doVerbFor() method" on page 150.

   Note: The other methods in the `BOHandlerCPP` class have their implementations provided. The `doVerbFor()` method is the *only* virtual method in this class. For more information, see Chapter 11, "BOHandlerCPP class," on page 231.

You might need to implement more than one business object handler for your connector, depending on the application and its API. For a discussion of some issues to consider when implementing business object handlers, see "Designing business object handlers" on page 73..

Implementing the doVerbFor() method

The `doVerbFor()` method provides the functionality for the business object handler. When the connector framework receives a request business object, it calls the `doVerbFor()` method in the appropriate business object handler to perform the action of this business object's verb. For a C++ connector, the `BOHandlerCPP` class defines the `doVerbFor()` virtual method. You must provide an implementation of this virtual method as part of your business-object-handler class.

Note: For a general description of the role of the `doVerbFor()` method, see "Handling the request" on page 76.. Figure 25 on page 77 provides the method's basic logic.

The role of the business object handler is to perform the following tasks:

1. Receive business objects from the connector framework
2. Process each business object based on the active verb
3. Send requests for operations to the application.
4. Return status to the connector framework.

Table 53 summarizes the steps in the basic logic for the verb processing that the `doVerbFor()` method typically performs. Each of the sections listed in the For More Information column provides more detailed information on the associated step in the basic logic.

Table 53. Basic logic of the doVerbFor() method

Business-object-handler step		For more information
1.	Obtain the active verb from the request business object.	"Obtaining the active verb" on page 151
2.	Verify that the connector still has a valid connection to the application.	"Verifying the connection before processing the verb" on page 152
3.	Branch on the value of the valid active verb.	"Branching on the active verb" on page 153
4.	For a given active verb, perform the appropriate request processing:	
	• Perform verb-specific tasks.	"Performing the verb operation" on page 156
	• Process the business object.	"Processing business objects" on page 157
5.	Send the appropriate status to the connector framework.	"Sending the verb-processing response" on page 168

In addition to the processing steps in Table 53,, this section also provides additional processing information in "Additional processing issues" on page 173.

Obtaining the active verb

To determine which actions to take, the `doVerbFor()` method must first retrieve the verb from the business object that it receives as an argument. This incoming business object is called the *request business object*. The verb that this business object contains is the *active verb*, which must be one of the verbs that the business object definition supports. Table 54 lists the method that the C++ connector library provides to retrieve the active verb from the request business object.

Table 54. Method for obtaining the active verb

C++ connector library class	Method
`BusinessObject`	`getVerb()`

Obtaining the active verb from the request business object generally involves the following steps:

1. Verify that the request business object is valid.

   Before the connector calls `getVerb()`, it should verify that the incoming request business object is *not* `null`. The incoming business object is passed into the `doVerbFor()` method as a `BusinessObject` object.
2. Obtain the active verb with the `getVerb()` method.

   Once the request business object is valid, you can use the `getVerb()` method in the `BusinessObject` class to obtain the active verb from this business object.
3. Verify that the active verb is valid.

   When the connector has obtained the active verb, it should verify that this verb is neither null nor empty.

If *either* the request business object or the active verb is invalid, the connector should *not* continue with verb processing. Instead, it should take the steps outlined in Table 55..

Table 55. Handling a verb-processing error

Error-handling step		Method or code to use
1.	Log an error message to the log destination to indicate the cause of the verb-processing error.	`BOHandlerCPP.logMsg()`, `BOHandlerCPP.generateAndLogMsg()`

Table 55. Handling a verb-processing error (continued)

Error-handling step		Method or code to use
2.	Set a message within the return-status descriptor to indicate the cause of the verb-processing failure.	`ReturnStatusDescriptor.seterrMsg()`
3.	Return a `BON_FAIL` outcome status from the `doVerbFor()` method.	`return BON_FAIL;`

Figure 58 contains a fragment of the `doVerbFor()` method that obtains the active verb with the `getVerb()` method. This code ensures that the request business object and its active verb are not null. If either of these conditions exists, the code fragment stores a message in the return-status descriptor and exits with an outcome status of `BON_FAIL`.

```
int ExampleBOHandler::doVerbFor(BusinessObject &theObj,
      ReturnStatusDescriptor *rtnStatusMsg)
{
   int status = BON_SUCCESS;

   //make sure that the incoming business object is not null
   if (theObj == null) {
      generateAndLogMsg(1100, CxMsgFormat::XRD_ERROR, NULL, 0, NULL);

      char errorMsg[512];
      sprintf(errorMsg,
         "doVerbFor: Invalid request business object .");
      rtnStatusMsg->seterrorMsg(errorMsg);
      status = BON_FAIL;
   }

   // obtain the active verb
   char *verb = theObj.getVerb();

   // make sure the active verb is neither null nor empty
   if (verb == null || strcmp(verb, "")){
      generateAndLogMsg(6548, CxMsgFormat::XRD_ERROR, NULL, 0, NULL);
      sprintf(errorMsg,"doVerbfor: Invalid active verb.");
      rtnStatusMsg->seterrorMsg(errorMsg);
      status = BON_FAIL;

   // perform verb processing here
   ...
}
```

Figure 58. Obtaining the active verb

Verifying the connection before processing the verb

When the `init()` method in the connector class initializes the application-specific component, one of its most common tasks is to establish a connection to the application. The verb processing that `doVerbFor()` performs requires access to the application. Therefore, before the `doVerbFor()` method begins processing the verb, it should verify that the connector is still connected to the application. The way to perform this verification is application-specific. Consult your application documentation for more information.

A good design practice is to code the connector application-specific component so that it shuts down whenever the connection to the application is lost. If the connection has been lost, the connector should *not* continue with verb processing. Instead, it should take the steps outlined in Table 56 to notify the connector framework of the lost connection.

Table 56. Handling a lost connection

Error-handling step		Method or code to use
1.	Log an error message to the log destination to indicate the cause of the verb-processing error. The connector logs a fatal error message so that email notification is triggered if the `LogAtInterchangeEnd` connector configuration property is set to True.	`BOHandlerCPP.logMsg()`, `BOHandlerCPP.generateAndLogMsg()`
2.	Set a message within the return-status descriptor to indicate the cause of the lost connection.	`ReturnStatusDescriptor.seterrMsg()`
3.	Return the `BON_APPRESPONSETIMEOUT` outcome status from the `doVerbFor()` method.	`return BON_APPRESPONSETIMEOUT;`

Note: This return-status descriptor object is part of the verb-processing response that `doVerbFor()` sends to the connector framework. For information on these methods, see Chapter 18, "ReturnStatusDescriptor class," on page 299.

After the connector returns `BON_APPRESPONSETIMEOUT` to inform the connector controller that the application is not responding, it stops the process in which the connector runs. A system administrator must fix the problem with the application and restart the connector to continue processing events and business object requests.

Figure 59 contains code to handle the loss of connection to the application. In this example, error message 20018 is issued to inform an administrator that the connection from the connector to the application has been lost and that action needs to be taken.

```
int ExampleBOHandler::doVerbFor(BusinessObject &theObj,
      ReturnStatusDescriptor *rtn)
{
   ...
   if (//application is not responding ) {
      // Lost connection to the application
      // Log an error message
      logMsg(generateMsg(20018, CxMsgFormat::XRD_FATAL, NULL, 0,
         "MyConnector"));

      // Populate a ReturnStatusDescriptor object
      char errorMsg[512];
      sprintf(errorMsg, "Lost connection to application");
      rtnObj->seterrMsg(errorMsg);

      return BON_APPRESPONSETIMEOUT;   }
   ....
   // if connection is open, continue processing
   ...
}
```

Figure 59. Example of loss of connection in doVerbFor()

Branching on the active verb

The main task of verb processing is to ensure that the application performs the operation associated with the active verb. The action to take on the active verb depends on whether the `doVerbFor()` method has been designed as a basic method or a metadata-driven method:

- "Basic verb processing" on page 154
- "Metadata-driven verb processing" on page 155

Basic verb processing: For verb-processing that is *not* metadata-driven, you branch on the value of the active verb to perform the verb-specific processing. Your `doVerbFor()` method must handle *all* verbs that the business object supports.

Note: As part of the verb-branching logic, make sure you include a test for an invalid verb. If the request business object's active verb is *not* supported by the business object definition, the business object handler must take the appropriate recovery actions to indicate an error in verb processing. For a list of steps to handle a verb-processing error, see Table 55 on page 151..

Figure 60 shows a basic `doVerbFor()` method that handles create, update, retrieve, and delete operations. This code that branches off the active verb's value for the Create, Update, Retrieve, and Delete verbs. For each verb your business object supports, you must provide a branch in this code. It then calls the corresponding verb method to continue the business object processing.

At the top of this code fragment, this C++ `doVerbFor()` method defines special constants to identify the different verbs. Use of these verb constants make it easier to identify the active verbs in the code as well as to change their string representations. If your connector handles additional verbs, IBM recommends that you define `String` constants as part of your extended `BOHandlerCPP` class.

```
#define CREATE "Create"
#define UPDATE "Update"
#define RETRIEVE "Retrieve"
#define DELETE "Delete"

int ExampleBOHandler::doVerbFor(BusinessObject &theObj,
      ReturnStatusDescriptor *rtnStatusMsg)
{
   int status = BON_SUCCESS;

   // Determine the verb of the incoming business object
   char *verb = theObj.getVerb();

   if (strcmp(verb, CREATE) == 0)
      status = doCreate(theObj);
   else if (strcmp(verb, UPDATE) == 0)
      status = doUpdate(theObj);
   else if (strcmp(verb, RETRIEVE) == 0)
      status = doRetrieve(theObj);
   else if (strcmp(verb, DELETE) == 0)
      status = doDelete(theObj);
   else
   {
      // This verb is not supported.
      // Send the collaboration a message to that effect
      // in the ReturnStatusDescriptor object.
      char errorMsg[512];
      sprintf(errorMsg,"doVerbFor: verb '%s' is not supported ",
         verb);
      rtnStatusMsg->setErrorMsg(errorMsg);
      status = BON_FAIL;
   }

   // Return status to connector framework
   return status;
}
```

Figure 60. Branching on the active verb's value

The code fragment in Figure 60 is modularized; that is, it puts the actual processing of each supported verb into a separate *verb method*: `doCreate()`, `doUpdate()`, `doRetrieve()`, and `doDelete()`. Each verb method should meet the following minimal guidelines:

- Define a `BusinessObject` parameter, so the verb method can receive the request business object, and possibly send this updated business object back to the calling method.
- Return an outcome status, which `doVerbFor()` can then return to the connector framework.

This modular structure greatly simplifies the readability and maintainability of the `doVerbFor()` method.

Metadata-driven verb processing: For metadata-driven verb-processing method, the application-specific information for the verb contains metadata, which provides processing instructions for the request business object when that particular verb is active. Table 57 lists the method that the C++ connector library provides to obtain application-specific information for the verb of a business object.

Table 57. Method for retrieving the verb's application-specific information

C++ connector library class	Method
`BusObjSpec`	getVerbAppText()

The verb application-specific information can contain the name of the method to call to process the request business object for that particular verb. In this case, the `doVerbFor()` method does *not* need to branch off the value of the active verb because the processing information resides in the verb's application-specific information.

Figure 61 shows a forms-based, metadata-driven `doVerbFor()` method that implements all verb processing for a business object. Using the business object application-specific information, the method identifies a form name and loops through the business object attributes to retrieve attribute descriptions. Each attribute description is an instance of the `BOAttrType` class. Through this class, the method can obtain attribute application-specific information and other information about the attribute, such as whether it is a key.

Note: For more information on how to process business objects, see "Processing business objects" on page 157.

The method retrieves the attribute values from the business object instance, and fills in the form using the attribute metadata to identify the fields of the form for each attribute. The method identifies the verb operation in the business object, retrieves the verb metadata to get any processing instructions, and sends the complete form to the application. If this is a Create operation and the application creates new data, such as keys, the method retrieves the data from the application and processes it.

```
int ExampleBOHandler::doVerbFor(BusinessObject &theObj,
      ReturnStatusDescriptor *rtnObj)
{
   BusObjSpec                  *theSpec;
   int               status = BON_SUCCESS;

   // Get the business object definition and its metadata:
   // the name of the form. Open the specified form
   theSpec = theObj.getSpecFor();
   form = OpenForm(theObj.getAppText());

   // For each attribute, retrieve the attribute description,
   // get the attribute values and application-specific information,
   // and set the field of the form
   for (int i = 0; i < theObj.getAttrCount; i++) {
      BOAttrType * curAttr = theObj.getAttrDesc(i);
      Form.setfield(curAttr->getAppText(),theObj.getAttrValue(i));
   }

   // Get the verb and the verb metadata: the type of operation
   // to perform. Tell the application to do the operation
   Form.doOperation(theSpec->getVerbAppText(theObj.getVerb()));

   // Process returned attributes if any
   for (int k=0; k < theObj.getAttrCount() -1; k++) {
      BOAttrType * curAttr = theObj.getAttrDesc(k);
      value = Form.getField(curAttr->getAppText();
      theObj.setAttrValue(k, value);
   }
   return status;
}
```

Figure 61. Metadata-driven verb processing

Note: Another use of verb application-specific information can be to specify the application's API method to call to update the application entity for the particular verb.

Performing the verb operation

Table 58 lists the standard verbs that a `doVerbFor()` method can implement, as well as an overview of how each verb operation processes the request business object. For more information on processing business objects, see "Processing business objects" on page 157.

Table 58. Performing the verb operation

Verb	Use of request business object	For more information
Create	• Use any application-specific information in the business object definition to determine in which application structure to create the entity (for example, a database table). • Use any application-specific information for each attribute to determine in which application substructure to add the attribute values (for example, a database column). • Use attribute values as values to save in new application entity. If the application generates key values for the new entity, save the new key values in the request business object, which should then be included as part of the verb-processing response.	"Handling the Create verb" on page 80

Table 58. Performing the verb operation (continued)

Verb	Use of request business object	For more information
Retrieve	• Use any application-specific information in the business object definition to determine from which application structure (for example, a database table) to retrieve the entity. • Use attribute key value (or values) to identify which application entity to retrieve. If the application finds the requested entity, save its values in the request business object's attributes. The request business object should then be included as part of the verb-processing response.	"Handling the Retrieve verb" on page 83
Update	• Use any application-specific information of the business object definition to determine in which application structure (for example, a database table) to update the entity. • Use any application-specific information for each attribute to determine which application substructure to update with the attribute values (for example, a database column). • Use attribute key value (or values) to identity which application entity to update. • Use the attribute values as values to update the existing application entity. If the application is designed to create an entity if the one specified for update does not exist, save the new entity values in the request business object's attributes. The request business object should then be included as part of the verb-processing response.	"Handling the Update verb" on page 91
Delete	• Use any application-specific information in the business object definition to determine from which application structure (for example, a database table) to delete the entity. • Use attribute key value (or values) to identify which application entity to delete. The request business object should then be included as part of the verb-processing response so that InterChange Server can perform any required cleanup of relationship tables.	"Handling the Delete verb" on page 98

Processing business objects

Most verb operations involve obtaining information from the request business object. This section provides information about the steps your `doVerbFor()` method needs to take to process the request business object.

Note: These steps assume that your connector has been designed to be metadata-driven; that is, they describe how to extract application-specific information from the business object definition and attributes to obtain the location within the application associated with each attribute. If your connector is *not* metadata-driven, you probably do not need to perform any steps that extract application-specific information.

Table 59 summarizes the steps in the basic program logic for deconstructing a request business object that contains metadata.

Table 59. Basic logic for processing a request business object with metadata

Step		For more information
1.	Obtain the business object definition for the request business object.	"Accessing the business object definition" on page 158

Table 59. Basic logic for processing a request business object with metadata (continued)

Step		For more information
2.	Obtain the application-specific information in the business object definition to obtain the application structure to access.	"Extracting business object application-specific information" on page 159
3.	Obtain the attribute information.	"Accessing the attributes" on page 160
4.	For each attribute, get the attribute application-specific information in the business object definition to obtain the application substructure to access.	"Extracting attribute application-specific information" on page 162
5.	Make sure that processing occurs only for those attributes that are appropriate.	"Determining whether to process an attribute" on page 163
6.	Obtain the value of each attribute whose value needs to be sent to the application entity.	"Extracting attribute values from a business object" on page 165
7.	Notify the application to perform the appropriate verb operation.	"Initiating the application operation" on page 167
8.	Save any attribute values in the request business object that are required for the verb-processing response.	"Saving attribute values in a business object" on page 168

The section walks through the basic logic of an example Create method, explaining in detail how it works. This example verb method uses the basic program logic in Table 59 to deconstruct a business object and build an ODBC SQL command. To see the complete verb method, go to "Example: Create method for a flat business object" on page 171.

Accessing the business object definition: For a C++ connector, the `doVerbFor()` method receives the request business object as an instance of the `BusinessObject` class. However, to begin verb processing, the `doVerbFor()` method often needs information from the business object definition, which is an instance of the `BusObjSpec` class. Therefore, the first step in a typical C++ verb operation is to retrieve the pointer to the business object definition for the request business object.

Table 60 lists the method that the C++ connector library provides to obtain the business object definition for the current business object (a `BusinessObject` instance).

Table 60. Method for obtaining a business object definition

C++ connector library class	Method
`BusinessObject`	getSpecFor()

Suppose a verb method called `doSimpleCreate()` implements processing for the Create verb for a table-based application. Figure 62 shows one way to call `getSpecFor()` to obtain the business object definition (`theSpec`) for the request business object (`theObj`).

```
int doSimpleCreate(BusinessObject &theObj)
{
   ...
   BusObjSpec *theSpec = theObj.getSpecFor()
```

Figure 62. Obtaining the business object definition

Note: At connector startup, the connector instantiates `BusObjSpec` instances for *all* business object definitions that the connector supports. The `getSpecFor()` method returns a pointer to the instance of the business object definition associated with the request business object.

Once `getSpecFor()` obtains a reference to the `BusObjSpec` instance, the `doVerbFor()` method can use methods of the `BusObjSpec` class to obtain information from the business object definition, such as its application-specific information and access to the attribute descriptors. The business object definition includes the information shown in Table 61.. For a complete list of `BusObjSpec` methods, see Chapter 14, "BusObjSpec class," on page 265.

Table 61. Methods for obtaining information from the business object definition

Business object definition information	BusObjSpec method
The name of the business object definition	getName()
A verb list—contains the verbs that the business object supports.	isVerbSupported()
A list of attributes—for each attribute, the `BusObjSpec` object provides:	getAttributeCount()
• position in the list of attributes	getAttributeIndex()
• the attribute descriptors (`BOAttrType` instances) for each attribute; for more information, see "Accessing the attributes" on page 160.	getAttribute()
Application-specific information:	
• business object definition	getAppText()
• verb	getVerbAppText()
Note: Access to application-specific information for an attribute is provided in the `BOAttrType` class.	

A business object handler typically uses the business object definition to get information on the business object's attributes or to get the application-specific information from the business object definition, attribute, or verb.

Extracting business object application-specific information: Business objects for metadata-driven connectors are often designed to have application-specific information that provides information about the application structure. For such connectors, a typical verb operation must retrieve the application-specific information from the business object definition associated with the request business object. Table 62 lists the method that the C++ connector library provides to retrieve application-specific information from the business object definition.

Table 62. Method for obtaining business object application-specific information

C++ connector library class	Method
`BusObjSpec`	getAppText()

Note: The method to obtain application-specific information, shown in Table 62,, uses deprecated terminology in its method name. This method name refers to "application-specific text". The more current name for "application-specific text" is "application-specific information".

As Table 62 shows, the connector uses the `getAppText()` method to obtain the application-specific information for the business object definition.

```
char * appInfo = theSpec->getAppText());
```

The `getAppText()` method retrieves a character string containing the application-specific information from the business object definition. Using the example business object shown in Figure 38 on page 101,, the preceding line of code copies the table name `customer` into the variable `appText`.

For the `doSimpleCreate()` method in Figure 62,, the verb method implements processing for the Create verb for a a table-based application. For such an application, the business objects have usually been designed to have application-specific information provide the verb operations with information about the application structure (For more information, see Table 36 on page 103). The application-specific information in a business object definition can contain the name of the database table associated with the business object.

The verb method first accesses application-specific information through the business object definition. Therefore, the verb method calls `BusObjSpec::getAppText()` to obtain the name of the database table to access. The connector can then use the retrieved table name to begin building the SQL statement that accesses the application database. For a Create operation, the SQL statement is INSERT.

Using the example Customer business object shown, the code fragment in connector Figure 63 constructs an INSERT statement that adds a new row to an application database table named `customer`. At this point in the execution of the verb method, this SQL statement is:

```
INSERT INTO customer
```

```
int doSimpleCreate(BusinessObject &theObj)
{
   char                          table_name[64];
   char                          insertStatement[1024];
   BusObjSpec                     *theSpec;

   // Retrieve pointer to the business object definition
   theSpec = theObj.getSpecFor();

// Retrieve the table name from the AppSpecificInfo property
// for the business object definition
strcpy(table_name, theSpec->getAppText());

// Begin building the SQL INSERT statement
sprintf(insertStatement, "INSERT INTO %s (", table_name);
...
}
```

Figure 63. Obtaining the name of the database table

Accessing the attributes: For a C++ connector, the `doVerbFor()` method receives the request business object as an instance of the `BusinessObject` class. However, if the verb operation needs to obtain information about attribute properties, it needs to access an *attribute descriptor*, which is an instance of the `BOAttrType` class. Therefore, a typical C++ verb operation must retrieve a pointer to each attribute descriptor that it needs to access in the request business object.

Table 63 lists the methods that the C++ connector library provides to obtain the attribute descriptors from the current business object.

Table 63. Classes and methods for obtaining an attribute descriptor

C++ connector library class	Method
BusObjSpec	getAttribute(),, getAttributeCount(),, getAttributeIndex()
BusinessObject	getAttrDesc(),, getAttrCount()

To access an attribute descriptor, the connector can use either of the following methods:

- The `getAttribute()` method of the `BusObjSpec` class obtains an attribute descriptor from a business object definition.
- The `getAttrDesc()` method of the `BusinessObject` class obtains an attribute descriptor from a business object.

The `getAttribute()` and `getAttrDesc()` methods can access an attribute descriptor in one of two ways:

- Its attribute name—you can identify the attribute by its Name property to obtain its attribute descriptor:

  ```
  theAttr = theSpec->getAttribute(attrName);
  ```

- Its integer index—to obtain the attribute index (its ordinal position), you can either:
  - Obtain a count of all attributes in the business object definition with `getAttributeCount()` and loop through them one at a time, passing each index value to `getAttribute()` to get an attribute descriptor.

    Note: Alternatively, you can obtain an attribute count from the business object itself with the `BusinessObject::getAttrCount()` method.
  - Obtain the index for a particular attribute. You can obtain the index for an attribute by specifying its name to `getAttributeIndex()`

  The following call to `getAttribute()` returns a pointer to the `BOAttrType` instance that represents the attribute at the specified ordinal position of the business object definition, indicated by the variable `i`:

  ```
  for (i = 0; i < theSpec->getAttributeCount(), i++) {
     theAttr = theSpec->getAttribute(i);
  // do processing on the attribute descriptor
  }
  ```

Once the attribute descriptor exists, the connector can use methods of the `BOAttrType` class to obtain information about the properties of the associated attribute, such as its cardinality or maximum length. Table 64 lists the methods that the C++ connector library provides to retrieve information from an attribute descriptor. For a complete list of methods in the `BOAttrType` class, see Chapter 10, "BOAttrType class," on page 219.

Table 64. Methods for obtaining information about attribute properties

Attribute property	BusObjAttr method
Name	getName(),, hasName()
Type	getRelationType(),, getTypeName(),, getTypeNum(),, hasTypeName(),, isObjectType(),, isType()
Key	isKey()
Foreign key	isForeignKey()
Max Length	getMaxLength()
Required	isRequired()

Table 64. Methods for obtaining information about attribute properties (continued)

Attribute property	BusObjAttr method
Cardinality	getCardinality(),, hasCardinality(),, isMultipleCard()
Default Value	getDefault()
Attribute application-specific information	getAppText()

Important: The attribute value is *not* available from within the attribute descriptor (`BOAttrType` instance) in the business object definition. You must access an attribute value through the attribute in the `BusinessObject` instance. For more information, see "Extracting attribute values from a business object" on page 165.

Extracting attribute application-specific information: If business objects for metadata-driven connectors are designed to have application-specific information that provides information about the application structure, the next step after extracting the application-specific information from the business object definition is to extract the application-specific information from each attribute in the request business object. Table 65 lists the method that the C++ connector library provides to retrieve application-specific information from an attribute descriptor.

Table 65. Method for obtaining attribute application-specific information

C++ connector library class	Method
`BOAttrType`	getAppText()

Note: The method to obtain application-specific information, shown in Table 65,, uses deprecated terminology in its method name. This method name refers to "application-specific text". The more current name for "application-specific text" is "application-specific information".

The connector uses the `getAppText()` method to obtain the application-specific information for an attribute. If business objects have been designed to have application-specific information provide information for a table-based application, the application-specific information for the attribute contains the name of the application table's column associated with this attribute (For more information, see Table 36 on page 103).

Using the example business object shown in Figure 38 on page 101,, the code fragment in connector Figure 63 begins construction of an INSERT statement that adds a new row to an application database table named `customer`. After extracting the application-specific information from the business object definition, the next step is to determine what columns in the application table will be updated by the business object request. The connector can then use the retrieved column name to continue building the SQL statement. It would append each column name to the column list of the INSERT statement that adds a new row to an application database table named `customer`. After all attributes are processed, this SQL statement is:

```
INSERT INTO customer (cust_key, cust_name, cust_status, cust_region)
```

For a C++ connector, the verb method calls `BusinessObject::getAttrCount()` on the current business object to determine the number of attributes in a business object. To obtain the application-specific information for each attribute involves the following steps:

1. The verb method then traverses the business object definition and calls `BusObjSpec::getAttribute()` to retrieve each attribute descriptor.

   The `getAttribute()` method returns a pointer to an instance of the `BOAttrType` class. Each `BOAttrType` instance represents a single attribute descriptor for an attribute in a business object definition.
2. Through the attribute descriptor, the connector can retrieve information about attribute properties, such as whether the attribute is a key or foreign key.

   For each attribute, the method extracts the application-specific information for the attribute from the attribute descriptor with the `getAppText()` method, which is defined in the `BOAttrType` class.

In Figure 64,, the verb method copies the column name for each attribute into the `column` variable as it traverses through the business object.

```
for (i = 0; i < theObj.getAttrCount() - 1; i++)
   strcpy(column, theSpec->getAttribute(i)->getAppText());
```

Figure 64. Obtaining attribute application-specific information

The `for` loop in Figure 64 performs the following tasks:

- Loop index is initialized to zero.

  In this example, the destination application uses the same key value that was generated by the source application. This key value is simply passed to the destination application in the business object. If the destination application generates its own keys, the business object typically does not contain values for keys, and the key attribute might be set to the special Ignore value.

  If the Create verb method processes the first attribute, which contains the key, the loop index variable starts at 0. However, if your application generates keys, your Create verb method will *not* process attributes containing keys. In this case, the loop index variable starts at a value other than 0.
- Loop index increments until it reaches the total number of attributes in the business object definition.

  The `getAttrCount()` method returns the total number of attributes in the business object definition. However, this total includes the `ObjectEventId` attribute. Because the `ObjectEventId` attribute is used by the IBM WebSphere business integration system and is *not* present in application tables, a verb method does not need to process this attribute. Therefore, when looping through business object attributes, you loop from zero to one less than the total number of attributes:

  ```
  getAttrCount() - 1
  ```
- Loop index increments by one.

  This increment of the index obtains the next attribute descriptor when `getAttribute(i)` is called.

Determining whether to process an attribute: Up to this point, the verb processing has been using the application-specific information to obtain the application location for each attribute of the request business object. Once it has this location information, the `doVerbFor()` method can begin processing the attribute.

As the verb operation loops through the business object attributes, you might want to confirm that the method processes only certain attributes. Table 66 lists some of the methods that the C++ connector library provides to determine whether an

attribute should be processed.

Table 66. Classes and methods for determining attribute processing

Attribute test	C++ connector library class and method
An attribute is a simple attribute and *not* an attribute that represents a contained business object.	`BOAttrType` isObjectType()
The value of the attribute in the business object instance is *not* the special value of Blank (a zero-length string) or Ignore (a null pointer).	`BusinessObject` getAttrValue(),, isIgnoreValue(),, isIgnore(),, isBlankValue(),, isBlank()
The attribute is *not* a place-holder attribute. Place-holder attributes are used in business object definitions to separate attributes that contain child business objects.	`BOAttrType` getAppText()

Using the methods in Table 66,, a verb operation can determine that an attribute is one that the operation wants to process:

- Is the attribute simple or complex?

  The `BOAttrType::isObjectType()` method checks that the attribute value does *not* represent a contained business object. For more information on how to handle an attribute that *does* contain a business object, see "Accessing child business objects" on page 175.
- Is the attribute a place-holder attribute or the `ObjectEventId` attribute?

  You can use the `getAppText()` method to determine if the attribute in the business object definition has application-specific information. Because neither of these special types of attributes represent columns in an application entity, there is no need for the business object definition to include application-specific information for them.
- Is the attribute set to a value other than the special Blank or Ignore values?

  The verb operation can compare the attribute's value to the Ignore and Blank values with the `isIgnoreValue()` and `isBlankValue()` methods, respectively. For more information on the Ignore and Blank values, see "Handling the Blank and Ignore values" on page 173.

The code fragment in Figure 65 shows how the sample Create verb method determines that an attribute is one that the method wants to process. The method first gets the attribute value from the current business object by calling `BusinessObject::getAttrValue()`. It then performs the tests to determine if the attribute should be processed.

```
for (i = 0; i< theObj.getAttrCount()-1; i++) {
   theAttr = theSpec->getAttribute(i);
   theAttrVal = theObj.getAttrValue(i);

   if (!theAttr->isObjectType() && strlen(theAttr->getAppText()) > 0)
   {
      // Use only columns that contain a valid value
      if (!(theObj.isIgnoreValue((char *)theAttrVal)) &&
            !(theObj.isBlankValue((char *)theAttrVal))) {

         // Get the column name from the AppSpecificInfo text
         strcpy(column, theSpec->getAttribute(i)->getAppText());
   }
}
```

Figure 65. Determining whether to process an attribute

For single attributes that have application-specific information and that contain values that are *not* Ignore or Blank, the connector retrieves the column name from the attribute application-specific information in the business object definition and appends the names to the SQL statement.

Extracting attribute values from a business object: Once the verb operation has confirmed that the attribute is ready for processing, it usually needs to extract the attribute value:

- For a Create or Update verb, the verb operation needs the attribute value to send it to the application, where it can be added to the appropriate application entity. For an Update verb, the verb operation also needs the attribute value from any key attribute that holds search information. The application uses this search information to locate the entity to update.

  Note: If the Create or Update operation sends information back to the connector, the verb operation needs to store the returned information as values in the appropriate attributes. For more information, see "Saving attribute values in a business object" on page 168.
- For a Retrieve, RetrieveByContent, or Exist verb, the verb operation needs the attribute value from any key attribute (Retrieve or Exist) or non-key attribute (RetrieveByContent) that holds search information. The application uses this search information to retrieve the entity.

  Note: For a Retrieve or RetrieveByContent, the verb operation also needs to set the attribute value for any attribute associated with retrieved data. For more information, see "Saving attribute values in a business object" on page 168.
- For a Delete verb, the verb operation needs the attribute value from any key attribute that holds search information. The application uses this search information to locate the entity to delete.

The value of an attribute is part of the attribute information in the business object (`BusinessObject` instance). Table 67 lists the methods that the C++ connector library provides to obtain attribute values from a business object.

Table 67. Methods for obtaining attribute values

C++ connector library class	Method
`BusinessObject`	getAttrCount(),, getAttrValue()

As in the business object definition, each attribute in the business object can be accessed in one of two ways:

- Its attribute name—you can obtain the attribute name if you know its ordinal position with the `getAttrName()` method.
- Its integer index—to obtain the attribute index (its ordinal position), you can obtain a count of all attributes in the business object definition with `getAttrCount()` and loop through them one at a time, passing each index value to `getAttrValue()` to get an attribute value.

As Table 67 shows, the `BusinessObject` class provides a single method for obtaining attribute values of all valid data types, `getAttrValue()`. Because the type of an attribute in a business object definition can be any supported type, the return value of `getAttrValue()` is defined as a `void` pointer. You should check the type of the attribute in the business object definition, and based on the attribute type, cast the `void` pointer to a character pointer, a business object pointer, or a business object array before you assign the returned value to a variable.

Note: Attribute values that are neither business objects nor business object arrays are stored as pointers to character strings in the C++ connector library. If the value of an attribute is *not* a business object or business object array, you need to cast the `void` pointer to a character pointer.

After identifying the attributes to process, the `doSimpleCreate()` method (see Figure 62,, Figure 63,, and Figure 65) must obtain the data values to insert into columns in the application table. As the method processes each attribute, it adds the attribute value to the SQL statement. To create the list of attribute values, the verb method traverses the attributes of the business object definition a second time. For each attribute, it obtains the attribute value from the business object instance. In this second traversal of the business object, the verb method again checks the type and value of each attribute to determine whether it wants to process the attribute.

Note: Although this connector traverses a business object twice to construct a database query, if you are using an application API to set values in the application, the API might not need to loop through the business object in this way.

The connector can then use the retrieved column value to continue building the SQL statement. Using the example business object shown in Figure 38 on page 101,, the connector would append each column value to the VALUES clause of the INSERT statement that adds a new row to an application database table named `customer`.

Suppose the `doSimpleCreate()` method processed a sample `Customer` business object with the following data:

`CustomerId`	`87975`
`CustomerName`	`Trievers Inc.`
`CustomerStatus`	`3`
`CustomerRegion`	`NE`

After all attributes are processed, this SQL statement might be:

```
INSERT INTO customer (cust_key, cust_name, cust_status, cust_region)
   VALUES (87975, 'Trievers Inc.', 3, 'NE')
```

For a C++ connector, the verb method calls `BusinessObject::getAttrValue()` to retrieve the value of each attribute from the business object instance. The verb method casts returned attribute values to character pointers to generate the VALUES clause of the INSERT statement. Figure 66 shows a code fragment of the Create verb method that accesses the attribute values and appends them to the VALUES clause of the INSERT statement.

```
for (i = 0; i< theObj.getAttrCount()-1; i++) {
   theAttr = theSpec->getAttribute(i);
   theAttrVal = theObj.getAttrValue(i);

   // Process simple attributes
   if (!theAttr->isObjectType() && strlen (theAttr->getAppText()) > 0)
   {

      // Use columns that contain a valid value in
      // the business object
      if (!(theObj.isIgnoreValue((char *)theAttrVal)) &&
            !(theObj.isBlankValue((char *)theAttrVal))) {

         // Set the quote character for attributes that
         // are STRING type
         quote_str[0] = (theObj.getAttrType(i) ==
                     BOAttrType::STRING) ? '\'' : ' ';

         // Build the value and add it to insertStatement
         sprintf(clause, "%s %s%s%s", firstLoop ? " " : ",",
                  quote_str, (char *)theAttrVal, quote_str);
         strcat(insertStatement, clause);
         firstLoop = 0;
      }
   }
}
```

Figure 66. Accessing the attribute values in a C++ connector

Initiating the application operation: Once the verb operation has obtained the information it needs from the request business object, it is ready to send the application-specific command so that the application performs the appropriate operation. The command must be appropriate for the verb of the request business object. For a table-based application, this command might be an SQL statement or a ODBC call. Consult your application documentation for more information.

Important: Your `doVerbFor()` method must ensure that the application operation completes successfully. If this operation is unsuccessful, the `doVerbFor()` method must return the appropriate outcome status (such as `BON_FAIL`) to the connector framework. For more information, see "Sending the verb-processing response" on page 168.

When the `doSimpleCreate()` method has built the SQL statement, it is ready to execute it. When the INSERT statement is executed, the application creates a new row in the `customer` database table. To execute the SQL statement, you must use the application API that provides table access. The `doSimpleCreate()` verb method uses the standard ODBC API to execute the SQL statement. If your application has an API that executes SQL statements, use the application API. The code fragment in Figure 67 finishes the SQL statement and executes it using an ODBC call.

```
// Finish the INSERT statement
strcat(insertStatement, ")");

// Allocate an ODBC statement
rc = SQLAllocStmt(hdbc, &hstmt);
// Execute the SQL statement
rc = SQLExecDirect(hstmt, (unsigned char *)insertStatement, SQL_NTS);
// Free the ODBC statement handle
```

Figure 67. Executing the INSERT statement in a C++ connector

Saving attribute values in a business object: Once the application operation has completed successfully, the verb operation might need to save new attribute values retrieved from the application into the request business object:

- For a Create verb, the verb operation needs to save the new key values if the application has generated them as part of its Create operation.
- For an Update verb, the verb operation needs to save *all* attribute values, including any generated key values (if the application has been designed to create a new entity when it does not find the specified entity to update).
- For a Retrieve or RetrieveByContent, the verb operation needs to save the attribute value for any attributes retrieved.

Table 68 lists the methods that the C++ connector library provides to save attribute values in a business object.

Table 68. Methods for saving attribute values

C++ connector library class	Method
`BusinessObject`	getAttrCount(),, setAttrValue()

An attribute in the business object can be accessed by its name or its index (its ordinal position). You can obtain a count of all attributes in the business object definition with `getAttrCount()` and loop through them one at a time, passing each index value to `setAttrValue()` to get an attribute value.

As Table 68 shows, the `BusinessObject` class provides a single method for saving attribute values of all valid data types: `setAttrValue()`. Because the type of an attribute in a business object definition can be any supported type, the parameter value of `setAttrValue()` is defined as a `void` pointer.

Sending the verb-processing response

The C++ connector must send a verb-processing response to the connector framework, which in turn sends a response to the integration broker. This verb-processing response includes the following information:

- The integer return code of `doVerbFor()`
- A message in the return-status descriptor, if there is an information, warning, or error return message
- A response business object

The following sections provide additional information about how a C++ connector provides each of the pieces of response information. For general information about the connector response, see “Indicating the connector response” on page 107..

Returning the outcome status: The doVerbFor() method provides an integer outcome status as its return code. As Table 69 shows, the C++ connector library provides constants for the outcome-status values that doVerbFor() is mostly likely to return.

Important: The doVerbFor() method *must* return an integer outcome status to the connector framework.

Table 69. Outcome-status values for a C++ doVerbFor()

Condition in doVerbFor()	C++ outcome status
The verb operation succeeded.	BON_SUCCESS
The verb operation failed.	BON_FAIL
The application is not responding.	BON_APPRESPONSETIMEOUT
At least one value in the business object changed.	BON_VALCHANGE
The requested operation found multiple records for the same key value.	BON_VALDUPES
The connector finds multiple matching records when retrieving using non-key values. The connector will only return the first matching record in a business object.	BON_MULTIPLE_HITS
The connector was not able to find matches for Retrieve by non-key values.	BON_FAIL_RETRIEVE_BY_CONTENT
The requested business object entity does not exist in the database.	BON_BO_DOES_NOT_EXIST

Note: The C++ connector library provides additional outcome-status constants for use by other connector methods. For a complete list of outcome-status constants, see Table 69 on page 169..

The outcome status that doVerbFor() returns depends on the particular active verb it is processing. Table 70 lists the tables in this manual that provide possible return values for the different verbs.

Table 70. Return values for different verbs

Verb	For more information
Create	Table 28 on page 82
Retrieve	Table 29 on page 89
RetrieveByContent	Table 30 on page 90
Update	Table 32 on page 97
Delete	Table 34 on page 99
Exist	Table 35 on page 100

Using the outcome status that doVerbFor() returns, the connector framework determines its next action:

- If the outcome status is BON_APPRESPONSETIMEOUT, the connector framework shuts down the connector. For more information, see "Verifying the connection before processing the verb" on page 152.
- For all other outcome-status values, the connector framework continues in its present state. It includes the outcome status in its response to the integration broker. For some outcome-status values, the connector framework also includes a response business object. For more information, see "Updating the request business object" on page 170.

Populating the return-status descriptor: The return-status descriptor is a structure that holds additional information about the state of the doVerbFor() method when this method exits. The connector framework passes in an empty return-status

descriptor as an argument to doVerbFor(). The doVerbFor() method can update this return-status descriptor with a message. This updated return-status descriptor is then accessible by the connector framework when doVerbFor() exits. The connector framework then includes the return-status descriptor in the response it sends to the integration broker.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector framework returns the response to the connector controller, which routes it to the collaboration.

For a C++ connector, the return-status descriptor is a ReturnStatusDescriptor object. Table 71 lists the status information that this structure provides.

Table 71. Information in the return-status descriptor

Return-status descriptor information	Description	C++ accessor method
Error message	A string to provide a description of the error condition	getErrorMsg(),, seterrMsg()
Status value	An integer status value to detail the cause of the error condition	getStatus(),, setStatus()

The return-status descriptor is filled in one of two ways:

- Explicitly—when the verb processing in the doVerbFor() method is successful, the method can set values in this descriptor before it completes execution.
- Implicitly—when the connector framework copies the outcome status value into the status field of the return-status descriptor.

For example code that fills the return-status descriptor before exiting, see Figure 58 on page 152..

Updating the request business object: The connector framework passes in the request business object as an argument to doVerbFor(). The doVerbFor() method can update this business object with attribute values. This updated business object is then accessible by the connector framework when doVerbFor() exits.

The connector framework uses the outcome status to determine whether to return a business object as part of the connector's response, as follows:

- If the connector framework receives one of the following outcome-status values, it includes the request business object as part of its response:
  - BON_VALCHANGE
  - BON_MULTIPLE_HITS

  If your doVerbFor() method returns one of these outcome-status values, make sure it updates the request business object with response information.
- For any other outcome-status value, the connector framework does *not* include the request business object in its response.

Important: The value that the doVerbFor() method returns affects what the connector framework sends to InterChange Server. If the value is BON_VALCHANGE or BON_MULTIPLE_HITS, the connector framework returns a changed business object. You must ensure that the request business object is updated as appropriate for the returned outcome status.

Example: Create method for a flat business object

The C++ code sample in Figure 68 shows a Create method that uses the Open Database Connectivity (ODBC) API to insert a new record in an application database. The ODBC interface is a standard API for accessing a variety of database systems.

This code sample illustrates the basic logic of extracting information from a business object. It shows how the connector can use the metadata in the business object definition and the content of the business object instance to build a SQL INSERT statement.

The connector first calls `BusinessObject::getSpecFor()` to retrieve a pointer to the business object definition for the business object instance passed in as an argument to the Create method. Using `BusObjSpec::getAppText()`, the connector retrieves the name of the application table from the business object definition application-specific information and begins building the SQL statement. For each attribute in the business object instance that is a value other than the special Blank or Ignore value, the connector retrieves the column name from the attribute application-specific information in the business object definition and appends it to the SQL statement.

The connector then calls `BusinessObject::getAttrValue()` to retrieve the value of each attribute from the business object instance. When the SQL INSERT statement is complete, the method calls the ODBC API `SQLExecDirect()` to submit the statement. Typically, a Create method gets keys for new entities from an application, returns the keys to InterChange Server in a business object, and returns `BON_VALCHANGE`. However, because this method sets the key to the value in the source application, it simply returns `BON_SUCCESS`.

```
int doSimpleCreate(BusinessObject &theObj)
{
   char                          table_name[64];
   char                          column[64];
   char                          columnList[256];
   char                          clause[256];
   char                          insertStatement[1024];
   char                          quote_str[2] = " ";
   int                           firstLoop = 1;
   int                           j;
   BusObjSpec                     *theSpec;
   void                          *theAttrVal;
   BOAttrType                    *theAttr;
   RETCODE                       rc; /* return code for ODBC functions */
   HSTMT                         hstmt; /* pointer to ODBC statement handle */

   // Retrieve pointer to the business object definition
   theSpec = theObj.getSpecFor();

   // Retrieve the table name from the AppSpecificInfo property
   // for the business object definition
   strcpy(table_name, theSpec->getAppText());
   // Begin building the SQL INSERT statement
   sprintf(insertStatement, "INSERT INTO %s (", table_name);

   // Build the list of column names for the INSERT statement
   // For each attribute, extract the column name from the
   // attribute AppSpecificInfo property
   for (j = 0; j < theObj.getAttrCount()-1; j++) {
      theAttr = theSpec->getAttribute(j);
      theAttrVal = theObj.getAttrValue(j);

      // Process non-child objects only
      if (!theAttr->isObjectType() &&
               strlen (theAttr->getAppText()) > 0) {
         // Use only columns that contain a valid value
         // in the Business Object
         if (!(theObj.isIgnoreValue((char *)theAttrVal)) &&
               !(theObj.isBlankValue((char *)theAttrVal))) {
            // Get the column name from the AppSpecificInfo text
            strcpy(column,
               theSpec->getAttribute(j)->getAppText());

            sprintf(columnList, "%s %s", firstLoop ? " " : ",",
               column);
            strcat(insertStatement, columnList);

            firstLoop = 0;
         }
      }
   }

   // Add the VALUES SQL keyword
   sprintf(clause, ") VALUES (");
   strcat(insertStatement, clause);

   // Build the values to be inserted
   // For each attribute, extract the value from the business object
   firstLoop = 1;
   for (j = 0; j < theObj.getAttrCount()-1; j++) {
      theAttr = theSpec->getAttribute(j);
      theAttrVal = theObj.getAttrValue(j);
```

Figure 68. Example Create method (Part 1 of 2)

```
      // Process non-child objects only
      if (!theAttr->isObjectType() &&
            strlen (theAttr->getAppText()) > 0) {

         // Use columns that contain a valid value in
         // the business object
         if (!(theObj.isIgnoreValue((char *)theAttrVal)) &&
                  !(theObj.isBlankValue((char *)theAttrVal))) {

            // Set the quote character if this is a STRING attribute
            quote_str[0] = (theObj.getAttrType(j) ==
               BOAttrType::STRING) ? '\'' : ' ';

            // Build the value and add it to insertStatement
            sprintf(clause, "%s %s%s%s", firstLoop ? " " : ",",
               quote_str, (char *)theAttrVal, quote_str);
            strcat(insertStatement, clause);
            firstLoop = 0;
         }
      }
   }

   // Finish the INSERT statement
   strcat(insertStatement, ")");
   // Allocate an ODBC statement
   rc = SQLAllocStmt(hdbc, &hstmt);
   // Execute the SQL statement
   rc = SQLExecDirect(hstmt, (unsigned char *)insertStatement,
      SQL_NTS);
   // Free the ODBC statement handle
   SQLFreeStmt(hstmt, SQL_DROP);

   return BON_SUCCESS;
}
```

Figure 68. Example Create method (Part 2 of 2)

Note: The code sample in Figure 68 shows a general approach to metadata-driven connector design. However, much of the example is specific to an ODBC-based connector. The ODBC (Open Database Connectivity) API was used because it is a standard API for accessing a database. If your application provides an API that allows a connector to modify application data, it is best to use the application API. When using an application API, the implementation of verb operations might differ from the implementation shown in this example.

Additional processing issues

This section provides the following additional information about how to process the request business object:

- "Handling the Blank and Ignore values"
- "Accessing child business objects" on page 175

Handling the Blank and Ignore values: In addition to a regular attribute value, simple attributes in business objects can have either of the special values shown in Table 72..

Table 72. Special attribute values for simple attributes

Special attribute value	Represents
Blank	A zero-length string value
Ignore	A value that the connector should ignore

WebSphere InterChange Server

Important: If your business integration system uses InterChange Server, in the third-party maps, the string `CxIgnore` represents an Ignore value, and the string `CxBlank` represents a Blank value. These strings should be used *only* in maps. They should *not* be stored in business objects as attribute values because they are reserved keywords in the InterChange Server system.

The connector can call C++ connector library methods to determine whether a business object attribute is set to a special value:

- Blank—to process attributes with the Blank value, a connector can use any of the methods shown in Table 73..

Table 73. Methods for determining if an attribute contains the Blank value

BOAttrType method	Description
`isBlankValue(`*`value`*`)`	Determines whether a specified attribute value is equal to the Blank value
`isBlank(`*`attributeName`*`)isBlank(`*`position`*`)`	Determines whether a specified attribute contains the Blank value.

When an attribute contains the Blank value, the `doVerbFor()` method should process the attributes as Table 75 shows.

- Ignore— to process attributes with the Ignore value, a connector can use any of the methods shown in Table 74..

Table 74. Methods for determining if an attribute contains the Ignore value

BOAttrType method	Description
`isIgnoreValue(`*`value`*`)`	Determines whether a specified attribute value is equal to the Ignore value
`isIgnore(`*`attributeName`*`) isIgnore(`*`position`*`)`	Determines whether a specified attribute contains the Ignore value.

When attributes are set to the Ignore value, the connector should process the attributes shown in Table 76..

Table 75. Processing actions for the Blank Value

Verb	Processing action for Blank value
Create	Create the entity with an appropriate blank value for the attributes. The blank value might be configurable, or it might be specific to the application.
Update	Update the entity fields to “empty” for those attributes that are set to the Blank value.
Retrieve	If the attribute is a key or the connector is doing a retrieve by non-key values, retrieve an entity where this attribute is a zero-length string.
Delete	If the attribute is a key, delete an entity where this field is set to the Blank value.

Table 76. Processing actions for the Ignore value

Verb	Processing action for Ignore Value
Create	If the attribute is not a key, do not set a value in the application for the attribute. For key attributes, if the application generates keys, the key attributes might be set to the Ignore value. In this case, create the entity, retrieve the application-generated keys, and return the keys to the integration broker. Note that if the application does not generate key values, then all key attributes are expected to have valid values.
Update	If the attribute is not a key, do not set a value in the application for the attribute.
Retrieve	Do not match for Retrieve operations based on an attribute set to Ignore.
Delete	Do not match for Delete operations based on an attribute set to Ignore.

When a connector creates a new business object, all attribute values are set to Ignore internally. A connector must set appropriate values for attributes, since all unset attribute values remain defined as Ignore. To set attribute values to the special Ignore or Blank values, you use the `setAttrValue()` method (defined in the `BusinessObject` class), passing it a special attribute-value constant, as follows:

Blank constant	`BusinessObject::BlankValue`
Ignore constant	`BusinessObject::IgnoreValue`

For example, the following C++ code fragment sets all non-key attributes to the Ignore value.

```
for (i = 0; i < theObj.getAttrCount()-1; i++)
   {
   if (!theAttr->isKey())
      {
      attrname = theObj.getAttrName(i);
      theObj.setAttrValue(attrname, BusinessObject::IgnoreValue,
         theObj.getAttrType(i));
      }
   }
```

As another example, the C++ code fragment below retrieves application data and sets business object attributes that have `NULL` values in the application database to the Blank attribute value.

```
// Fetch application data into appdata variable
// Process record
for (i = 0; i < theObj.getAttrCount()-1; i++)
   {
   if (!theSpec->getAttribute(i)->isObjectType())
      {
      if (strlen(appdata)==0)
         sprintf(attrValue, theObj.getBlankValue());
      theObj.setAttrValue(i, (void*)attrValue,theObj.getAttrType(i));
      }
   }
```

Accessing child business objects: As discussed in "Processing hierarchical business objects" on page 103,, a C++ connector uses the methods of the C++ connector library shown in Table 77 to access a child object.

Table 77. Classes and methods for accessing child business objects

C++ connector library class	Method
BOAttrType	isObjectType(),, isMultipleCard() OBJECT attribute-type constant
BusinessObject	getAttrValue()
BusObjContainer	getObjectCount(),, getObject()

The verb processing in the doVerbFor() method uses the isObjectType() method to determine if the attribute contains a business object (its attribute type is set to the OBJECT attribute-type constant). When doVerbFor() finds an attribute that is a business object, the method checks the cardinality of the attribute using isMultipleCard(). Based on the results of isMultipleCard(), the method takes one of the following actions:

- If the attribute has single cardinality, the verb operation can perform the requested operation on the single child business object.
- If an attribute has multiple cardinality, the verb operation must first access the business object array using the getAttrValue() method, which returns values as follows:
  - If the attribute is a simple attribute, getAttrValue() returns a void pointer to the value.
  - If the attribute is a business object, getAttrValue() returns a void pointer to the business object.
  - If the attribute is an array of business objects, getAttrValue() returns a void pointer to the BusObjContainer object containing the array.

The return value of getAttrValue() must be cast to the correct type to use the data. For example, to access the contents of a business object array, the return value must be cast to the BusObjContainer type, as shown in this code fragment:

```
theAttr = theSpec->getAttribute(i);
   if (theAttr->isObjectType()) {
      if(theAttr->isMultipleCard()) {
         // Multiple cardinality object so cast attribute value
         // to a BusObjContainer object
         BusObjContainer *busObjContnr =
            (BusObjContainer *) theObj.getAttrValue(i);
```

If the attribute contains a business object array, the doVerbFor() method obtains access to this array through the casted BusObjContainer object that getAttrValue() has returned.

Note: The deprecated name for an array of business objects is a "business object container". This term is also used to name the connector library class that provides methods for accessing the child business objects in a business object array (BusObjContainer). You can think of this class as providing methods for handling an array of business objects.

To access individual business objects within the business object array, take the following steps:

1. Call BusObjContainer::getObjectCount() to get the number of child business objects in the array.
2. As it iterates through the business object array, the verb processing can get each individual child object within the business object array using the BusObjContainer::getObject(*index*) method, where *index* is the array element

index. This method returns a pointer to a child business object or NULL if there is no business object at the specified position.

Figure 69 shows the C++ code to access child business objects.

```
for (int i=0; i < busObjContnr->getObjectCount(); i++) {
   BusinessObject *currBusObj = busObjContnr->getObject(i);
   status = do<i>Verb</i>Method(*currBusObj);
}
```

Figure 69. Accessing child business objects in a C++ connector

Figure 70 shows a C++ submethod, do*Verb*Method(), that might be called by a main verb method to process child objects. For a business object such as the one shown in Figure 43 on page 105,, a Create method might first create the application entity for the parent Customer business object, and then call the submethod to traverse the parent business object to find attributes referring to contained business objects.

```
int GenBOHandler::doChildCreate(BusinessObject &theObj)
{
   int      i, k;
   int       status = BON_SUCCESS;

   for (i = 0; i < theObj.getAttrCount() -1; i++) {
      theAttr = theSpec->getAttribute(i);
      theAttrVal = theObj.getAttrValue(i);
      if (theAttr->isObjectType()) {
         if(theAttr->isMultipleCard()) {
            // Multiple cardinality object so cast attribute value
            // to a BusObjContainer object
            BusObjContainer *Cont = (BusObjContainer *) theAttrVal;
            if (theAttrVal != NULL) {
               for (k=0; k < Cont->getObjectCount(); k++) {
                  BusinessObject *curObj = Cont->getObject(k);
                  status = doCreate(*curObj);
                  if (status == BON_FAIL)
                     return status;
               }
            }
         }
         else {
            // Single cardinality object
            if (theAttrVal != NULL) {
               status = doCreate(*(BusinessObject *)theAttrVal);
               if (status == BON_FAIL)
                  return status;
            }
         }
      }
   }

   return status;
}
```

Figure 70. Processing child business objects in a C++ submethod

Polling for events

For a C++ connector, the GenGlobals class defines the pollForEvents() method. You must provide an implementation of this virtual method as part of your connector class.

Note: For an introduction to event notification, see "Event notification" on page 22.. For a discussion of event-notification mechanisms and the implementation of `pollForEvents()`, see Chapter 5, "Event notification," on page 109.

The C++-based pseudo-code in Figure 71 shows the basic logic flow for a `pollForEvents()` method. This method first retrieves a pointer to the subscription manager. The subscription manager manages subscriptions to business objects supported by the connector.

The `pollForEvents()` method then retrieves a set of events from the event store and, for each event, the method calls the subscription manager class method `isSubscribed()` to determine whether any subscriptions exist for the corresponding business object. If there are subscriptions, the method retrieves the data from the application, creates a new business object, and calls the subscription manager method `gotApplEvent()` to send the business object to InterChange Server. If there are no subscriptions, the method archives the event record with a status value of unprocessed.

```
int ExampleGenGlob::pollForEvents()
{
   SubscriptionHandlerCPP *mySubHandler =
      GenGlobals::getTheSubHandler();
   int status = 0;
   get the events from the event list
   for events 0 to PollQuantity in eventlist {
      extract BOName, verb, and key from the event record
      if(mySubHandler->isSubscribed(BOName,BOverb) {
         BO = new BusinessObject(BOName)
         BO.setAttrValue(key)
         retrieve application data using doVerbFor()
         BO.setVerb(Retrieve)
         BO.doVerbFor()
         BO.setVerb(BOverb)
         status = mySubHandler->gotApplEvent(BusinessObject);
         archive event record with success or failure status
      }
      else {
         archive event record with unprocessed status      }
   return status;
}
```

Figure 71. C++ pollForEvents() example

Note: For a flow chart of the poll method's basic logic, see Figure 52 on page 122..

This section provides more detailed information on each of the steps in the basic logic for the event processing that the `pollForEvents()` method typically performs. Table 78 summarizes these basic steps.

Table 78. Basic logic of the pollForEvents() method

Step		For more information
1.	Set up a subscription manager for the connector.	"Accessing the subscription manager" on page 179
2.	Verify that the connector still has a valid connection to the event store.	"Verifying the connection before accessing the event store" on page 179

Table 78. Basic logic of the pollForEvents() method (continued)

Step		For more information
3.	Retrieve specified number of event records from the event store and store them in an events array. Cycle through the events array. For each event, mark the event in the event store as In-Progress and begin processing.	"Retrieving event records" on page 180
4.	Get the business object name, verb, and key data from the event record.	"Getting the business object name, verb, and key" on page 182
5.	Check for subscriptions to the event.	"Checking for subscriptions to the event" on page 183
	If the event has subscribers:	
	• Retrieve application data and create the business object.	"Retrieving application data" on page 184
	• Send the business object to the connector framework for event delivery.	"Sending the business object to the connector framework" on page 186
	• Complete event processing.	"Completing the processing of an event" on page 188
	If the event does *not* have subscribers, update the event status to Unsubscribed.	"Checking for subscriptions to the event" on page 183
6.	Archive the event.	"Archiving the event" on page 189

Accessing the subscription manager

As part of connector initialization, the connector framework instantiates a subscription manager. This subscription manager keeps the subscription list current. (For more information, see "Business object subscription and publishing" on page 13.) A C++ connector has access to the subscription manager and the connector subscription list through a *subscription handler*, which is encapsulated by the `SubscriptionHandlerCPP` class. It can use methods of this class to determine whether business objects have subscribers and to send business objects to the connector controller.

Table 79 lists the method that the C++ connector library provides to obtain a reference to a subscription handler.

Table 79. Method for obtaining a subscription handler

C++ connector library class	Method
`GenGlobals`	getTheSubHandler()

For a C++ connector, the `pollForEvents()` method first sets up a subscription manager for the connector by calling the `GenGlobals::getTheSubHandler()`. For example:

```
SubscriptionHandlerCPP *mySub = GenGlobals::getTheSubHandler();
```

The `getTheSubHandler()` method returns a pointer to the subscription manager, which is an instance of the `SubscriptionHandlerCPP` class. Through this subscription handler, the connector can query its subscription manager to find out whether the integration broker is interested in a particular type of business object.

Verifying the connection before accessing the event store

When the `init()` method in the connector class initializes the application-specific component, one of its most common tasks is to establish a connection to the application. The poll method requires access to the event store. Therefore, before

the pollForEvents() method begins processing events, it should verify that the connector is still connected to the application. The way to perform this verification is application-specific. Consult your application documentation for more information.

A good design practice is to code the connector application-specific component so that it shuts down whenever the connection to the application is lost. If the connection has been lost, the connector should *not* continue with event polling. Instead, it should take the steps in Table 56 on page 153 to notify the connector framework of the lost connection. The pollForEvents() method should return BON_APPRESPONSETIMEOUT to notify the connector framework of the loss of connection to the application.

Figure 72 contains code to handle the loss of connection to the application. In this example, error message 20018 is issued to inform an administrator that the connection from the connector to the application has been lost and that action needs to be taken.

```
int ExampleGlobals::pollForEvents()
{
   ...

   if (//application is not responding ) {
      // Lost connection to the application
      // Log an error message
      logMsg(generateMsg(20018, CxMsgFormat::XRD_FATAL, NULL, 0,
                     "MyConnector"));

      // Populate a ReturnStatusDescriptor object
      char errorMsg[512];
      sprintf(errorMsg, "Lost connection to application");
      rtnObj->seterrMsg(errorMsg);

      return BON_APPRESPONSETIMEOUT;      }

      ....
      // if connection is open, continue processing
      ...
}
```

Figure 72. Loss of connection in pollForEvents()

Retrieving event records

To send event notifications to the connector framework, the poll method must first retrieve event records from the event store. For a C++ connector, you must use an application-specific interface to retrieve event records from the event store.

The poll method can retrieve one event record at a time and process it or it can retrieve a specified number of event records per poll and cache them to an events array. Processing multiple events per poll can improve performance when the application generates large numbers of events.

The number of events picked up in any polling cycle should be configurable using the connector configuration property PollQuantity. At install time, a system administrator sets the value of PollQuantity to an appropriate number, such as 50. The poll method can use the getConfigProp() to retrieve the value of the PollQuantity property, and then retrieve the specified number of event records and process them in a single poll.

Note: Because many C++ connectors are single-threaded, the connector framework does *not* accept request business objects while the poll method is running. This means that request processing is blocked while the poll method is processing events. Keep this in mind when implementing the processing of multiple events per poll. For more information on single- and multi-threaded C++ connectors, see "Threading issues" on page 125..

The connector should assign the In-Progress status to any event that it has read out of the event store and has started to process. If the connector terminates while processing an event and before updating the event status to indicate that the event was either sent or failed, it will leave an In-Progress event in the event store. For more information on how recover these In-Progress events, see "Recovering In-Progress events" on page 65..

To retrieve event records from the event store, a C++ connector must use whatever technique the application provides. As an example, an implementation for a table-based application might create an event table from which the connector application-specific code retrieves event data. The code fragment in Figure 73 shows how a connector for this kind of implementation can retrieve events from an event table using the ODBC API and ODBC SQL commands. The program initially defines the macro for a SQL SELECT statement that retrieves event records from the event table.

```
#define GET_EVENT_QUERY                                    \
      "SELECT event_id, object_name, object_verb, object_key \
      FROM cw_events WHERE event_status = 0 \
      ORDER BY event_time"
```

This SELECT statement retrieves the event identifier, business object name, verb, and key data from those event records whose status is 0 (Ready-for-Poll). These event records are ordered by their timestamp. For information on retrieving event records by event priority, see "Processing events by event priority" on page 125..

The poll method allocates memory for the data, binds the memory to specific columns of data, and retrieves the data one event record at a time. For the complete example program, see "Example of a basic pollForEvents() method" on page 190..

```
// Allocate a statement handle for the SQL statement
rc = SQLAllocStmt(gHdbc1, &hstmt1);

// Execute the SELECT query. Use the macro GET_EVENT_QUERY
rc = SQLExecDirect(hstmt1, GET_EVENT_QUERY, SQL_NTS);

// Allocate memory for event data
ev_id = new char[80];
obj_name = new char[80];
obj_verb = new char[80];
obj_key = new char[80];

query = new char[255];
key_value = new char[80];

// Bind all results set columns to event variables
rc = SQLBindCol(hstmt1, 1, SQL_C_CHAR, ev_id,    80, &cbValue);
rc = SQLBindCol(hstmt1, 2, SQL_C_CHAR, obj_name, 80, &cbValue);
rc = SQLBindCol(hstmt1, 3, SQL_C_CHAR, obj_verb, 80, &cbValue);
rc = SQLBindCol(hstmt1, 4, SQL_C_CHAR, obj_key,  80, &cbValue);

// Fetch the event data
while (rc == SQL_SUCCESS || rc != SQL_SUCCESS_WITH_INFO) {
      ev_id = '\0';
      obj_name = '\0';
      obj_verb = '\0';
      obj_key = '\0';

      rc = SQLFetch(hstmt1);
      if (rc == SQL_SUCCESS) {
         // Process the record
         if ((cp = strchr(ev_id, ' ')) != NULL)
            *cp = NULL;
         if ((cp = strchr(obj_name, ' ')) != NULL)
            *cp = NULL;
         if ((cp = strchr(obj_verb, ' ')) != NULL)
            *cp = NULL;
         if ((cp = strchr(obj_key, ' ')) != NULL)
            *cp = NULL;
      }
```

Figure 73. Retrieving an event

Note: This example uses the standard ODBC API to retrieve event data from the event table. If your application has an API that provides access to data in the event table, use the application API.

Getting the business object name, verb, and key

Once the connector has retrieved an event, it extracts the event ID, the object key, and the name and verb of the business object from the event record. The connector uses the business object name and verb to determine whether the integration broker is interested in this type of business object. If the business object and its active verb have subscribers, the connector uses the entity key to retrieve the complete set of data.

For a C++ connector, you must use an application-specific interface to obtain this information from the event records in the event store. Figure 73 showed one way that a C++ connector can process event data. In the example, the appropriate event data is retrieved into the connector variables:

Event ID	`ev_id`

Business object name	`obj_name`
Verb	`obj_verb`
Object key	`obj_key`

The connector should send the business object with the same verb that was in the event record.

Checking for subscriptions to the event

To determine whether the integration broker is interested in receiving a particular business object and verb, the poll method calls the `isSubscribed()` method. The `isSubscribed()` method takes the name of the current business object and a verb as arguments. The name of the business object and verb must match the name of the business object and verb in the repository.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the poll method can determine if any collaboration subscribes to the business object with a particular verb. At initialization, the connector framework requests its subscription list from the connector controller at connector initialization. At runtime, the application-specific component can use `isSubscribed()` to query the connector framework to verify that some collaboration subscribes to a particular business object. The application-specific connector component can send the event only if some collaboration is currently subscribed.

Other integration brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the connector framework assumes that the integration broker is interested in *all* the connector's supported business objects. If the poll method uses the `isSubscribed()` method to query the connector framework about subscriptions for a particular business object, the method returns `true` for *every* business object that the connector supports.

Table 80 lists the method that the C++ connector library provides to check for subscriptions to the event.

Table 80. Method for checking subscriptions

C++ connector library class	Method
`SubscriptionHandlerCPP`	isSubscribed()

Based on the value that `isSubscribed()` returns, the poll method should take one of the following actions:

- If there are subscribers for an event, the connector takes the following actions:

Connector action taken	For more information
Retrieve the complete set of business object data from the entity in the application database.	"Retrieving application data" on page 184

Connector action taken	For more information
Send the business object to the connector framework, which routes it to the integration broker.	"Sending the business object to the connector framework" on page 186
Archive the event (if archiving is implemented) in case an integration broker subscribes at a later time.	"Archiving the event" on page 189

- If there are no subscriptions for the event, the connector should take the following actions:
  - Update the status of the event to "Unsubscribed" to indicate that there were no subscribers.
  - Archive the event (if archiving is implemented) in case the integration broker subscribes at a later time. Moving the event record to the archive store prevents the poll method from picking up unsubscribed events. For more information, see "Archiving the event" on page 189.
  - Return "fail" (`BON_FAIL` outcome status for a C++ connector) to indicate there are events pending for which no subscriptions currently exist.

    IBM suggests that the connector return "fail" if no subscriptions exist for the event. However, you can return the outcome status that your design dictates.

No other processing should be done with unsubscribed events. If at a later date, the integration broker subscribes to these events, a system administrator can move the unsubscribed event records from the archive store back to the event store.

As Table 80 shows, the `isSubscribed()` method is provided in the subscription manager, the `SubscriptionHandlerCPP` object. The method returns `1` if there are subscribers, and `0` if there are no subscribers. An example of a C++ connector checking for subscriptions follows:

```
if (mySubHndlr->isSubscribed(obj_name, obj_verb) = TRUE) {
      // handle event
   }
else {
      // archive the event (if archiving is supported)
   }
```

Retrieving application data

If there are subscribers for an event, the poll method must take the following steps:

1. Retrieve the complete set of data for the entity from the application.

   To retrieve the complete set of entity data, the poll method must use name of the entity's key information (which is stored in the event) to locate the entity in the application database. The poll method must retrieve the complete set of application data when the event has the following verbs:

   - Create
   - Update
   - Delete event for an application that supports logical deletes

   For a Delete event from an application that supports physical deletes, the application may have already deleted the entity from the database, and the connector may not be able to retrieve the entity data. For information on delete processing, see "Processing Delete events" on page 126..
2. Package the entity data in a business object.

   Once the populated business object exists, the poll method can publish the business object to subscribers.

Table 80 lists the methods that the C++ connector library provides to retrieve entity data from the application database and populate a business object.

Table 81. Methods for retrieving business object data

C++ Connector Library Class	Method
`BusinessObject`	doVerbFor(),, getAttrCount(),, getAttrType(),, getAttrValue(),, setAttrValue(),, setVerb()

Note: If the event is a delete operation and the application supports physical deletions of data, the data has most likely been deleted from the application, and the connector cannot retrieve the data. In this case, the connector simply creates a business object, sets the key from the object key of the event record, and sends the business object. For more information on handling delete events, see "Processing Delete events" on page 126..

For a C++ connector, the standard way of retrieving application data from within `pollForEvents()` is to use the Retrieve method in the connector's business object handler. With this approach, you do not have to recode data retrieval in the `pollForEvents()` method.

To retrieve application data using the `doVerbFor()` method of the `BusinessObject` class, follow these general steps:

1. Create a new business object instance for the business object that corresponds to the application entity.
2. Call `BusinessObject::setVerb()` to set the verb of the business object to Retrieve.
3. Get the key or keys for the application entity from the event record.
4. Call `BusinessObject::setAttrValue()` to set the key values in the business object.

   The `setAttrValue()` method takes as arguments the name of the attribute, a string representation of the value or pointer to the value of the attribute, and the attribute type.
5. Call `BusinessObject::doVerbFor()` on this business object.

   The `BusinessObject` class implementation of the `doVerbFor()` method calls the business object handler specified in the business object definition to perform the action of the verb. The `doVerbFor()` method operates on the current business object, filling the business object with the current values of the application entity.

The C++ code fragment in Figure 74 illustrates this approach.

```
// Create a new business object
pBusObj = new BusinessObject(obj_name);

// Set verb to Retrieve
pBusObj->setVerb("Retrieve");

// Extract value of key from key:value pair
if ((cp = strchr(obj_key, ':')) != NULL) {
   cp++;
   strcpy(key_value, cp);
   cp--;
   *cp = NULL;
}

// Find the key attribute in the business object and
// set it to the key value
for (i = 0; i < pObj->getAttrCount()-1; i++) {
   if (pBusObj->getSpecFor()->getAttribute(i)->isKey()) {
      pBusObj->setAttrValue(pBusObj->getAttrName(i),key_value,
         pBusObj->getAttrType(pBusObj->getAttrName(i)));
   }
}

// Call the business object handler doVerbFor()
if (pBusObj->doVerbFor() == BON_FAIL) {
   // Log error message if retrieve fails
   retcode = BON_FAIL;
}
```

Figure 74. Retrieving application data

The code fragment in Figure 74 uses `getAttrName()` to get the name of each attribute that is a key. It then calls `getAttrType()` to get the type of the key attribute. The `setAttrValue()` method verifies that the new value has the correct data type before changing the attribute value.

The `ObjectEventId` attribute is used in the IBM WebSphere business integration system to track the flow of business objects through the system. In addition, it is used to keep track of child business objects across requests and responses, as child business objects in a hierarchical business object request might be reordered in a response business object.

Connectors are not required to populate `ObjectEventId` attributes for either a parent business object or its children. If business objects do not have values for `ObjectEventId` attributes, the IBM WebSphere business integration system generates values for them. However, if a connector populates child `ObjectEventIds`, the values must be unique across all other `ObjectEventId` values for that particular business object regardless of level of hierarchy. `ObjectEventId` values can be generated as part of the event notification mechanism. For suggestions on how to generate `ObjectEventId` values, see “Event identifier” on page 111..

Sending the business object to the connector framework

Once the data for the business object has been retrieved, the poll method performs the following tasks:

- “Setting the business object verb” on page 187
- “Sending the business object” on page 187

Table 80 lists the methods that the C++ connector library provides to set the business object verb and send the business object.

Table 82. Classes and methods for setting verb and sending business object

C++ connector library class	Method
`BusinessObject`	setVerb()
`SubscriptionHandlerCPP`	gotApplEvent()

Setting the business object verb

To set the verb in a business object to the verb specified in the event record, the poll method calls the business object method `setVerb()`. The poll method should set the verb to the same verb that was in the event record in the event store.

Note: If the event is a physical delete, use the object keys from the event record to set the keys in the business object, and set the verb to Delete.

For a C++ connector, the populated `BusinessObject` object that the `doVerbFor()` method returns still has a verb of Retrieve. The poll method must set the business object's verb to its original value with the `setVerb()` method, as the following code fragment shows:

```
// Set verb to action as indicated in the event record
pBusObj->setVerb(obj_verb);
```

In this code fragment, `obj_verb` is the verb value that has previously been obtained from the event record, as shown in Figure 73..

Sending the business object

The poll method uses the method `gotApplEvent()` to send the business object to the connector framework. This method takes the following steps:

- Check that the connector is active.
- Check that there are subscriptions for the event.
- Send the business object to the connector framework.

The connector framework does some processing on the event object to serialize the data and ensure that it is persisted properly. It then makes sure the event is sent.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector framework makes sure the event is either sent to the ICS through CORBA IIOP or written to a queue (if you are using queues for event notification). If sending the event to ICS, the connector framework forwards the business object to the connector controller, which in turn performs any mapping required to transform the application-specific business object to a generic business object. The connector controller can then send the generic business object to the appropriate collaboration.

Other integration brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the connector framework makes sure the event is converted to an WebSphere MQ message and written to the appropriate MQ queue.

The poll method should check the return code from `gotApplEvent()` to ensure that returned errors are handled appropriately. For example, until the event delivery is successful, the poll method should *not* remove the event from the event table. Table 83 shows the possible event-status values, based on whether event delivery is successful.

Table 83. Possible event status after event delivery

State of Event Delivery	Event Status
If the event delivery is successful	1 (see Table 40 on page 112)
If the event delivery fails	-2 (see Table 40 on page 112)

The following code fragment shows the call to `gotApplEvent()` for a C++ connector:

```
// Send business object to connector framework
if (( retcode = mySub->gotApplEvent(*pBusObj)) == BON_FAIL) {
   // Log an error message
   // Update event status to "error posting event"
   // Event remains in event table and is not archived
}

// Update event status to "event successfully posted"
```

The `gotApplEvent()` method returns `BON_SUCCESS` if the connector framework successfully delivers the business object, and returns a nonzero value (such as `BON_FAIL`) if the delivery fails.

Note: If this call is *not* successful, the event must be archived.

Completing the processing of an event

The processing of an event is complete when the tasks in Table 84 complete.

Table 84. Steps in processing an event

Processing task	For more information
The poll method has retrieved the application data for the event and created a business object that represents the event.	"Retrieving application data" on page 184
The poll method has sent the business object to the connector framework.	"Sending the business object to the connector framework" on page 186

Note: For hierarchical business objects, the event processing is complete when the poll method has retrieved the application data for the parent business object and all child business objects and sent the complete hierarchical business object to the connector framework. The event notification mechanism must retrieve and send the entire hierarchical business object, not just the parent business object.

The poll method must ensure that the event status correctly reflects the completion of the event processing. Therefore, it must handle *both* of the following conditions:

- "Handling successful event processing" on page 189
- "Handling unsuccessful event processing" on page 189

Handling successful event processing

The processing of an event is successful when the tasks in Table 84 successfully complete. The following steps show how the poll method should finish processing a successful event:

1. Receive a "success" return code from the `gotApplEvent()` method signifying the connector framework's successful delivery of the business object to the messaging system.
2. Copy the event to the archive store. For more information, see "Archiving the event" on page 189.
3. Set the status of the event in the archive store.
4. Delete the event record from the event store.

   Until the event delivery is successful, the poll method should *not* remove the event from the event table.

Note: The order of the steps might be different for different implementations.

Handling unsuccessful event processing

If an error occurs in processing an event, the connector should update the event status to indicate that an error has occurred. Table 85 shows the possible event-status values, based on errors that can occur during event processing.

Table 85. Possible event status after unsuccessful event processing

State of Event Delivery	Event Status
If an error occurs in processing an event	-1 (see Table 40 on page 112)
If the event delivery fails	-2 (see Table 40 on page 112)

For example, if there are no application entities matching the entity key, the event status should be updated to "error processing event". As discussed in "Sending the business object" on page 187, the poll method should check the return code from `gotApplEvent()` to ensure that any errors that are returned are handled appropriately. If the event cannot be successfully delivered, its event status should be updated to "error posting event".

In either case, the event should be left in the event store to be analyzed by a system administrator. When the poll method queries for events, it should exclude events with the error status so that these events are not picked up. Once an event's error condition has been resolved, the system administrator can manually reset the event status so that the event is picked up by the connector on the next poll.

Archiving the event

Archiving an event consists of creating an archive record by moving the event record from the event store to an archive store. To archive event records to the archive store, a C++ connector must use whatever technique the application provides. Usually, the connector uses whatever method it used to access event records in the event store, such as the ODBC API and ODBC SQL commands. Consult your application documentation for more information.

Note: For a general introduction to archiving, see "Archiving events" on page 123..

To archive event records from this event store, the poll method takes the following actions:

1. Ensure that archiving is implemented by checking the value of the appropriate connector configuration property, such as `ArchiveProcessed`. For more information, see “Configuring a connector for archiving” on page 124..
2. Copy the event record from the archive store to the event store.
3. Update the event status of the archive record to reflect the reason for archiving the event.

   Table 86 shows the event-status values that the archive record will usually have.

Table 86. Event-status values in an archive record

Event-status value	Description
1	The event was detected, and the connector created a business object for the event and sent the business object to the connector framework. For more information, see “Handling successful event processing” on page 189.
2	The event was detected, but there were no subscriptions for the event, so the event was not sent to the connector framework and on to the integration broker. For more information, see “Checking for subscriptions to the event” on page 183.
-1	The event was detected, but the connector encountered an error when trying to process the event. The error occurred either in the process of building a business object for the event or in sending the business object to connector framework. For more information, see “Handling unsuccessful event processing” on page 189.

4. Delete the event record from the event store.

After archiving is complete, your poll method should set the appropriate return code:

- If the archiving takes place after an event is successfully delivered, the return code is “success”, indicated with the `BON_SUCCESS` outcome-status constant.
- If archiving is due to some error condition (such as unsubscribed events or an error in processing the event), the poll method might need to return a “fail” status, indicated with the `BON_FAIL` outcome-status constant.

Example of a basic pollForEvents() method

The section provides implementations of the basic logic for the `pollForEvents()` method in a C++ connector. The code sample in Figure 75 demonstrates event handling in a C++ connector that uses the ODBC API to communicate with an application database. In this example, the connector retrieves one event at a time and processes it.

Note: This code assumes that the connector’s `init()` method has initialized the ODBC interface before the `pollForEvents()` method is called.

As a first step in the `pollForEvents()` method, a utility function allocates handles for two ODBC connections to the application database. The connector uses these connection handles to interact with the database.

The `pollForEvents()` method gets a pointer to the subscription manager using `GenGlobals::getTheSubHandler()`. It then executes a SQL SELECT statement using the ODBC `SQLExecDirect()` interface to determine what data is returned. When the SQL statement returns, the method allocates memory for the data, assigns the memory to specific columns of data, and retrieves the data one event record at a time into allocated memory.

For each event, the connector calls the subscription manager method `isSubscribed()` to determine whether there are subscribers to this particular business object and verb. If there are subscribers, the connector builds a new business object, sets the key, sets the business object's verb to Retrieve, and calls the business object's `doVerbFor()` method to retrieve the complete set of data for the application entity. If `doVerbFor()` returns success, the poll method sets the appropriate verb in the business object and sends the business object to InterChange Server using the method `gotApplEvent()`.

Once the business object is sent, the poll method executes a SQL statement to remove the event from the event table. This action causes a delete trigger to store the event in the archive table.

```
// The event table is named cw_events.
#define GET_EVENT_QUERY                                        \
      "SELECT event_id, object_name, object_verb, object_key \
      FROM cw_events WHERE status = 0 AND priority = 1 \
      ORDER BY event_time"

#define REMOVE_EVENT_QUERY                                      \
      "DELETE FROM cw_events WHERE event_id = '%s'";

int ExampleGlobals::pollForEvents()
{
SubscriptionHandlerCPP  *mySub;
BusinessObject          *pObj = NULL;
char                    *ev_id = 0;
char                    *obj_name = 0;
char                    *obj_verb = 0;
char                    *obj_key = 0;
char                    *query = 0;
char                    *key_value = 0;
char                    *cp;
int                     i;
RETCODE                 rc;
HSTMT                   hstmt1 = 0;
HSTMT                   hstmt2 = 0;
SDWORD                  cbValue = SQL_NO_TOTAL;
int                     retcode = BON_SUCCESS;

   // Private function for establishing and checking the two
   // database connections that the poll function needs
   checkOdbcConnections();

   // Set up the subscription manager
   mySub = GenGlobals::getTheSubHandler();

   // Allocate a statement handle for the SQL statement
   rc = SQLAllocStmt(gHdbc1, &hstmt1);
   if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
      // handle odbc errors
      return (BON_APPRESPONSETIMEOUT);
   }

   // Execute the event SQL SELECT query to determine
   // what data will be returned.
   // Use the macro GET_EVENT_QUERY
   rc = SQLExecDirect(hstmt1, GET_EVENT_QUERY, SQL_NTS);
   if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
      // handle odbc errors
      rc = SQLFreeStmt(hstmt1, SQL_CLOSE);
      return (BON_APPRESPONSETIMEOUT);
   }

   // Allocate memory for event table columns
   ev_id = new char[80];
   obj_name = new char[80];
   obj_verb = new char[80];
   obj_key = new char[80];

   query = new char[255];
   key_value = new char[80];
```

Figure 75. Implementation of basic logic for pollForEvents() (Part 1 of 4)

```
// Bind all results set columns to variables
rc = SQLBindCol(hstmt1, 1, SQL_C_CHAR, ev_id,    80, &cbValue);
rc = SQLBindCol(hstmt1, 2, SQL_C_CHAR, obj_name, 80, &cbValue);
rc = SQLBindCol(hstmt1, 3, SQL_C_CHAR, obj_verb, 80, &cbValue);
rc = SQLBindCol(hstmt1, 4, SQL_C_CHAR, obj_key,  80, &cbValue);

// Fetch the results
while (rc == SQL_SUCCESS || rc != SQL_SUCCESS_WITH_INFO) {
   ev_id = '\0';
   obj_name = '\0';
   obj_verb = '\0';
   obj_key = '\0';

   rc = SQLFetch(hstmt1);
   if (rc == SQL_SUCCESS || rc != SQL_SUCCESS_WITH_INFO) {
      // Process the record
      // Trim off rest of string at the first
      // space character found
      if ((cp = strchr(ev_id, ' ')) != NULL)
         *cp = NULL;
      if ((cp = strchr(obj_name, ' ')) != NULL)
         *cp = NULL;
      if ((cp = strchr(obj_verb, ' ')) != NULL)
         *cp = NULL;
       if ((cp = strchr(obj_key, ' ')) != NULL)
         *cp = NULL;

      // Determine whether there are subscribers to the event
      if (mySub->isSubscribed(obj_name, obj_verb) != TRUE) {
         // log message
         // delete event from event table
         // add event to archive table
         continue;
      }

      // Prepare to retrieve data into the business object
      pObj = new BusinessObject(obj_name);
      pObj->setVerb("Retrieve");

      // Get key:value pair
      if ((cp = strchr(obj_key, ':')) != NULL) {
         cp++;
         strcpy(key_value, cp);
         cp--;
         *cp = NULL;
      }

      // Find the first key in the object and set it
      for (i = 0; i < pObj->getAttrCount()-1; i++) {
         if (pObj->getSpecFor()->getAttribute(i)->isKey()) {
               pObj->setAttrValue(pObj->getAttrName(i),
               key_value,
               pObj->getAttrType(pObj->getAttrName(i)));
               break;
         }
      }
```

Figure 75. Implementation of basic logic for pollForEvents() (Part 2 of 4)

```
      // Call the business object handler doVerbFor()
      // to retrieve application data
      if (pObj->doVerbFor() == BON_FAIL) {
         // Log error message if retrieve fails
         // Handle retrieve errors
         retcode = BON_FAIL;
         break;
      }

      // Call gotApplEvent() to send the business object
      // with the info
      pObj->setVerb(obj_verb);
      if ((mySub->gotApplEvent(*pBusObj)) == BON_FAIL) {
            // Log error message
         retcode = BON_FAIL;
         break;
      }

      // Allocate statement handle for the SQL
      // statement on the second connection
      rc = SQLAllocStmt(gHdbc2, &hstmt2);
      if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
         // Handle ODBC errors
         hstmt2 = (HSTMT)0;
         retcode = BON_APPRESPONSETIMEOUT;
         break;
      }

      // Remove the event from the event table
      // This will execute a trigger that will archive
      // the event in the archive table
      // Use the REMOVE_EVENT_QUERY macro.
      sprintf(query, REMOVE_EVENT_QUERY, ev_id);
      rc = SQLExecDirect(hstmt2, query, SQL_NTS);
      if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
         // Handle odbc errors
         retcode = BON_APPRESPONSETIMEOUT;
         break;

      } else {
            // Handle odbc errors
            retcode = BON_APPRESPONSETIMEOUT;
            break;
      }
   }

// Clean up and free resources associated with
// the statement handles
if (hstmt1) {
   rc = SQLFreeStmt(hstmt1, SQL_DROP);
}
if (hstmt2) {
   rc = SQLFreeStmt(hstmt2, SQL_DROP);
}
```

Figure 75. Implementation of basic logic for pollForEvents() (Part 3 of 4)

```
if (ev_id) {
      delete ev_id;
   }
   if (obj_name) {
      delete obj_name;
   }
   if (obj_verb) {
      delete obj_verb;
   }
   if (query) {
      delete query;
   }
   if (key_value) {
      delete key_value;
   }

   return (retcode);
}
```

Figure 75. Implementation of basic logic for pollForEvents() (Part 4 of 4)

Shutting down the connector

In the C++ connector library, the `terminate()` method is defined in the `GenGlobals` class. Typical return codes used in `terminate()` are `BON_SUCCESS` and `BON_FAIL`.

Note: It is important that the `terminate()` method for a C++ connector free allocated memory and close the connection with the application.

Figure 76 shows a sample `terminate()` method for a C++ connector.

```
int ExampleGenGlob::terminate()
{
   // log off application and
   // release memory and other resources
   ...

   traceWrite(Tracing::LEVEL3, "terminate() completed.", 0);
   return BON_SUCCESS;
}
```

Figure 76. C++ terminate() method

Handling errors and status

This section provides the following information about how the methods of the C++ connector library indicate error conditions:

- "C++ return codes"
- "Return-status descriptor" on page 197

Note: You can also use error logging and message logging to handle error conditions and messages in your connector. For more information, see Chapter 6, "Message logging," on page 135.

C++ return codes

In the C++ connector library, the outcome-status constants in the `BusObjStatus.h` file define the C++ return codes. Table 87 lists the C++ outcome-status constants.

Table 87. C++ return codes

Outcome-status constant	Description
BON_SUCCESS	The operation succeeded.
BON_FAIL	The operation failed.
BON_APPRESPONSETIMEOUT	The application is not responding.
BON_BO_DOES_NOT_EXIST	The connector performed a Retrieve operation, but the entity that the business object represents does not exist in the application database.
BON_MULTIPLE_HITS	The connector found multiple matching records when retrieving using non-key values. The connector returns only the first matching record in a business object.
BON_FAIL_RETRIEVE_BY_CONTENT	The connector was not able to find matches for retrieve by non-key values.
BON_UNABLETOLOGIN	The connector is unable to log in to the application.
BON_VALCHANGE	At least one value in a business object has changed.
BON_VALDUPES	There are multiple records in the application database with the same key values.
BON_CONNECTOR_NOT_ACTIVE	The connector is not active; it has been paused.
BON_NO_SUBSCRIPTION_FOUND	No subscriptions exist for the event.

Outcome-status constants are provided for use in user implementations of many of the C++ virtual methods, as Table 88 shows. Although your code can return these values from within any method, some of the return codes were designed with specific uses in mind. For example, VALCHANGE informs an integration broker that the connector is sending a business object with changed values.

Table 88. Outcome-status values for C++ virtual methods

Virtual method	Possible outcome-status codes
init()	BON_SUCCESS, BON_FAIL, BON_UNABLETOLOGIN
doVerbFor()	BON_SUCCESS, BON_FAIL, BON_APPRESPONSETIMEOUT, BON_VALCHANGE, BON_VALDUPES, BON_MULTIPLE_HITS, BON_FAIL_RETRIEVE_BY_CONTENT, BON_BO_DOES_NOT_EXIST
gotApplEvent()	BON_SUCCESS , BON_FAIL, BON_CONNECTOR_NOT_ACTIVE, BON_NO_SUBSCRIPTION_FOUND
pollForEvents()	BON_SUCCESS, BON_FAIL, BON_APPRESPONSETIMEOUT
terminate()	BON_SUCCESS , BON_FAIL

The outcome-status constant that the connector framework receives helps to determine its next action, as follows:

- If the outcome status is BON_APPRESPONSETIMEOUT, the connector framework shuts down the connector.

  When the connector framework receives this outcome status, it copies the BON_APPRESPONSETIMEOUT status into the return-status descriptor and returns this descriptor to inform the connector controller that the application is *not* responding. Once it has sent this return-status descriptor, the connector framework stops the process in which the connector runs. A system administrator must fix the problem with the application and restart the connector to continue processing events and business object requests.
- For *all* other outcome-status values, the connector framework continues execution of the connector.

During request processing, the connector framework copies the outcome status into the status field of the return-status descriptor and includes this descriptor in its response to the integration broker. It continues execution of the connector. For some outcome-status values, the connector framework also includes a response business object in its response. For more information, see "Updating the request business object" on page 170.

Important: The connector framework does *not* stop execution of the connector when it receives the `BON_FAIL` outcome-status constant.

Return-status descriptor

During request processing, the connector framework sends an empty structure, called a return-status descriptor, into the business object handler's `doVerbFor()` method. When `doVerbFor()` completes verb processing (either successfully or otherwise), the connector framework includes the return-status descriptor as part of its response to the integration broker.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the collaboration can access the information in this return-status descriptor to obtain the status of its service call request. Therefore, the `doVerbFor()` can provide status information about verb processing to the collaboration by setting the message and status code within the return-status descriptor.

The connector framework automatically copies the outcome status that `doVerbFor()` returns into the status field of the return-status descriptor. Therefore, the `doVerbFor()` method can set a message in the return-status descriptor but it should *not* set the status, as this status will be overwritten when the connector framework copies the outcome status into this status field. For more information about the return-status descriptor and the `doVerbFor()` method, see "Populating the return-status descriptor" on page 169.

Chapter 8. Adding a connector to the business integration system

To run in the IBM WebSphere business integration system, a connector must be defined in the repository. Pre-defined adapters, which the WebSphere Business Integration Adapters product provides, have predefined connector definitions in the repository. A system administrator need only configure the application and set the connector's configuration properties to run the connector.

For the IBM WebSphere business integration system to be able to access a connector that you have developed, you must take the following steps:

1. Create the connector definition in the repository.
2. If WebSphere MQ will be used for messaging between connector components, add message queues for the connector.
3. Create the connector's initial configuration file.
4. Create the connector's startup script.

This chapter provides information on adding a new connector to the IBM WebSphere business integration system. This chapter includes the following sections:

Naming the connector

This chapter provides suggested naming conventions for the files and directories used in connector development. Naming conventions provide a way to make you connector files more easy to locate and identify. Table 89 summarizes the suggested naming conventions for connector files. Many of these files are based on the *connector name*, which should uniquely identify it within the WebSphere business integration system. This name (*connName*) can identify the application or technology with which the connector communicates.

Table 89. Suggested naming conventions for a connector

Connector file	**Name**
Connector definition	*connName*Connector
Connector directory	*ProductDir*\connectors*connName*
Initial connector configuration file	File name: CN_*connName*.txt Directory name: *ProductDir*\repository*connName*
User-customized connector configuration file	File name: CN_*connName*.txt Directory name: *ProductDir*\connectors*connName*
Connector class	*connName*Globals.cpp

Table 89. Suggested naming conventions for a connector (continued)

Connector file	Name
Connector library	*connDir**connName*.dll Java package: com.crossworlds.connectors.*connName*. where *connDir* is the name of the connector directory, as defined above.
Connector startup script	Windows platforms: *connDir*\start_*connName*.bat UNIX-based platforms: *connDir*\connector_manager_*connName*.sh where *connDir* is the name of the connector directory, as defined above.

For more information on naming conventions for connectors, see *Naming IBM WebSphere InterChange Server Components* in the IBM WebSphere InterChange Server documentation set.

Compiling the connector

Once you have written the connector's application-specific component, you must compile it into an executable format, its *connector library*. This section provides information on how to compile and link a connector.

This section provides the following information:

- "Compiling and linking a C++ connector"
- "Running a debug version of a C++ connector" on page 201

Compiling and linking a C++ connector

To build the application-specific component of the connector, you need to include the connector header file, include any other required header files, compile the source files, and link `CwConnector.lib` to create the connector's dynamically loadable library (DLL).

Important:

1. Previous releases of IBM WebSphere InterChange Server and WebSphere Business Integration Adapters provided the Cayenne libraries for use in C++ connectors. However, with this release (version 2.4 of the WebSphere Business Integration Adapter Framework) the Cayenne libraries have been replaced by Standard Template Libraries (STL). Therefore, you must recompile any existing custom C++ connectors so that they use the new STL.
2. On Windows systems, the `CwConnector.lib` file is provided as part of the C++ Connector Development Kit (CDK). The CDK is supported *only* on Windows systems. Therefore, creation of a C++ connector is supported *only* on Windows systems. On a UNIX-based system, you *cannot* compile and link a C++ connector; however, you can run existing C++ connectors.

On a Windows system, use the MicroSoft Visual C++ 6.0 programming environment to build your connector, and follow these instructions:

1. Make sure that the system PATH variable includes the C++ connector library, `CwConnector.dll`, which resides in the `bin` subdirectory of the product directory.

2. In the Project Settings window under C/C++, add CDKIMPORT to the Preprocessor definitions for the project.
3. In Project Settings, C/C++, Additional include directories under the Preprocessor category, add:

   ```
   ..\..\generic_include
   ```

   Important: Because of the replacement of the Cayenne libraries with STL, you no longer need to include the `cayenne_include` directory in your C++ files. In addition, you can no longer use any Cayenne classes or methods in your C++ connector.
4. Define the information displayed in the Version tab of the Properties window for your connector DLL. Follow these steps:
   a. Create a file named `ConnectorVersion.h` and define the constants for your connector, such as Product Name and Product Version. A sample for this file is located in the following subdirectory of the product directory:

      ```
      DevelopmentKits\cdk\samples\sampleconnector\include
      ```

      Note: This sample `ConnectorVersion.h` file provides values for the Product Name and Product Version. Make sure you change these sample values to values that are appropriate for your connector. To check the version of a DLL, right-click on the DLL and choose the Properties>Version tab. The correct version must appear here.
   b. In the project file, make sure that the following file is added to the project:

      ```
      DevelopmentKits\cdk\ConnectorVersion.rc
      ```

   c. In the Additional Resources Include section, make sure the following `include` directories exist:
      - `..\..\generic_include`
      - the `include` directory for your connector

   The Version window uses the files `ConnectorVersion.rc` and `generic_include\CxResourceVersion.h`, which are shipped with the CDK. You need to define the `ConnectorVersion.h` file for your connector.
5. In the Project Setting Link tab, add the appropriate version of the C++ connector library (`CwConnector.lib`) to your project, as follows:
   - If you are building a debug version of your connector, add:

     ```
     ProductDir\DevelopmentKits\cdk\lib\Debug\CwConnector.lib
     ```

   - If you are building a release version of your connector, add:

     ```
     ProductDir\DevelopmentKits\cdk\lib\Release\CwConnector.lib
     ```
6. Compile and link the connector.
7. Create the C++ connector's library file, which is a dynamically loadable library (DLL).

   The suggested naming convention for the connector DLL file is to have it match the connector name (Table 89 on page 199). For more information, see "Naming the connector" on page 199.

   For example, for a C++ connector with a connector name of `MyCPP`, the name of its DLL is:

   ```
   MyCPP.dll
   ```

Running a debug version of a C++ connector

Use the MicroSoft Visual C++ 6.0 programming environment to run a debug version of a C++ connector. Assuming that the WebSphere Business Integration

Adapters product is installed into a directory that *ProductDir* represents, to run a debug version of a C++ connector, set the executable for the debug session under Project Settings to the following:

```
ProductDir\bin\java.exe
```

In the program arguments, set the debug parameters to:

```
-Duser.home=ProductDir
-classpath ProductDir\lib\crossworlds.jar;ProductDir\lib\rt.jar;
ProductDir\lib\mq.jar AppEndWrapper -ddllName -nconnectorName
-sICSinstanceName
```

Creating the connector definition

To run in the IBM WebSphere business integration system, a connector must be defined in the *repository*. Pre-defined adapters, which the WebSphere Business Integration Adapters product provides, have predefined connector definitions that are loaded in the repository at installation time. To run a predefined connector, a system administrator need only configure the application and set the connector's configuration properties. However, before the IBM WebSphere business integration system can access a connector that you have developed, you must take the following steps:

- Create a connector definition to define the connector within the repository.
- Create an initial configuration file to assist users in connector configuration (optional).

Defining the connector

To define the connector within the WebSphere business integration system, you create a *connector definition*. This connector definition includes the following information to define the connector in the repository:

- The name of the connector definition
- Supported business objects and associated maps
- Connector configuration properties

A tool called Connector Configurator collects this information and stores it in the repository.

WebSphere InterChange Server

When your integration broker is InterChange Server, the repository is a database that InterChange Server communicates with to obtain information about components in the WebSphere business integration system. In this repository, connector definitions reside. These connector definitions include both standard and connector-specific connector configuration properties that the connector controller and the client connector framework require. The connector can also have a local configuration file, which provides configuration information for the connector locally. When a local configuration file exists, it takes precedence over the information in the InterChange Server repository.

You update the connector definitions in the InterChange Server repository with Connector Configurator from within the System Manager tool. You can update the locale configuration file with the standalone version of Connector Configurator, which resides in the `bin` subdirectory of your product directory.

WebSphere MQ Integrator Broker

When your integration broker is WebSphere MQ Integrator Broker, the repository is a directory of files that the connector framework uses to obtain information about components of the WebSphere business integration system. In this repository, connector definitions for each adapter in the system resides.

You update the connector definitions in the local repository with Connector Configurator, which resides in the `bin` subdirectory of your product directory.

For information on how to use Connector Configurator, refer to Appendix B, "Connector Configurator," on page 329

The connector definition name

The connector definition name uniquely identifies the connector within the WebSphere business integration system. By convention, a connector definition name usually takes the following form:

connName`Connector`

where *connName* is the connector name (see Table 89 on page 199). For more information on the connector name, see "Naming the connector" on page 199. For example, if the connector name is `MyConn`, the name of its connector definition is `MyConnConnector`.

Supported business objects and maps

A connector definition must specify the following information about the business objects that the connector supports:

- The business object definitions

  Each business object that the connector is able to send to or receive from the integration broker must be specified as a supported business object. Connector Configurator provides a Supported Business Objects tab in which you can enter the connector's supported business objects.

  Note: All application-specific business objects that the connector supports must be defined in the repository *before* you can include them as supported business objects in the connector definition. For information on how to define application-specific business objects, see the *Business Object Development Guide*.

- Associated maps

WebSphere InterChange Server

Only the connector definition for a connector that communicates with InterChange Server as its integration broker includes the maps associated with the connector. Associated maps are those maps that convert between the connector's application-specific business objects and the appropriate generic business objects.

Connector Configurator provides an Associated Maps tab in which you can enter the connector's associated maps.

Connector configuration properties

The connector definition also contains the connector configuration properties. To initialize these properties, you must take the following steps:

- Assign values for standard connector configuration properties.
- Define any connector-specific configuration properties that your connector uses and assign them values as appropriate.

Connector Configurator provides two tabs for specifying connector configuration properties: Standard Properties and Connector-Specific Properties. For more information on connector configuration properties, see "Using connector configuration property values" on page 69..

Creating the initial configuration file

By convention, pre-defined adapters provide an initial configuration file for users to use the first time that they configure the adapter with Connector Configurator. The suggested name for this configuration file is:

`CN_`*`connName`*`.txt`

where *connName* is the connector name (see Table 89 on page 199). For more information on the connector name, see "Naming the connector" on page 199. This initial configuration file resides in the following directory:

`ProductDir``\repository\`*`connName`*

That is, the `repository` subdirectory of the product directory contains directories for each connector. Each connector's directory (*connName*) is named with its unique connector name and within this directory resides the initial configuration file with the following name.

For users to configure a connector that you have developed, you can provide an initial configuration file for your new connector. As part of your connector development, you have probably specified the settings for the standard configuration properties as well as defining any connector-specific configuration properties. This connector configuration information should reside in your repository. However, once your connector is moved to some other environment, it loses access to this repository. Therefore, you should create an initial configuration file that is part of your released connector.

To create this initial configuration file, bring up Connector Configurator for your connector and save its configuration in the following file:

`ProductDir``\repository\`*`connName`*`\CN_`*`connName`*`.txt`

Note: These steps assume that during the course of development, you have already created a connector configuration file (.cfg) for your connector. The preceding step just saves this connector configuration information in a separate file, which is included as part of the released connector.

Starting up a new connector

To start up the connector, you execute a *connector startup script*. As Table 90 shows, the name of this startup script depends on the operating system which you are using.

Table 90. Startup scripts for a connector

Operating system	Startup script
UNIX-based systems	`connector_manager_connName`
Windows	`start_connName.bat`

The startup script supports those adapters that the WebSphere Business Integration Adapters product provides. To start up a predefined connector, a system administrator runs its startup script. The startup scripts for most predefined connectors expect the following command-line arguments:

1. The first argument is the connector name, which identifies the following:
   - The name of the connector's directory under the `connectors` subdirectory of the product directory
   - The connector library, which resides in the connector's directory
2. The second argument is the name of the integration broker instance against which the connector runs.

WebSphere InterChange Server

When your integration broker is InterChange Server (ICS), the startup script specifies the name of the ICS instance against which your connector runs. On Windows systems, this ICS instance name (which was specified in the installation process) appears in each of the connector shortcuts of the startup script.

Other integrator brokers

When your integration broker is a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the startup script specifies the name of the broker instance against which your connector runs. On Windows systems, this instance name (which was specified in the installation process) appears in each of the connector shortcuts of the startup script.

3. Optional additional startup parameters can be specified on the command line and are passed to the connector runtime.

   For more information about the startup parameters, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set or your implementation guide in the WebSphere Business Integration Adapters documentation set.

WebSphere InterChange Server

Before you start a connector, InterChange Server must be running for the connector to complete its initialization and obtain its business objects from the repository.

Before you can start up a connector that you have developed, you need to ensure that a startup script supports your new connector. To enable a startup script to start your own connector, you must take the following steps:

1. Prepare a connector directory for your connector.
2. Create the startup script for your connector. For Windows systems, also create a shortcut for your connector startup.
3. Set up the startup script as a Windows service (optional).

The following sections describe each of these steps.

Preparing the connector directory

The *connector directory* contains the runtime files for your connector. To prepare the connector directory, take the following steps:

1. Create a connector directory for your new connector under the `connectors` subdirectory of the product directory:

   ProductDir`\connectors\`*connName*

   By convention, this directory name matches the connector name (*connName*). The connector name is a string that uniquely identifies the connector. For more information, see "Naming the connector" on page 199.
2. Move your connector's library file to this connector directory.

   A C++ connector's library file is a DLL. You created this DLL when you compiled the connector. For more information, see "Compiling and linking a C++ connector" on page 200.

Creating startup scripts

As Table 90 on page 205 shows, a connector requires a startup script for the system administrator to start execution of the connector process. The startup script to use depends on the operating system on which you are developing your connector.

Startup script and shortcut on Windows systems

Starting a connector on a Windows system involves the following steps:

1. Call the connector's startup script, `start_`*connName*`.bat`.

   The `start_`*connName*`.bat` script (where *connName* is the name of your connector) is a connector-specific startup script. It provides connector-specific information (such as application-specific libraries and their locations). By convention, this script resides in the connector directory:

   ProductDir`\connectors\`*connName*

   It is this `start_`*connName*`.bat` script that the user invokes to start the connector on a Windows system.
2. Call the generic connector-invocation script, `start_adapter.bat`

   The `start_adapter.bat` file is generic to all connectors. It performs the actual invocation of the connector within the JVM. It resides in the `bin` subdirectory of the product directory. The `start_`*connName*`.bat` script must call the `start_adapter.bat` script to actually invoke the connector.

Figure 77 shows the steps to start a connector on a Windows system.

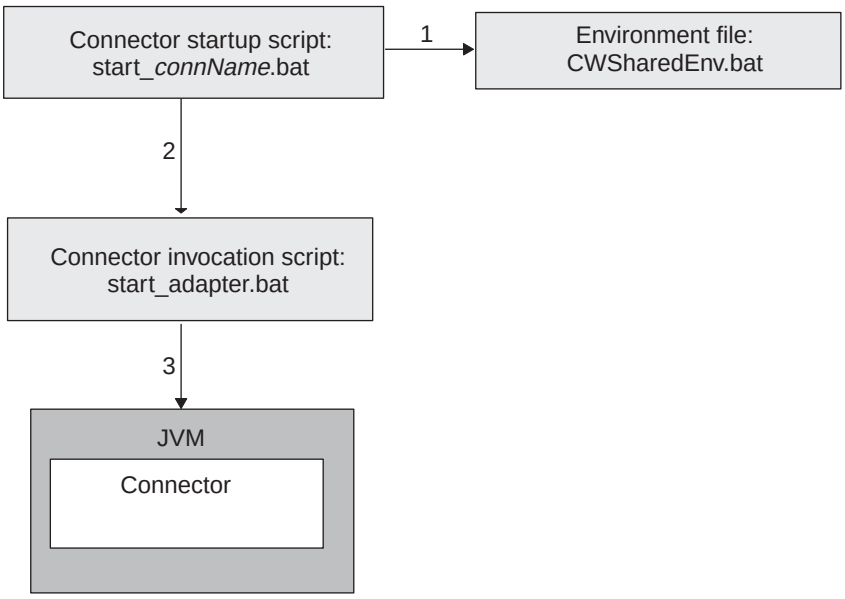

Figure 77. Starting a connector on a Windows system

When a WebSphere Business Integration Adapters Installer installs a predefined connector on a Windows system, it takes the following steps:

- Install a startup script for the predefined connector.
- Create a menu option for the predefined connector under the `Programs > IBM WebSphere Business Integration Adapters > Adapters > Connectors` menu.

Important: In previous releases, C++ connectors required a special startup script, called `start_connector.bat`. With this release (4.2.2 of IBM WebSphere InterChange Server and version 2.4 of WebSphere Business Integration Adapters), Java and C++ connectors can use the same startup script: `start_connName.bat`. New development of C++ connectors should use `start_connName.bat` as the startup script. However, the `start_connector.bat` script continues to be supported for backward compatibility.

To provide the ability to start up your own connector, you must duplicate these steps by:

- Generating the `start_connName.bat` startup script and putting it in the `connector\connName` subdirectory of the product directory
- Providing a menu option for the connector under the `Programs > IBM WebSphere Business Integration Adapters > Adapters > Connectors` menu. Each menu option is a shortcut that invokes the Windows startup script, `start_connName.bat`, for the particular connector.

Creating the startup script: To create a custom connector startup script, you create a new connector-specific startup script called `start_connName.bat` (where *connName* is your C++ connector name). For example, if your C++ connector has a connector name of `MyCPP`, its startup script name is `start_MyCPP.bat`. As a starting point, you can copy the startup-script template, which is located in the following file:

`ProductDir\templates\start_connName.bat`

Figure 78. shows a sample of the contents of the startup-script template for Windows. Please consult the version of this file released with your product for the most current contents.

```
REM A sample of start_connName.bat which calls start_adapter.bat
@echo off

call "%ADAPTER_RUNTIME%"\bin\wbia_connEnv.bat
setlocal

REM If required, goto the connector specific directory. CONNDIR is defined
RED by caller
cd /d %CONNDIR%

REM set variables that need to pass to start_adapter.bat
REM set JVMArgs=
REM set JCLASSES=
REM set LibPath=
REM set ExtDirs=

REM A sample to start a C++ connector
REM call start_adapter.bat -nconnName -sServerName -dconnectorDLLfile -f...
REM -p... -c... ...

REM A sample to start a Java connector
call start_adapter.bat -nconnName -sserverName -lconnectorSpecificClasses
-f... -p... -c... ...

endlocal
```

Figure 78. Sample contents of the startup-script template for Windows

By convention, the `start_`*`connName`*`.bat` script has the standard syntax shown in Figure 79, with *connName* being the name of the connector, *ICSinstance* being the name of the InterChange Server instance, and *additionalOptions* specifies additional startup parameters to pass to the connector invocation. These options include -c, -f, -t, and -x. For more information, see Table 92 on page 210.

```
start_connName connName ICSinstance additionalOptions
```

Figure 79. Standard syntax for Windows connector startup script

As the connector developer, you control the content of `start_`*`connName`*`.bat`. Therefore, you can change the syntax of your connector startup script. However, if you change this standard syntax, make sure that all information that `start_adapter.bat` requires is available at the time of its invocation within `start_`*`connName`*`.bat`.

Note: In the `start_`*`connName`*`.bat` syntax in Figure 79, the *connName* and *ICSinstance* arguments are required. The *additionalOptions* argument is optional.

The startup script with the standard syntax makes the following assumptions about your connector's runtime files based on the connector name (*connName*):

- The connector name is the same as name of the connector directory under the `connectors` subdirectory of the product directory
- The connector name is the same as the C++ connector's library file (its DLL: *connName*`.dll`), which resides in the connector directory

For example, for the `MyCPP` connector to meet these assumptions, its runtime files must reside in the *ProductDir*`\connectors\MyCPP` directory and its DLL must reside in that directory with the name `MyCPP.dll`. If y our connector cannot meet these

assumptions, you must customize its startup script to provide the appropriate information to the generic connector-invocation script, `start_adapter.bat`.

In this `start_connName.bat` file, take the following steps:

1. Call the `CWConnEnv.bat` environment file to initialize the startup environment.
2. Move into the connector directory.
3. Set the startup environment variables within the startup script with any connector-specific information and any connector-specific variables.
4. Call the `start_adapter.bat` script to invoke the connector.

The following sections describe each of these steps.

Calling the environment file: The `CWConnEnv.bat` file contains environment settings for the IBM Java Object Request Broker (ORB) and the IBM Java Runtime Environment (JRE). The following line invokes this environment file within the startup script:

```
call "%ADAPTER_RUNTIME%"\bin\CWConnEnv
```

Moving into the connector directory: The `start_connName.bat` script must change to the connector directory *before* it calls the `start_adapter.bat` script. The connector directory contains the connector-specific startup script as well as other files needed at connector startup. You can define the name of this connector directory any way you wish. However, as discussed in "Preparing the connector directory" on page 206, by convention the connector directory name matches the connector name.

If the `start_connName.bat` script uses the standard syntax (see Figure 79 on page 208), the connector name is passed in as the first argument (%1). In this case, the following lines move into the connector directory:

```
REM set the directory where the specific connector resides
set CONNDIR=%CROSSWORLDS%\connectors\%1

REM goto the connector specific drive & directory
cd /d %CONNDIR%
```

Alternatively, because the connector name is used in several components of the connector, you can define an environment variable to specify this connector name and then evaluate this environment variable for all subsequent uses of the connector name within the `start_connName.bat` script. The lines to set the environment variables for the connector name and connector directory could be as follows:

```
REM set the name of the connector
set CONNAME=%1

REM set the directory where the specific connector resides
set CONNDIR=%CROSSWORLDS%\connectors\%CONNAME%

REM goto the connector specific drive & directory
cd /d %CONNDIR%
```

Setting the environment variables: In the `start_connName.bat` script, you must provide any of the connector-specific information that the environment variables listed in Table 91 specify.

Table 91. Environment variables in the connector startup script

Variable name	Value
ExtDirs	Specify the location of any application-specific jar files.

Table 91. Environment variables in the connector startup script (continued)

Variable name	Value
JCLASSES	Specify any application-specific jar files. Jar files are separated with a semicolon (;).
JVMArgs	Add any arguments to be passed to the Java Virtual Machine (JVM).
LibPath	Specify any application-specific library paths.

The `start_adapter.bat` file uses the information in Table 91 as follows:

- It appends the JCLASSES and `LibPath` environment variables to the appropriate variables within the connector framework.
- It sets the external directories (`java.ext.dirs`) with the `ExtDirs` environment variable.
- It includes the `JVMArgs` environment variable in its list of arguments it passes to the JVM.

In addition to the environment variables in Table 91, you can also define your own connector-specific environment variables. Such variables are useful for information that can change from release to release. You can set the variable to a value appropriate for this release and then include the variable in the appropriate line of the startup script. If the information changes in the future, you only have to change the variable's value. You do not have to locate all lines that use this information.

Invoking the connector: To actually invoke the connector within the JVM, the `start_`*`connName`*`.bat` script must call the `start_adapter.bat` script. The `start_adapter.bat` script provides information to initialize the necessary environment for the connector runtime (which includes the connector framework) with its *startup parameters*. Therefore, you must provide the appropriate startup parameters to `start_adapter.bat`. Table 92 shows the startup parameters that the `start_adapter.bat` script recognizes.

Table 92. Startup parameters for start_adapter.bat script

Startup parameter	Description	Required?	Valid as additional command-line option to start_connName.bat?
-c*configFile*	The full path name of the connector's configuration file	Required if integration broker is other than ICS	Yes
-d*dllName*	The name of the C++ connector's library file (*dllName*), which is a dynamic link library (DLL). This DLL name should *not* include the `.dll` file extension.	Yes, for all C++ connectors	No

Table 92. Startup parameters for start_adapter.bat script (continued)

Startup parameter	Description	Required?	Valid as additional command-line option to start_connName.bat?
-f*pollFrequency*	The amount of time between polling actions. Possible *pollFrequency* values are: • The number of milliseconds between polling actions • `key`: causes the connector to poll only when you type the letter p in the connector's startup window. The `key` option must be specified in lowercase. • `no`: causes the connector not to poll. The `no` option must be specified in lowercase.	No Default is 1000 milliseconds	Yes
-j	Indicates that the connector is written in Java	No, as long as you specify the `-l` option for Java connectors	No
-l*classname*	The name of the Java connector's connector class (*className*)	Yes, for all Java connectors	No
-n*connectorName*	The name of the connector (*connectorName*) to start	Yes	No
-s*brokerName*	The name of the integration broker (*brokerName*) to which the connector connects	Yes	No
-t	Boolean value to turn off or on the connector property `SingleThreadAppCalls`, which guarantees that all calls the connector framework makes to the application-specific component are with one call-triggered flow. The default value is `false`.	No	Yes
-x*connectorProps*	Initializes the value of an application-specific connector property. Use the following format for each property you specify: *propName=value*	No	Yes

Make sure that the call to `start_adapter.bat` includes the following startup parameters:

- All required startup parameters:
  - To specify the name of the connector definition: `-n`

    If the name of the connector is passed in as the first argument (%1) to the `start_`*`connName`*`.bat` script (see Figure 79 on page 208), the `-n` startup parameter can be specified as follows:

    ```
    -n%1Connector
    ```

    If you define an environment variable for the connector name (such as `CONNAME`), this `-n` parameter could appear as follows:

    ```
    -n%CONNAME%Connector
    ```

  - To specify the name of the InterChange Server instance: `-s`

    If the name of the ICS instance is passed in as the second argument (%2) to the `start_`*`connName`*`.bat` script (see Figure 79 on page 208), the `-s` startup parameter can be specified as follows:

    ```
    -s%2
    ```

Other integration brokers

When your integration broker is a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, WebSphere Integration Message Broker), or WebSphere Application Server, the -c option is also a required startup parameter.

- Language-specific startup parameters required for a C++ connector:

  To specify the name of the connector DLL: `-d`

  For example, if you follow the recommended naming conventions, the language-specific parameter for the C++ connector name is `MyCPP` would be:

  ```
  -dMyCPP
  ```

  If you define an environment variable for the connector name (such as `CONNAME`), this `-d` parameter could appear as follows:

  ```
  -d%CONNAME%
  ```

- Any optional startup parameters that apply to all invocations of your connector. Consult Table 92 on page 210. for a list of optional startup parameters.

The syntax for the call to `start_adapter.bat` should have the following format:

```
call start_adapter.bat -nconnName -sICSinstance languageSpecificParams
 -cCN_connNameConnector.cfg
 -...
```

For example, the following line invokes the `MyCPP` connector:

```
call start_adapter.bat -dMyCPP
    -nMyCPP -sICSserver -cMyCPPConnector.cfg  -...
```

Note: The preceding command line assumes that the connector is running against an InterChange Server instance whose name is `ICSserver`. If the connector runs against an instance of WebSphere MQ Integrator Broker or WebSphere Message Broker, that instance name would need to appear in the command line.

With the use of the `CONNAME` environment variable to hold the connector name, this call can be generalized to the following:

```
call start_adapter.bat -n%CONNAME% -s%2 languageSpecificParams
 -cCN_%CONNAME%Connector.cfg
 -...
```

For the call to `start_adapter.bat`, keep the following points in mind:

- Make sure that the line to invoke the connector runtime is all *on one line* in your startup script; that is, no carriage returns should exist at the line breaks shown in the sample startup line.
- The order of the parameters listed in the call to `start_adapter.bat` is *not* important.
- You might also want to have your call to `start_adapter.bat` handle any additional options that the user might pass into the call to `start_connName.bat`. In this case, you should provide "extra" arguments to pass to `start_adapter.bat` so that additional options are passed down to the actual connector invocation. For example, the following call to `start_adapter.bat` handles up to three additional command-line options:

  ```
  call start_adapter.bat -n%CONNAME% -s%2 languageSpecificParams
   -cCN_%CONNAME%Connector.cfg %3 %4 %5
  ```

Creating the shortcut: A shortcut enables a connector to be started from a menu option within `Programs > IBM WebSphere Business Integration Adapters > Adapters > Connectors`. The shortcut should list the call to the `start_`*`connName`*`.bat` script. If this script uses the standard syntax (see Figure 79 on page 208), the shortcut would have the following form:

```
ProductDir\connectors\start_connName connName ICSinstance
```

If you define your own syntax for your `start_`*`connName`*`.bat` script, you must ensure that the shortcut uses this custom syntax.

If your menu already contains a shortcut for a C++ connector that uses the `start_`*`connName`*`.bat` startup script, an easy way to create a shortcut is to copy this existing connector's shortcut and edit the shortcut properties to change the connector name or add any other startup parameters.

For example, for the `MyCPP`connector that uses the standard syntax for its startup script, you could create the following shortcut:

```
ProductDir\bin\start_MyCPP.bat MyCPP ICSinstance
```

Note: The preceding command line assumes that the connector is running against an InterChange Server instance whose name is `ICSinstance`. If the connector runs against a WebSphere MQ Integrator Broker instance, that instance name would appear in the shortcut command line.

Startup script on UNIX systems

The Connector Development Kit (CDK) is supported *only* on Windows systems. It is *not* supported on UNIX-based systems. Because creation of a C++ connector is not supported on a UNIX-based system, you do not need to create startup scripts for such a connector.

Starting a connector as a Windows service

You can set up a connector to run as a Windows service that can be started and stopped by a remote administrator. For more information, see the *System Installation Guide for Windows* in the IBM WebSphere InterChange Server documentation set or your implementation guide in the IBM WebSphere Business Integration Adapter documentation set.

Note: If you are using InterChange Server as your integration broker and you want to use the automatic-and-remote restart feature with the connector, do *not* start connector as a Windows service. Instead, start the MQ Trigger Monitor as a service. For more information, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set.

Part 3. C++ connector library API reference

Chapter 9. Overview of the C++ connector library

The C++ connector library include class libraries that you need to use when developing a connector. This connector library contains predefined classes for connectors in C++. You use these class libraries to derive connector classes and methods. The class libraries also provide utilities, such as methods to implement tracing and logging services.

For the development of a C++ connector, the C++ Connector Development Kit (CDK) provides the following versions of the C++ connector library:

- `CwConnector.dll` – A dynamic link library that contains C++ classes and methods that provide connectors with interfaces and utilities for connector initialization, business object handling, and interaction with InterChange Server.
- `CwConnector.lib` – Export library for `CwConnector.dll`

Important: Because the CDK is supported *only* on Windows systems, creation of a C++ connector is supported *only* on Windows systems.

These `CwConnector.lib` and `CwConnector.dll` files reside in the following subdirectory of the product directory:

```
DevelopmentKits\cdk\lib
```

Within this directory, the `Release` and `Debug` subdirectories include release and debug versions of these development libraries.

Note: For instructions on building a C++ connector to run on Windows, see "Compiling and linking a C++ connector" on page 200..

Classes

Table 93 lists the classes in the C++ connector library.

Table 93. Classes in the C++ connector library

Class	Description	Page
`BOAttrType`	Represents an attribute descriptor, which contains information about the properties of an attribute	219
`BOHandlerCPP`	Represents the base class for a business object handler. You extend this class to define one or more business object handlers for your connector.	231
`BusinessObject`	Represents a business object instance. It provides access to the names and values of attributes	241
`BusObjContainer`	Represents an array of business objects. It provides access to elements of the business object array	261
`BusObjSpec`	Represents a business object definition. It provides access to application-specific information for the business object as well as attribute properties	265
`CxMsgFormat`	Provides support for internationalized messages. *This class has been deprecated.*	271
`CxVersion`	Represents object versions for business objects	273
`GenGlobals`	Represents the base class for a connector. You extend this class to define your connector class and implement the required virtual methods	281

Table 93. Classes in the C++ connector library (continued)

Class	Description	Page
`ReturnStatusDescriptor`	Represents a return-status descriptor, which contains error and informational messages	299
`StringMessage`	Provides methods for accessing the contents of a StringMessage object.	307
`SubscriptionHandlerCPP`	Represents a subscription handler, which handles subscription management for the connector.	303
`Tracing`	Provides tracing services	309

Note: The `BusObjAndSpecSerializer` class is part of the `CwConnector.dll` file. However, it is *only* for internal use. Do *not* use this class or its methods when developing a connector.

Chapter 10. BOAttrType class

The BOAttrType class represents attributes of business objects. Each instance of the BOAttrType class is an attribute descriptor, which contains information about the properties of an attribute. You can access attribute descriptors through either the business object (BusinessObject instance) or the business object definition (BusObjSpec instance). Through the methods of the BOAttrType class, you can get property information, such as the name, type, properties, and application-specific information, for each attribute.

The header file for this class is BOAttrType.hpp. It resides in the following subdirectory of your product directory:

```
DevelopmentKits\cdk\generic_include
```

The BOAttrType class contains the following:

- "Attribute-type constants"
- "Member methods"

Attribute-type constants

The BOAttrType class defines numeric and type equivalents for attribute types. Table 94 summarizes the attribute-type constants in the BOAttrType class.

Table 94. Static constants of the BOAttrType class

Attribute data type	Numeric attribute-type constant	String attribute-type value
Boolean	BOOLEAN	"Boolean"
Business object: multiple cardinality		N/A
Business object: single cardinality		N/A
Date	DATE	"Date"
Double	DOUBLE	"Double"
Float	FLOAT	"FLFS
Integer	INTEGER	INTSTRING
Long text	LONGTEXT	LONGTEXTSTRING
Object	OBJECT	
String	STRING	STRSTRING

Member methods

The BOAttrType class defines methods that provide access to an attribute within a business object. Table 95 summarizes the methods in the BOAttrType class.

Table 95. Member methods of the BOAttrType class

Member method	Description	Page
BOAttrType()	Creates an instance of the BOAttrType class, which a business object definition can use as an attribute of a business object.	220
getAppText()	Retrieves the application-specific information for an attribute.	220
getBOVersion()	Retrieves the version of an attribute's business object definition.	221
getCardinality()	Retrieves the number of child business objects, 1 or n, that an attribute can reference.	221
getDefault()	Retrieves the default value of an attribute.	222

Table 95. Member methods of the BOAttrType class (continued)

Member method	Description	Page
`getMaxLength()`	Retrieves the maximum length of an attribute value.	222
`getName()`	Retrieves the name of an attribute.	222
`getRelationType()`	Retrieves the type of relationship between a parent business object and a child business object.	223
`getTypeName()`	Retrieves the name of the data type of an attribute.	223
`getTypeNum()`	Retrieves the integer that specifies the data type of an attribute.	224
`hasCardinality()`	Determines whether an attribute has a specified cardinality.	225
`hasName()`	Determines whether an attribute has a specified name.	225
`hasTypeName()`	Determines whether the data type of an attribute has a specified type name.	226
`isForeignKey()`	Determines whether an attribute is a foreign key value in the application.	226
`isKey()`	Determines whether an attribute value is a key value in the application.	226
`isMultipleCard()`	Determines whether an attribute can reference multiple child business objects.	227
`isObjectType()`	Determines whether the data type of an attribute is a business object type.	227
`isRequired()`	Determines whether an attribute is required for a business object.	228
`isType()`	Determines whether an attribute has a specified data type.	228

BOAttrType()

Creates an instance of the `BOAttrType` class, which is an attribute descriptor a business object definition can use as an attribute of a business object.

Syntax

```
public BOAttrType();
```

Return values

A newly instantiated attribute descriptor, which is an instance of the `BOAttrType` class.

Notes

The `BusObjSpec()` constructor in the `BusObjSpec` class instantiates the attribute descriptors for each attribute in a business object definition. To retrieve information about attribute properties, use the get methods of the `BOAttrType` class.

See also

getAttrDesc(), getAttribute()

getAppText()

Retrieves the application-specific information for an attribute.

Syntax

```
char * getAppText();
```

Parameters

None.

Return values

A character string that contains application-specific information for the attribute.

Notes

You can use the `getAppText()` method to retrieve the value of the AppSpecificText property of a business object definition attribute. If there is no application-specific information, the `getAppText()` method returns an empty string (“ “).

getBOVersion()

Retrieves the version of the business object definition that a child business object references.

Syntax

```
CXVersion getBOVersion();
```

Parameters

None.

Return values

The version of a business object definition.

Notes

You can use the `getBOVersion()` method to determine the business object definition for a child business object. The `getBOVersion()` method returns the version information in a `CxVersion` object. You can use methods of the `CxVersion` class to obtainversion information.

Note: This operation applies *only* to attributes that specify a contained business object. If the attribute does not contain a business object, the `getBOVersion()` method returns a version string with a default value of 0.0.0.

getCardinality()

Retrieves the cardinality of an attribute.

Syntax

```
char * getCardinality();
```

Parameters

None.

Return values

The cardinality of an attribute:

1 Indicates that the attribute has single cardinality: it can reference one child business object.

`n` Indicates that the attribute has multiple cardinality: it can reference multiple child business objects.

Notes

Cardinality defines whether an attribute references a single child business object or an array of child business objects. This operation applies only to attributes that contain a business object. If the attribute does *not* contain a business object, the `getCardinality()` method returns the value 1.

See also

hasCardinality(), isMultipleCard()

---

getDefault()

Retrieves the default value of an attribute.

Syntax

```
char * getDefault();
```

Parameters

None.

Return values

The default value of an attribute. If the attribute's default value is not set, the `getDefault()` method returns an empty string (" ").

---

getMaxLength()

Retrieves the maximum length of an attribute from the business object definition.

Syntax

```
int getMaxLength();
```

Parameters

None.

Return values

An integer that specifies the maximum length, in bytes, that an attribute value can have.

Notes

If the maximum length is 0 (zero) there is no length restriction on the attribute.

---

getName()

Retrieves the name of an attribute.

Syntax

```
char * getName();
```

Parameters

None.

Return values

A character string that contains the name of the attribute.

Notes

You can use the `getName()` method to retrieve the name of an attribute.

Examples

```
// Get the business object definition for the business object
BusinessObject *pBusObj = new BusinessObject ("Example");
BusObjSpec *theSpec = pBusObj->getSpecFor();
BOAttrType *theAttr;

for (int i=0; i < theSpec->getAttributeCount(); i++)
{
   // Determine Attribute Name
   theAttr = theSpec->getAttribute(i);

   // Set the attribute values to Fred
   pBusObj->setAttrValue(theAttr->getName(), "Fred"
   BOAttrType::STRING);

}
```

See also

hasName()

getRelationType()

Retrieves the type of relationship between a parent business object and a child business object.

Syntax

```
char * getRelationType();
```

Parameters

None.

Return values

A relationship type. Currently the only relationship type is Containment, which indicates that the parent business object contains one or multiple child business objects. If the attribute is *not* a child business object, the `getRelationType()` method returns a null value.

See also

hasCardinality(), isMultipleCard(), isObjectType()

getTypeName()

Retrieves the name of the data type of an attribute as a string.

Syntax

```
char * getTypeName();
```

Parameters

None.

Return values

The string type name for the data type of the attribute. The `getTypeName()` method returns the attribute-type strings shown in Table 97.

Table 96. String attribute-type values

Attribute data type	String attribute type
Boolean	`"Boolean"`
Date	`"Date"`
Double	`"Double"`
Float	`"Float"`
Integer	`"Integer"`
Long text	`"LongText"`
Object	`"Object"`
String	`"String"`

See also

getTypeNum(), hasTypeName(), isType()

getTypeNum()

Retrieves the numeric type code for the data type of the current attribute.

Syntax

```
int getTypeNum();
```

Parameters

None.

Return values

The numeric type code for the data type of the attribute. Compare this integer value with the attribute-type constants shown in Table 97 to determine the type.

Table 97. Numeric attribute-type constants

Attribute data type	Numeric attribute-type constant
Boolean	`BOOLEAN`
Date	`DATE`
Double	`DOUBLE`
Float	`FLOAT`
Integer	`INTEGER`
Long text	`LONGTEXT`
Object	`OBJECT`
String	`STRING`

Note: The BOAttrType class defines the numeric attribute-type constants listed in Table 97.

Examples

```
if (theSpec->getAttribute(i)->getTypeNum() == BOAttrType::STRING)
{
   ...
}
```

See also

getTypeName(), isType()

hasCardinality()

Determines whether an attribute has a specified cardinality.

Syntax

```
unsigned char hasCardinality(char * card);
```

Parameters

card **[in]** Is the cardinality value to use for checking. Valid cardinality values are:

1 - single cardinality

n - multiple cardinality

Return values

TRUE if the attribute has the specified cardinality or FALSE if it does not.

Notes

This operation applies only to attributes that contain child business objects.

If you specify the cardinality as null, the hasCardinality() method returns FALSE.

See also

getCardinality(), getRelationType(), isMultipleCard()

hasName()

Determines whether an attribute has the specified name.

Syntax

```
unsigned char hasName(char * name);
```

Parameters

name **[in]** Is the name of an attribute.

Return values

Returns TRUE if the attribute has the name that you specify or FALSE if it does not.

Notes

You can use the `hasName()` method to determine whether a business object definition uses the attribute that you name. If you specify the name as *`null`*, the `hasName()` method returns `FALSE`.

See also

getName()

hasTypeName()

Determines whether an attribute has a specified data type.

Syntax

```
unsigned char * hasTypeName(char * name);
```

Parameters

name **[in]** Is the name of a data type for an attribute. For a list of valid string attribute-type values, see Table 96 on page 224.

Return values

Returns `TRUE` if the attribute has the data type that you specify or `FALSE` if it does not.

Notes

You can use the `hasTypeName()` method to determine whether a business object definition uses the data type you have named for this attribute. If you specify the name as `null`, the `hasTypeName()` method returns `FALSE`.

See also

getTypeName()

isForeignKey()

Determines whether an attribute value is a foreign key value in the application.

Syntax

```
unsigned char isForeignKey();
```

Parameters

None.

Return values

Returns `TRUE` if the attribute is a foreign key field of the business object or `FALSE` if the attribute is not a foreign key field.

isKey()

Determines whether an attribute value is a key value in the application.

Syntax

```
unsigned char isKey();
```

Parameters

None.

Return values

Returns TRUE if the attribute is a key field of the business object or FALSE if the attribute is not a key field.

Notes

You can use the isKey() method to find all the key attributes for a business object.

isMultipleCard()

Determines whether an attribute has multiple cardinality.

Syntax

```
unsigned char isMultipleCard();
```

Parameters

None.

Return values

TRUE if the attribute has multiple cardinality or FALSE if it does not.

Notes

This operation applies only to attributes that contain child business objects. If the attribute is a simple one without a child business object, the isMultipleCard() method returns FALSE.

See also

See also the descriptions of the getCardinality() and hasCardinality() methods.

isObjectType()

Determines if an attribute's data type is an object type; that is, if it is a complex attribute (an array or a subobject).

Syntax

```
unsigned char isObjectType();
```

Parameters

None.

Return values

TRUE if the attribute has a business object type or FALSE if it does not.

Notes

An attribute that has a business object data type refers to a child business object.

isRequired()

Determines whether a business object requires a value for an attribute.

Syntax

```
unsigned char isRequired();
```

Parameters

None.

Return values

Returns `TRUE` if the attribute is required for the business object or `FALSE` if it is not.

Notes

You can use the `isRequired()` method to find all the required attributes for a business object.

isType()

Determines whether an attribute has the integer data type that you specify.

Syntax

```
unsigned char isType(int type);
```

Parameters

type **[in]** Is one of the following attribute-type constants, which specifies an attribute data type:

```
BOAttrType::OBJECT
BOAttrType::BOOLEAN
BOAttrType::INTEGER
BOAttrType::FLOAT
BOAttrType::DOUBLE
BOAttrType::STRING
BOAttrType::DATE
BOAttrType::LONGTEXT
```

Notes

You can use the `isType()` method to find an attribute of a certain data type in a business object definition. If you specify an invalid data type, the `isType()` method returns `FALSE`.

Examples

```
char *cp = NULL;
if(getTheSpec()->getAttribute(name)->isType(BOAttrType::STRING))

{
   cp = new char[strlen(newval)+1];
```

```
    strcpy(cp, newval);

    Values[getTheSpec()->getAttributeIndex(name)] = cp;
}
```

See also

getTypeNum()

Chapter 11. BOHandlerCPP class

The BOHandlerCPP class is the base class for the business object handlers of a C++ connector. It contains the code for one business object handler. From this class, a connector developer must derive business-object-handler classes (as many as needed) and implement the abstract method, doVerbFor(), for the business object handler.

Important: All C++ connectors *must* extend this virtual class. Developers *must* implement the single virtual method, doVerbFor(), in their derived business-object-handler class. If your connector handles request processing, your doVerbFor() method must provide verb processing for all supported verbs for the business object (or objects) it handles. If your connector does *not* provide request processing, it must still provide verb processing for the Retrieve verb.

An connector includes one or more business object handlers to perform tasks for the verbs in business objects. Depending on the active verb, a business object handler can insert business object data into an application, retrieve data, delete application data, or perform another task. For an introduction to request processing and business object handlers, see "Request processing" on page 25. For information on how to implement a business object handler, see Chapter 4, "Request processing," on page 73

The header file for this class is BOHandlerCPP.hpp. It resides in the following subdirectory of your product directory:

```
DevelopmentKits\cdk\generic_include
```

Table 98 summarizes the methods in the BOHandlerCPP class.

Table 98. Member methods of the BOHandlerCPP class

Member method	Description	Page
BOHandlerCPP()	Creates a business object handler.	232
doVerbFor()	Performs the action for the active verb of a business object.	232
generateAndLogMsg()	Generates a message from a set of predefined messages in a message file and logs the generated message in the connector's log destination.	234
generateAndTraceMsg()	Generates a trace message from a set of predefined messages in a message file and sends the generated trace message to the connector's log destination.	234
generateMsg()	Generates a message from a set of predefined messages in a message file.	235
getConfigProp()	Retrieves a connector configuration property from the repository.	237
getTheSubHandler()	Retrieves a pointer to the subscription handler. The caller can use this pointer to determine whether any subscriptions to a particular business object definition exist for the incoming business object.	237
logMsg()	Logs a message to the connector's log destination. Log messages must be contained in a message file that you provide for your connector.	238

Table 98. Member methods of the BOHandlerCPP class (continued)

Member method	Description	Page
`traceWrite()`	Logs a message to the connector's log destination. Log messages must be contained in a message file that you provide for your connector.	238

BOHandlerCPP()

Creates a business object handler.

Syntax

```
BOHandlerCPP();
```

Parameters

None.

Return values

None.

Notes

The `BOHandlerCPP()` constructor creates an instance of the `BOHandlerCPP` class, to which business object definitions can refer for performing the tasks of verbs in business objects. Typically, a connector developer derives a class from `BOHandlerCPP` and implements the constructor and `doVerbFor()` method for this derived class. The developer can call the constructor of this derived class in the `getBOHandlerforBO()` method of the `GenGlobals` class to instantiate one or more business object handlers.

See also

getBOHandlerforBO()

doVerbFor()

Performs the action for the active verb of a business object.

Syntax

```
virtual int doVerbFor(BusinessObject & theBusObj,
   ReturnStatusDescriptor * rtnStatusDesc);
```

Parameters

theBusObj **[in]** Is the business object whose active verb is to be processed.

rtnStatDesc **[out]**
Is the return-status descriptor object that `doVerbFor()` should update with an error or informational message to send to the integration broker to indicate the status of the operation.

Return values

An integer that indicates the outcome status of the verb operation. Compare this integer value with the following outcome-status constants to determine the status:

`BON_SUCCESS` The verb operation succeeded.

`BON_FAIL` The verb operation failed.

`BON_APPRESPONSETIMEOUT`
The application is not responding.

`BON_BO_DOES_NOT_EXIST`
The connector performed a Retrieve operation, but the application database does not contain a matching entity for the requested business object

`BON_MULTIPLE_HITS`
The connector found multiple matching records when retrieving using non-key values. The connector returns a business object only for the first matching record.

`BON_FAIL_RETRIEVE_BY_CONTENT`
The connector was not able to find matches for Retrieve by non-key values.

`BON_VALCHANGE`
At least one value in the business object changed.

`BON_VALDUPES` The requested operation found multiple records in the application database with the same key values.

Notes

The `doVerbFor()` method performs the action of the business object's active verb. This method is the primary public interface for the business object handler.

Important: The `doVerbFor()` method is a virtual method.Therefore, the connector *must* implement this method as part of the business object handler.

When a business object arrives from InterChange Server, the connector framework creates a return-status descriptor object and passes it (along with the business object) to the `doVerbFor()` method. This method performs the verb operation and then calls methods in the `ReturnStatusDescriptor` class to set the appropriate values in the return-status descriptor, as follows:

`seterrMsg()`	Sets a message in the return-status descriptor object if there is an informational, warning, or error return message.

The connector framework returns this return-status descriptor to the integration broker. It also returns the outcome status, which is the return code of the `doVerbFor()` method.

See also

seterrMsg()

generateAndLogMsg()

Generates a message from a set of predefined messages in a message file and logs the generated message in the connector's log destination.

Syntax

```
void generateAndLogMsg(int msgNum, int msgType, int argCount, ...);
```

Parameters

msgNum **[in]** Specifies the message number (identifier) in the message file.

msgType **[in]** Is one of the following message-type constants defined in the `CxMsgFormat` class:

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

argCount **[in]** Is an integer that specifies the number of parameters within the message text.

... **[in]** Is a list of message parameters for the message text.

Return values

None.

Notes

The `generateAndLogMsg()` method combines the functionality of the `generateMsg()` and `logMsg()` methods. By combining these two methods, `generateAndLogMsg()` frees up the memory required for the message string that `generateMsg()` produces.

Note: The `generateAndLogMsg()` method is also available in the `GenGlobals` class. It is provided in the `BOHandlerCPP` class for access to logging from within the business object handler.

Examples

The following example performs the same task as the example provides for the `generateMsg()` method:

```
ret_code = connect_to_app(userName, password);
// Message 1100 - Failed to connect to application
if (ret_code == -1) {
      msg = generateAndLogMsg(1100, CxMsgFormat::XRD_ERROR, 0, NULL);
      return BON_FAIL;
}
```

generateAndTraceMsg()

Generates a trace message from a set of predefined messages in a message file and sends the generated trace message to the connector's log destination.

Syntax

```
void generateAndTraceMsg(int msgNum, int msgType, int traceLevel,
      int argCount, ...);
```

Parameters

msgNum **[in]** Specifies the message number (identifier) in the message file.

msgType **[in]** Is one of the following message-type constants defined in the `CxMsgFormat` class:

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

traceLevel Is one of the following trace-level constants to identify the trace level used to determine which trace messages are output:

```
CWConnectorUtil.LEVEL1
CWConnectorUtil.LEVEL2
CWConnectorUtil.LEVEL3
CWConnectorUtil.LEVEL4
CWConnectorUtil.LEVEL5
```

The method writes the trace message when the current trace level is greater than or equal to `traceLevel`.

Note: Do not specify a trace level of zero (`LEVEL0`) with a tracing message. A trace level of zero indicates that tracing is turned off. Therefore, any trace message associated with a *`traceLevel`* of `LEVEL0` will never print.

argCount **[in]** Is an integer that specifies the number of parameters within the message text.

... **[in]** Is a list of message parameters for the message text.

Return values

None.

Notes

The `generateAndTraceMsg()` method combines the functionality of the `generateMsg()` and `traceWrite()` methods. By combining these two methods, `generateAndTraceMsg()` frees up the memory required for the message string that `generateMsg()` produces. You no longer need to include the call to the `freeMemory()` method to release the memory allocated for the message string.

Note: The `generateAndTraceMsg()` method is also available in the `GenGlobals` class. It is provided in the `BOHandlerCPP` class for access to tracing from within the business object handler.

Examples

```
if(tracePtr->getTraceLevel()>= Tracing::LEVEL3) {
   // Message 3033 - Opened main form for object
   msg = generateAndTraceMsg(3033,CxMsgFormat::XRD_FATAL,
      Tracing::LEVEL3,0, NULL);
}
```

generateMsg()

Generates a message from a set of predefined messages in a message file.

Syntax

```
char * generateMsg(int msgNum, int msgType, char * info,
      int argCount, ...);
```

Parameters

msgNum **[in]** Specifies the message number (identifier) in the message file.

msgType **[in]** Is one of the following message types defined in the `CxMsgFormat` class:

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

info **[in]** Is an informational value, such as the name of the class for which the IBM WebSphere business integration system generated the message.

argCount **[in]** Is an integer that specifies the number of parameters within the message text (optional).

... **[in]** Is an optional list of message parameters for the message text, separated by commas. Each parameter is a char * value.

Return values

A pointer to the generated message.

Notes

The `generateMsg()` method allocates memory to store a generated message. When the connector has logged the message, it should call the `freeMemory()` method to release the allocated memory. This method is a member of the connector framework class `JToCPPVeneer`. The syntax of the call is `void freeMemory(char * mem)`, where `mem` is the memory allocated by `generateMsg()`. See the sample code below for an example of how to call this method.

If the `msgtype` parameter is invalid, the message generation process does not validate it. The `generateMsg()` method displays an alert that the message is not in the message file.

Note: The `generateMsg()` method is also available in the `GenGlobals` class. It is provided in the `BOHandlerCPP` class for access to message-file messages from within the business object handler.

Examples

```
char * msg;
ret_code = connect_to_app(userName, password);
// Message 1100 - Failed to connect to application
if (ret_code == -1) {
   msg = generateMsg(1100, CxMsgFormat::XRD_ERROR, NULL, 0, NULL);
   logMsg(msg);
   JToCPPVeneer::getTheHandlerStuff()->freeMemory(msg);
   return BON_FAIL;
}
```

getConfigProp()

Retrieves a connector configuration property from the repository.

Syntax

```
int getConfigProp(char * prop, char * val, int nMaxCount);
```

Parameters

prop **[in]** Is the name of the property to retrieve.

val **[out]** Is a pointer to a buffer to which the method can write the property value.

nMaxCount **[in]**
Is the number of bytes in the value buffer.

Return values

An integer that specifies the number of bytes that the method copied to the value buffer.

Notes

When you call `getConfigProp("ConnectorName")` in a parallel-process connector (one that has the `ParallelProcessDegree` connector property set to a value greater than 1), the method always returns the name of the connector-agent master process, regardless of whether it is called in the master process or a slave process.

Examples

```
if (getConfigProp("LoginId", val, 255) == 0);
{
   logMsg("Invalid LoginId";
   traceWrite(Tracing::LEVEL3, "Invalid LoginId", NULL);
}
```

getTheSubHandler()

Retrieves a pointer to the subscription handler. The caller can use this pointer to determine whether any subscriptions to a particular business object definition exist for the incoming business object.

Syntax

```
SubscriptionHandlerCPP * getTheSubHandler() const;
```

Parameters

None.

Return values

A pointer to the subscription handler.

Notes

Through the subscription handler, the connector keeps track of the subscribers for every verb of each business object definition that the connector publishes, in a consolidated list of all active subscriptions.

Examples

```
if (getTheSubHandler->isSubscribed(theObj->getName(), "Create"){
}
```

See also

See the descriptions of the `BusinessObject` and `BusObjSpec` classes.

logMsg()

Logs a message to the connector's log destination.

Syntax

```
void logMsg(char * msg);
```

Parameters

msg **[in]** Is a pointer to the message text.

Return values

None.

Notes

To generate the message string for `logMsg()`, use the `generateMsg()` method. The `generateMsg()` method retrieves a predefined message from a message file, formats the text, and returns a pointer to a generated message string.

Connector messages logged with `logMsg()` are viewable using LogViewer if the message strings were generated with `generateMsg()`. Trace messages logged with `traceWrite()` are not displayed in the LogViewer.

Note: The `logMsg()` method is also available in the `GenGlobals` class. It is provided in the `BOHandlerCPP` class for access to logging from within the business object handler.

Examples

```
if ((form = CreateMainForm(conn, getFormName(theObj))) < 0) {
   msg = generateMsg(10, CxMsgFormat::XRD_FATAL, NULL, 0, NULL);
   logMsg(msg);
}
```

See also

See the description of the `GenGlobals::generateMsg()` utility.

traceWrite()

Writes a tracing message to the log destination.

Syntax

```
void traceWrite(int traceLevel, char * info,
        char * filterName);
```

Parameters

traceLevel **[in]** Is one of the following trace levels, to use for writing the message:

```
Tracing::LEVEL1
Tracing::LEVEL2
Tracing::LEVEL3
Tracing::LEVEL4
Tracing::LEVEL5
```

The method writes the trace message when the current trace level is greater than or equal to `traceLevel`.

Note: Do not specify a trace level of zero (`LEVEL0`) with a tracing message. A trace level of zero indicates that tracing is turned off. Therefore, any trace message associated with a *`traceLevel`* of `LEVEL0` will never print.

info **[in]** Is a pointer to the message text.

filterName **[in]** Is a pointer to the filter to use for writing the message.

Return values

None.

Notes

You can use the `traceWrite()` method to write your own tracing messages for an application.

To write a tracing message without a filter, specify NULL for `filterName`.

Note: The `traceWrite()` method is also available in the `GenGlobals` class. It is provided in the `BOHandlerCPP` class for access to tracing from within the business object handler.

Examples

```
traceWrite(Tracing::LEVEL3, "Invalid LoginId", NULL);
```

See also

See also the description of the Tracing class.

Chapter 12. BusinessObject class

The BusinessObject class represents a the business objects for an application. Each instance of the BusinessObject class represents a single business object and references a single instance of the BusObjSpec class.

The header file for this class is BusinessObject.hpp. It resides in the following subdirectory of your product directory:

```
DevelopmentKits\cdk\generic_include
```

The BusinessObject class contains the following:

- "Attribute-value constants"
- "Member methods"

Attribute-value constants

Table 99 summarizes the attribute-value constants in the BusinessObject class.

Table 99. Static constants of the BusinessObject class

Special attribute value	Attribute-value constant
Blank	BlankValue
Ignore	IgnoreValue

Member methods

Table 100 summarizes the methods in the BusinessObject class.

Table 100. Member methods of the BusinessObject class

Member method	Description	Page
BusinessObject()	Creates a new business object that refers to the business object definition (BusObjSpec).	242
clone()	Copies an existing business object.	243
doVerbFor()	Calls the business object handler (instance of the BOHandlerCPP class) to perform the actions of the business object's active verb.	243
dump()	Formats and returns business object information in a standard or defined format for logging, and tracing.	244
getAttrCount()	Retrieves the number of attributes that the business object has.	245
getAttrDesc()	Retrieves an attribute description (BOAttrType) by name or position.	245
getAttrName()	Retrieves the name of an attribute by position.	246
getAttrType()	Retrieves an attribute type by name or position.	246
getAttrValue()	Retrieves an attribute value by name or position.	247
getBlankValue()	Retrieves the special blank value.	254
getDefaultAttrValue()	Retrieves the default value of an attribute value by name or position.	249
getIgnoreValue()	Retrieves the special "ignore" value.	250
getLocale()	Retrieves the locale associated with the business object.	250
getName()	Retrieves the name of the business object specification that the business object references.	251

Table 100. Member methods of the BusinessObject class (continued)

Member method	Description	Page
`getParent()`	Retrieves the parent business object of the current business object.	251
`getSpecFor()`	Retrieves a pointer to the name of the business object definition (`BusObjSpec`) to which the business object refers.	251
`getVerb()`	Retrieves the active verb for the business object.	252
`getVersion()`	Retrieves the version of the business object specification that the business object references.	252
`initAndValidateAttributes()`	If the connector configuration property UseDefaults is `TRUE`, this method sets any attributes with `NULL` values with the default values from the business object definition.	252
`isBlank()`	Determines whether the value of the attribute with the specified name or position is blank.	254
`isBlankValue()`	Determines whether a specified value is blank.	254
`isIgnore()`	Determines whether the value of the attribute with the specified name or position is "ignore".	255
`isIgnoreValue()`	Determines whether a specified value is the "ignore" value.	255
`makeNewAttrObject()`	Creates a new object of the correct type for the attribute with the specified name or position. This operation typically applies only to attributes that contain child objects.	256
`setAttrValue()`	Sets the value of an attribute by name or position.	256
`setDefaultAttrValues()`	Initializes the business object's attributes with their default values.	257
`setLocale()`	Sets the locale associated with the business object.	257
`setVerb()`	Sets the active verb for the business object.	258

BusinessObject()

Creates an instance of the `BusinessObject` class (a business object). The new business object refers to an instance of the `BusObjSpec` class (a business object definition), either the latest version of the business object definition or a version that you specify.

Syntax

```
BusinessObject(char * busObjName);
BusinessObject(char * busObjName, CxVersion & version);
BusinessObject(char * busObjName, char * localeName);
```

Parameters

busObjName **[in]**
Is the name of the new business object.

version **[in]** Is the version number of the business object definition to which the new business object refers. If you do not specify version, the new business object refers to the latest version of the business object definition. The version number is a String value.

*localeName***[in]** Is the name of the locale to associate with the business object.

Return values

None.

Notes

The `BusinessObject()` constructor creates a new business object instance whose type is the business object definition you specify in *busObjName*.

A business object (instance of the `BusinessObject` class) contains a set of attribute values. The definitions of the attributes are in the business object definition to which the business object refers. For each attribute, the business object definition defines a name, a position in the list of attributes, a data type, and several properties. The business object definition also contains the list of verbs that the business object supports.

If you specify a `localeName`, this locale applies to the data in the business object, *not* to the name of the business object definition or its attributes (which must be in English characters). For a description of the format for locale, see "Design considerations for an internationalized connector" on page 56

To determine whether the business object constructor has failed, check the business object definition pointer using the `getSpecFor()` method in this class. The `getSpecFor()` method will return `NULL` if the constructor has failed. The constructor might fail if an invalid name is specified.

Examples

```
BusinessObject *pObj = new BusinessObject("Customer");
```

See also

See also the description of the `BusObjSpec` class.

---

clone()

Copies an existing business object.

Syntax

```
BusinessObject * clone();
```

Parameters

None.

Return values

A copy of the current business object.

See also

See also the description of the `BusinessObject()` constructor.

---

doVerbFor()

Calls the business object handler (instance of the `BOHandlerCPP` class) to perform the action of the business object's active verb.

Syntax

```
int doVerbFor(ReturnStatusDescriptor * retStatusMsg);
```

Parameters

retStatusMsg **[out]**
Is the name of the status descriptor object containing an error or informational message for the integration broker.

Return values

An integer that specifies the outcome status of the verb operation.

Notes

The business object provides all the operations for the verbs that the BusinessObject definition supports.

The active verb is one of the list of verbs that the business object definition contains. To determine the active verb for a business object, you can use the `getVerb()` method.

Examples

```
BusinessObject *pObj;
...
pObj = new BusinessObject("Customer");
pObj->SetVerb ("Create");
pObj->setDefaultAttrValues();
retval = pObj->doVerbFor();
```

See also

`getVerb()`, `setVerb()`

See also the description of the `BOHandler` class.

---

dump()

Returns business object information in a format for logging and tracing.

Syntax

```
char * dump(char * buf, int bufSize);
```

Parameters

buf **[in]** Is the address of a buffer in which to store the business object information.

bufSize **[in]** Is the size of the buffer.

Return values

An integer that indicates the outcome of the operation.

Notes

If the saved business object does not fit in the buffer, the `dump()` method returns the error code -1.

Examples

```
BusinessObject *pObj;
int status;
. . .
status = pObj->dump(buf, 255);
```

See also

See also the description of the `Tracing` class.

getAttrCount()

Retrieves the number of attributes that are in the business object's attribute list.

Syntax

```
int getAttrCount();
```

Parameters

None.

Return values

An integer that specifies the number of attributes in the attribute list.

Examples

```
for(i=0; i<pObj.getAttrCount(); i++){
   char*cp;

   ...
   pObj.setAttrValue(i,cp,theObj.getAttrType(i));
}
```

See also

See also the description of the `getAttrIndex()` method under Chapter 14, "BusObjSpec class," on page 265

getAttrDesc()

Retrieves the attribute descriptor for an attribute of a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
BOAttrType * getAttrDesc(char * attrName);
BOAttrType * getAttrDesc(int position);
```

Parameters

attrName **[in]** Is the name of the attribute whose attribute descriptor is retrieved.

position **[in]** Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

Return values

A pointer to an instance of the BOAttrType class. The methods return NULL if an invalid attribute name or position is specified.

Notes

The getAttrDesc() method returns an attribute descriptor, which is an instance of the BOAttrType class, for an attribute in the business object (BusinessObject instance). The BOAttrType class provides methods to obtain information about the attribute properties. To retrieve the attribute descriptor for an attribute, you can identify the attribute by its name or its ordinal position in the list of attributes. If you specify an empty string (" ") or NULL as the attribute name, the getAttrDesc() method returns 0 (zero).

Note: If the connector is running at a trace level of 5, an appropriate trace message is also printed.

See also

getAttribute() getAttrName() getAttrType() getAttrValue()

For the methods of the BOAttrType class, see Chapter 10, "BOAttrType class," on page 219

getAttrName()

Retrieves the name of an attribute that you specify by its position in the business object's attribute list.

Syntax

```
char * getAttrName(int position);
```

Parameters

position **[in]** Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

Return values

The name of the specified attribute. The method will return NULL if an invalid position is specified.

Examples

```
strcpy(attr_name, pObj.getAttrName(1));

for (int i = 0; i<pObj.getAttrCount(); i++) {
   name = pObj.getAttrName(i);
   value = pObj.getAttrValue(i);
   cout << "name: " << name << "value " << value;
}
```

getAttrType()

Retrieves the data type of an attribute of a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
int getAttrType(char * attrName);
int getAttrType(int position);
```

Parameters

attrName **[in]** Is the name of the attribute whose data type is retrieved.

position **[in]** Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

Return values

An integer that represents the data type of an attribute. Attribute data types are defined in `BOAttrType.hpp`.

`0 = BOAttrType::OBJECT1 = BOAttrType::BOOLEAN2 = BOAttrType::INTEGER3 = BOAttrType::FLOAT4 = BOAttrType::DOUBLE5 = BOAttrType::STRING6 = BOAttrType::DATE7 = BOAttrType::LONGTEXT`

Notes

To retrieve the data type of an attribute of the business object, you can specify the attribute name or its position in the list of attributes. If you pass an empty string (" ") as an attribute name, or an invalid attribute position, the `getAttrType()` method returns -1. If the connector is running at a trace level of 5, an appropriate trace message is also generated.

Examples

```
pObj.setAttrValue("sti_address.docid","1234",
   pObj.getAttrType("sti_address.docid"));
```

See also

See also the descriptions of the `getAttrDesc()` and `getAttrName()` methods.

---

getAttrValue()

Retrieves the value of an attribute of a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
void * getAttrValue(char * attrName);
void * getAttrValue(int position);
```

Parameters

name **[in]** Is the name of an attribute whose value is retrieved.

position **[in]** Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The value of the specified attribute, in the format defined for the attribute's data type. The method will return NULL if an invalid position is specified.

Notes

When you use `getAttrValue()`, make sure that you check the type of the attribute with `getAttrType()` before you assign the returned value to a variable. Based on the attribute type, cast the `void` pointer to a character pointer, a business object pointer, or a business object container before assigning the returned value to a variable.

WebSphere InterChange Server

Attribute values that are not business objects or business object containers are stored as strings in C++ to handle the special "Blank"or "Ignore" values. The Blank value means "Clear this field; there is no data in it." The Ignore value means "The collaboration doesn't know or care what is in this field."

To handle attribute values that are not business objects or business object containers, cast the void pointer to a character pointer, and then determine whether the attribute is a special value. If the attribute value is not a special value and is a data type other than a string, you will need to convert the value to the correct type.

If you pass an empty string (" ") or `NULL` as a parameter for the attribute name or position, the `getAttrValue()` method displays Error Message 73:

```
Failed to set the attribute value because the attribute position "{1}" or
attribute type "{2}" is invalid.
```

If a business object has a specific attribute whose value has not been set, the `getAttrValue()` method returns `NULL`.

Examples

```
BusinessObject *pObj;
void *attr;
...
attr = pObj->getAttrValue(0):
if (pObj->getAttrType(0) == BOAttrType::STRING)
{
   char *attrStr = (char *) attr;
   if (pObj->isIgnoreValue(attrStr))
   {
      // ignore this value
   }
   else
      if (pObj->isBlankValue(attrStr))
      {
         // value should be blank
      }
...
}
```

See also

getAttrName() getDefaultAttrValue() setAttrValue()

getBlankValue()

Retrieves the special Blank attribute value.

Syntax

```
static char * getBlankValue();
```

Parameters

None.

Return values

The special Blank attribute value.

See also

getIgnoreValue() isBlank() isBlankValue()

---

getDefaultAttrValue()

Retrieves the default value of a business object attribute, given the attribute's name or its position in the business object's attribute list.

Syntax

```
char * getDefaultAttrValue(char * attrName);
char * getDefaultAttrValue(int position);
```

Parameters

name **[in]** Is the name of an attribute whose default value is retrieved.

position **[in]** Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The default value of the specified attribute, in the format defined for the attribute's data type. The methods will return `NULL` in the following cases:

- If an invalid position or name is specified
- If the default value is `NULL`
- If the attribute type is `BOAttrType::OBJECT`

Notes

To retrieve the default value of an attribute of the business object, you can specify the attribute name or its position in the list of attributes. If you pass an empty string or `NULL` as a parameter, the `getDefaultAttrValue()` method returns the "ignore value" 0 (zero).

An attribute can have a special default value, either blank or "ignore". Blank means "clear this field; there is no data in it." Ignore means "I don't know or don't care what is in this field."

Examples

```
tempStr = pObj->getDefaultAttrValue(i);
```

See also

getAttrValue() setDefaultAttrValues()

getIgnoreValue()

Retrieves the special Ignore value.

Syntax

```
static char * getIgnoreValue();
```

Parameters

None.

Return values

A string that contains the special Ignore attribute value.

Notes

The Ignore value indicates that the connector can ignore the value of this attribute.

See also

getBlankValue() isIgnore(), isIgnoreValue()

getLocale()

Retrieves the locale associated with the business object.

Syntax

```
char * getLocale();
```

Parameters

None.

Return values

A string that contains the name of the locale associated with the current business object. For information on the format of a locale name, see "What is a locale?" on page 56

Notes

The `getLocale()` method returns the locale that is associated with the flow for the business object, when the business object is created. This locale indicates the language and code set associated with the data in the business object, not to the name of the business object definition or its attributes (which must be characters in the code set associated with the U.S. English locale, `en_US`). If the business object does not have a locale associated with it, the connector framework uses the connector framework's locale for the business object.

See also

BusinessObject(), setLocale()

getName()

Retrieves the name of the business object definition that the business object references.

Syntax

```
char * getName();
```

Parameters

None.

Return values

The name of a business object definition.

See also

getVersion()

getParent()

Retrieves the parent business object of the current business object.

Syntax

```
BusinessObject *getParent() const;
```

Return values

The business object that contains the current business object. The method will return `NULL` if the business object is the top-level object in a hierarchy.

getSpecFor()

Retrieves a pointer to the business object definition (BusObjSpec instance) to which the business object refers.

Syntax

```
BusObjSpec * getSpecFor();
```

Return values

Pointer to the business object definition to which the business object refers.

Notes

You can use the `getSpecFor()` method to retrieve information about attributes of a business object.

Examples

```
int i = pObj.getSpecFor()->getAttributeIndex(name);
```

See also

See also the description of the BusObjSpec class.

getVerb()

Retrieves the active verb for the business object.

Syntax

```
char * getVerb();
```

Parameters

None.

Return values

The active verb for the business object. If the verb is not set in the business object, the method returns `NULL`.

Notes

The business object definition contains the list of verbs that the business object supports. The `getVerb()` method enables you to determine which verb is active for the current business object instance.

Examples

```
char * verb = pObj->getVerb();
if ((verb!=NULL) && (strcmp(verb,"Create")) == 0 {
   // perform the create operation
   ...
}
```

See also

`doVerbFor()`, `setVerb()`

getVersion()

Retrieves the version of the business object definition that the business object references.

Syntax

```
CxVersion * getVersion();
```

Parameters

None.

Return values

The version of a business object definition.

See also

getName()

initAndValidateAttributes()

Initializes attributes with default values and validates that required values exist.

Syntax

```
bool initAndValidateAttributes();
```

Parameters

None.

Return values

Returns TRUE if after processing default values if all required attributes have been set (have non-null values). If a required value has *not* been set and there is no default value specified for the attribute in the business object definition, returns FALSE.

Notes

The initAndValidateAttributes() method has two purposes:

- It *initializes* attributes by setting the value for each attribute to its default value under the following conditions:
  - When the UseDefaults connector configuration property is set to true
  - When the attribute's isRequired property is set to true (the attribute is required)
  - When the attribute's value is *not* currently set (has the special Ignore value of CxIgnore)
  - When the attribute's Default Value property specifies a default value
- It *validates* attributes by returning FALSE under the following conditions:
  - When the attribute's isRequired property is set to true
  - When the attribute does *not* have a Default Value property that defines its default value

  In case of failure, no value exists for some attributes (those without default values) after initAndValidateAttributes() finishes default-value processing. You might want to code your connector's application-specific component to catch this exception and return BON_FAIL.

The initAndValidateAttributes() method loops through every attribute in all levels of a business object and determines the following:

- Whether an attribute is required
- Whether the attribute has a value in the business object instance
- Whether the UseDefaults configuration property is set to true

If an attribute is required and UseDefaults is true, initAndValidateAttributes() sets the value of any unset attribute to its default value. If the attribute does *not* have a default value, initAndValidateAttributes() returns FALSE.

Note: If an attribute is a key or other attribute whose value is generated by the application, the business object definition should not provide default values, and the Required property for the attribute should be set to false.

The initAndValidateAttributes() method is usually called from the business-object-handler doVerbFor() method to ensure that required attributes have values before a Create operation is performed in an application. In the doVerbFor() method, you can call initAndValidateAttributes() for the Create verb. You can also call it for the Update verb, before it performs a Create.

To use initAndValidateAttributes(), you must also do the following:

- Design business object definitions so that the IsRequired attribute property is set to true for required attributes and that required attributes have default values specified in their Default Value property.
- Add the UseDefaults connector configuration property to the list of connector-specific properties for the connector. Set this property to true.

Examples

```
int ExampleBOHandler::doVerbFor(BusinessObject &theObj,
   ReturnStatusDescriptor *rtnStatusMsg)
{
   int status = BON_SUCCESS;

   // Determine the verb of the incoming business object
   char *verb = theObj.getVerb();

   if (strcmp(verb, CREATE) == 0)
      {
         if (!theObj->initAndValidateAttributes())
            return BON_FAIL;
      }
   else ...
```

isBlank()

Determines whether the value is the special Blank value for the attribute with the specified name or at the specified position in the attribute list.

Syntax

```
unsigned char isBlank(char * name);
unsigned char isBlank(int position);
```

Parameters

name **[in]** Is the name of an attribute.

position **[in]** Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns TRUE if the attribute value equals the blank value or FALSE if it does not.

See also

getBlankValue() isBlankValue() isIgnore()

isBlankValue()

Determines whether a specified attribute value is the special Blank attribute value.

Syntax

```
unsigned char isBlankValue(char * value);
```

Parameters

value **[in]** Is the attribute value to compare with the special Blank value.

Return values

Returns `TRUE` if the passed-in value equals the blank value or `FALSE` if it does not.

See also

getBlankValue() isBlank() isIgnoreValue()

isIgnore()

Determines whether the attribute value is the special Ignore value for the attribute with the specified name or at the specified position in the attribute list.

Syntax

```
unsigned char isIgnore(char * name);
unsigned char isIgnore(int position);
```

Parameters

name **[in]** Is the name of an attribute.

position **[in]** Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns 1 for True if the attribute value equals the special Ignore value or 0 for False if it does not.

See also

getIgnoreValue() isBlank() isIgnoreValue()

isIgnoreValue()

Determines whether a specified attribute value is the special Ignore value.

Syntax

```
unsigned char isIgnoreValue(char * value);
```

Parameters

value **[in]** is the attribute value to compare with the special Ignore value. This argument is defined as a `char *` because the method expects the special Ignore value.

Return values

Returns 1 for True if the passed-in value equals the special Ignore value or 0 for False if it does not.

Notes

The Ignore value, which `isIgnoreValue()` checks for, is a special attribute value that represents a value that the connector should ignore. After retrieving the value of an attribute with `getAttrValue()`, you can pass the returned value to `isIgnoreValue()` to determine whether the value is the special Ignore value.

See also

getIgnoreValue() isBlankValue() isIgnore()

makeNewAttrObject()

Creates a new business object of the correct type for the attribute with the specified name or at the specified position in the attribute list. This method is typically used with attributes that contain child objects.

Syntax

```
void * makeNewAttrObject(char * name);
void * makeNewAttrObject(int position);
```

Parameters

name **[in]** Is the name of an attribute.

*pos ition***[in]** Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

A pointer to a newly created instance of the BusinessObject class.

Notes

This method should only be used for attributes with an attribute type of `OBJECT`. The method creates a new business object of the proper type for an attribute but does not change the existing attribute. To set the value of the new business object, use `setAttrValue()`.

setAttrValue()

Sets the value of an attribute.

Syntax

```
unsigned char setAttrValue(char * name, void * newval, int type);
unsigned char setAttrValue(int position, void * newVal, int type);
```

Parameters

name **[in]** Is the name of the attribute whose value you want to set.

*position***[in]** Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

newVal **[in]** Is either a string representation of the value for the attribute or a pointer to the value. Must be char * or BusinessObject *.

type **[in]** Is one of the following attribute data types for the new value, as represented by one of the following constants:

```
BOAttrType::OBJECT
BOAttrType::BOOLEAN
BOAttrType::INTEGER
BOAttrType::FLOAT
```

```
BOAttrType::DOUBLE
BOAttrType::STRING
BOAttrType::DATE
BOAttrType::LONGTEXT
```

Return values

Returns True when the operation succeeded or False when the operation failed.

Notes

You can use the name or position method to set an attribute value. These methods verify that the new value has the correct data type before changing the attribute value. If `newval` is a character pointer, the method sets the value. If `type` is `OBJECT` and the attribute refers to a single cardinality object, the `setAttrValue()` method overwrites the previous business object with the new business object. If `type` is `OBJECT` and the attribute refers to a multiple cardinality object, the `setAttrValue()` method appends the business object to the container.

You can set an attribute to a special value, either `BusinessObject::BlankValue` or `BusinessObject::IgnoreValue`. Blank means "clear this field; there is no data in it." Ignore means "I don't know or don't care what is in this field."

Examples

```
unsigned char status;
if (status = pObj->setAttrValue("Interest Rate","0.065",
      BOAttrType::FLOAT);) == 0)
   // continue
```

See also

getAttrValue() setDefaultAttrValues()

setDefaultAttrValues()

Initializes the business object's attributes with their default values.

Syntax

```
void setDefaultAttrValues();
```

Parameters

None.

Return values

None.

See also

getDefaultAttrValue() setAttrValue()

setLocale()

Sets the locale associated with the business object.

Syntax

```
void setLocale(const char * localeName);
```

Parameters

localeName Is the name of the locale to associate with the current business object. For information on the format of a locale name, see "What is a locale?" on page 56

Return values

None.

Notes

The `setLocale()` method sets the business-object locale, which identifies the locale that is associated with the business object. This locale indicates the language and code encoding associated with the data in the business object, *not* with the name of the business object definition or its attributes (which must be characters in the code set associated with the U.S. English locale, `en_US`). If the business object does not have a locale associated with it, the connector framework assigns the connector-framework locale as the business-object locale.

See also

getLocale()

---

setVerb()

Sets the active verb of the current business object.

Syntax

```
void setVerb(char * newVerb);
```

Parameters

newVerb **[in]** Is a verb that is in the verb list of the business object definition to which the business object refers.

Return values

None.

Notes

The business object definition (`BusObjSpec` instance) contains the list of verbs that the business object supports. The verb that you set as the active verb must be on this list. Only one verb is active at a time for a business object.

Business objects typically support the Create, Retrieve, and Update verbs. A business object might support additional verbs, such as Delete. Every connector that supports a business object must implement all the verbs that it supports.

Examples

```
BusinessObject *pObj;
...
pObj = new BusinessObject("Customer");
pObj->setVerb("Create");
```

See also

doVerbFor(), getVerb()

Chapter 13. BusObjContainer class

The BusObjContainer class creates and maintains an array of one or more child business objects. This class supports business objects with a hierarchical structure. Each BusObjContainer instance is a container object into which you can insert business objects that are instances of a business object definition referenced by a compound attribute of a parent business object. The inserted objects are child business objects in the hierarchy.

Note: The deprecated name for an array of child business objects is a "business object container". This term is also used to name the connector library class that provides methods for accessing the child business objects in a business object array. You can think of this class as providing methods for handling an array of business objects.

The header file for this class is BusObjContainer.hpp. It resides in the following subdirectory of your product directory:

```
DevelopmentKits\cdk\generic_include
```

Table 101 summarizes the methods in the BusObjContainer class.

Table 101. Member methods of the BusObjContainer class

Member method	Description	Page
BusObjContainer()	Creates a business object array (container). You should never use this method to create business object arrays. The IBM WebSphere business integration system creates a business object array as needed for each attribute that refers to child business objects.	
getObject()	Retrieves the child business object that occupies a specified position in a business object array.	261
getObjectCount()	Retrieves the number of child business objects in a business object array.	262
getTheSpec()	Retrieves the business object definition for a business object array.	262
insertObject()	Inserts a child business object into a business object array at the next available position.	263
removeAllObjects()	Removes all business objects in a business object array.	263
removeObjectAt()	Removes the business object at the specified position in a business object array.	264
setObject()	Inserts a child business object into a business object array at a specified position.	264

getObject()

Retrieves the child business object that occupies a specified position in a business object array.

Syntax

```
BusinessObject * getObject(int index);
```

Parameters

index **[in]** Is an integer that specifies the position of a child business object in a business object array.

Return values

A pointer to a child business object, or `NULL` if there is no business object at the specified position in the business object array.

Notes

You can use the `setObject()` method to specify the position of the business object in the business object array.

See also

See also the description of the `setObject()` method.

---

getObjectCount()

Retrieves the number of child business objects in a business object array.

Syntax

```
int getObjectCount();
```

Parameters

None.

Return values

An integer that indicates the number of child business objects in a business object array.

Notes

You can use the `insertObject()` method to insert child business objects into the business object array.

Examples

```
// iterate through objects in a BusObjContainer
for(int i = 0; i < myCont->getObjectCount(); i++)
   BusinessObject *myObj = myCont->getObject(i);
   if(myObj != NULL)
      myObj->doVerbFor();
}
```

See also

See also the description of the `insertObject()` method.

---

getTheSpec()

Retrieves the business object definition for a business object array.

Syntax

```
BusObjSpec * getTheSpec() const;
```

Parameters

None.

Return values

A pointer to a business object definition.

Notes

Every child business object stored in a `BusObjContainer` object must have the same business object definition as the business object array does.

See also

See also the description of the `BusObjSpec()` class.

insertObject()

Inserts a child business object into a business object array at the next available position.

Syntax

```
void insertObject(BusinessObject * busObj);
```

Parameters

*busObj***[in]** Is a pointer to a child business object.

Return values

None.

Notes

`BusinessObject::SetAttrValue()` calls this method when you pass it a child business object as a parameter.

See also

See also the description of the `setObject()` method.

removeAllObjects()

Removes all business objects in a business object array.

Syntax

```
void removeAllObjects();
```

Parameters

None.

Return values

None.

removeObjectAt()

Removes a business object at a specified position in a business object array.

Syntax

```
int removeObjectAt(int index);
```

Parameters

index **[in]** Is an integer that specifies the position for a child business object in a business object array.

Return values

Returns -1 for failure or zero for success.

Notes

After the remove operation, the business object array is compacted. Indexes are decremented for all business objects that have an index number higher than that of the removed business object.

setObject()

Inserts a child business object into a business object array at a specified position.

Syntax

```
BusinessObject * setObject(int index, BusinessObject * busObj);
```

Parameters

index **[in]** Is an integer that specifies the position for a child business object in a business object array.

busObj **[in]** Is a pointer to a child business object.

Return values

A pointer to a child business object.

Notes

If there is already a business object at the specified position, the new one replaces it. The old one is deleted.

See also

See also the description of the `getObject()` method.

Chapter 14. BusObjSpec class

Each instance of a BusObjSpec class defines the content, format, and behavior of a business object. This is the business object definition. More than one business object can refer to the same business object definition. Different instances of the BusObjSpec class describe different business objects, varying only in the values of their attributes. Business object definitions that have the same name but different versions are separate instances of the BusObjSpec class.

A connector supports a set of business object definitions. Each connector's configuration information in the repository identifies the business object definitions that the connector supports.

WebSphere InterChange Server

A collaboration can subscribe to a business object definition for notification of events for all verbs or for individual verbs. The business object definition includes a list of verbs that it supports.

Each business object definition contains a reference to a business object handler, which performs the tasks for business object verbs.

The header file for this class is BusObjSpec.hpp. It resides in the following subdirectory of your product directory:

```
DevelopmentKits\cdk\generic_include
```

The BusObjSpec class has methods for retrieving information about business object attributes. Table 102 summarizes the methods in the BusObjSpec class.

Table 102. Member methods of the BusObjSpec class

Member method	Description	Page
BusObjSpec()	Do *not* call the constructor to create business object definitions in the repository. To create business object definitions, use Business Object Designer.	
getAppText()	Retrieves the application-specific information for the business object definition.	266
getAttribute()	Retrieves a business object attribute by name or position.	266
getAttributeCount()	Retrieves the number of attributes that are in the attribute list of the business object definition.	267
getAttributeIndex()	Retrieves the position of a business object attribute in the attribute list.	267
getMyBOHandler()	Retrieves the business object handler to which the business object definition refers.	267
getName()	Retrieves the name of a business object or business object definition.	268
getVerbAppText()	Retrieves the application-specific information for a verb.	268
getVersion()	Retrieves the version of the business object definition.	269
isVerbSupported()	Determines whether the business object definition supports a particular verb.	269

getAppText()

Retrieves the application-specific information for the business object definition.

Syntax

```
char * getAppText() const;
```

Parameters

None.

Return values

A character string that contains application-specific information for the business object definition. This method can return NULL.

Notes

If the business object definition has a value for the AppSpecificInfo property, the getAppText() method retrieves this value. Each attribute can also have is own application-specific information.

getAttribute()

Retrieves an attribute descriptor for an attribute of a business object, given the attribute's name or its position in the business object definition's attribute list.

Syntax

```
BOAttrType * getAttribute(char * name);
BOAttrType * getAttribute(int position);
```

Parameters

name **[in]** Is the attribute name.

position **[in]** Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

Return values

A pointer to an instance of the BOAttrType class. These methods return NULL if an invalid attribute name or position is specified.

Notes

The getAttribute() method returns an attribute descriptor, which is an instance of the BOAttrType class, for an attribute in the business object description (BusObjSpec instance). The BOAttrType class provides methods to obtain information about the attribute properties, such as its application-specific information, default value, and whether it is required.

See also

For the methods of the BOAttrType class, see Chapter 10, "BOAttrType class," on page 219

getAttributeCount()

Retrieves the number of attributes that are in the attribute list of the business object definition.

Syntax

```
int getAttributeCount();
```

Parameters

None.

Return values

An integer that specifies the number of attributes in the attribute list.

Notes

The number of attributes in the attribute list is the number of attributes that the business object definition supports.

See also

See also the description of the `BOAttrType` class.

getAttributeIndex()

Retrieves the position of a business object attribute in the attribute list.

Syntax

```
int getAttributeIndex(char * name);
```

Parameters

name **[in]** Is the name of the attribute.

Return values

An integer that specifies the position of the attribute in the attribute list of the business object definition.

Notes

After you retrieve the position of an attribute in attribute list, you can use the position to refer to the attribute.

See also

See also the description of the `BOAttrType` class.

getMyBOHandler()

Retrieves the business object handler to which the business object definition refers.

Syntax

```
BOHandlerCPP * getMyBOHandler() const;
```

Parameters

None.

Return values

A pointer to the business object handler.

Notes

More than one business object definition can use the same business object handler.

This method returns the handler that you derived from the `BOHandlerCPP` class and set with a call to the `getBOHandlerforBO()` method of the `GenGlobals` class.

See also

See also the description of the `BOHandlerCPP` class.

---

getName()

Retrieves the name of the business object definition.

Syntax

```
char * getName() const;
```

Parameters

None.

Return values

A character string that contains the name of the business object definition.

See also

See also the description of the `BusinessObject` class.

---

getVerbAppText()

Retrieves the application-specific information for the business object verb.

Syntax

```
char * getVerbAppText(char * verb);
```

Parameters

verb **[in]** Is the name of the verb whose application-specific information is retrieved.

Return values

A character string that contains application-specific information for the business object verb.

Notes

If the business object verb has a value for the `AppSpecificInfo` property, the `getVerbAppText()` method retrieves this value.

getVersion()

Retrieves the version of the business object definition.

Syntax

```
CxVersion * getVersion();
```

Parameters

None.

Return values

A pointer to an instance of the `CxVersion` class.

Notes

A business object definition refers to an instance of the `CxVersion` class. Each instance of this class contains a version number, subversion number, and point version number, separated by periods, as in 3.0.0.

See also

See also the description of the `CxVersion` class.

isVerbSupported()

Determines whether the business object definition supports a particular verb.

Syntax

```
unsigned char isVerbSupported(char * verb);
```

Parameters

verb **[in]** Is the name of a verb that the method determines if the current business object definition supports.

Return values

Returns `TRUE` if the business object definition supports the verb and `FALSE` if it does not.

Chapter 15. CxMsgFormat class

The CxMsgFormat class provides message-type constants to generate messages in different languages. The header file for this class is CxMsgFormat.hpp. It resides in the following subdirectory of your product directory:

```
DevelopmentKits\cdk\generic_include
```

This class provides the following:

- "Message-type constants"
- "Methods"

Message-type constants

T he CxMsgFormat class defines the message-type constants shown in Table 103..

Table 103. Message-type constants defined in the CxMsgFormat class

Message-type constant	Meaning
XRD_WARNING	A warning message
XRD_TRACE	A trace message
XRD_INFO	An informational message
XRD_ERROR	An error message
XRD_FATAL	A fatal error message

Methods

Table 104 summarizes the methods in the CxMsgFormat class.

Table 104. Member methods of the CxMsgFormat class

Member method	Description	Page
CxMsgFormat()	Creates an instance of the CxMsgFormat class. You *never* call this function to create a message object; the IBM WebSphere business integration system creates a message object automatically.	
generateMsg()	Generates a message.	271

Important: The methods of the CxMsgFormat class have been deprecated. For more information, see "Deprecated methods" on page 272.

generateMsg()

Generates a message with a predefined message from a message file.

Syntax

```
char * generateMsg(int msgNum, int msgType, char * info,
   int argCount, va_list v1);
```

Parameters

msgNum **[in]** Is an integer that specifies the message number in the message file.

msgType **[in]** Is one of the following message types:

```
XRD_UNKNOWN
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

info **[in]** Is an informational value, such as the name of the class for which the IBM WebSphere business integration system generated the message.

argCount **[in]** Is an integer that specifies the number of parameters within the message text (optional).

v1 **[in]** Is an optional list of parameters for the message text, separated by commas. Each parameter is a char * value.

Return values

A pointer to the generated message.

Notes

You can use the `generateMsg()` method to generate messages in different languages. For each language that your application supports, you can create a separate message file for messages in that language.

With the `XRD_TRACE` message type, you can generate trace messages.

Important: The `generateMsg()` method in this class has been deprecated. Use the `generateMsg()` method found in the `GenGlobals` and `BOHandlerCPP` classes instead.

Examples

```
generateMsg(3160, XRD_ERROR, "Logon ID is invalid.", 0);
```

See also

See also the description of the `generateMsg()` method under `BOHandlerCPP` Class and `GenGlobals` Class.

Deprecated methods

Methods in the `CxMsgFormat` class were supported in earlier versions but are no longer supported. These *deprecated methods* will not generate errors, but IBM recommends that you avoid their use and migrate existing code to the new methods. The deprecated methods might be removed in a future release.

Table 105 lists the deprecated methods for the `CxMsgFormat` class. If you are writing a new connector (not modifying an existing connector), you can ignore this section.

Table 105. Deprecated methods of the CxMsgFormat class

Former method	Replacement
`generateMsg()`	generateMsg() in the `GenGlobals` class generateMsg() in the `BOHandlerCPP` class

Chapter 16. CxVersion class

The CxVersion class represents business object versions. You can use methods of this class to set and retrieve version information. The header file for this class is CxVersion.hpp. It resides in the following subdirectory of your product directory:

```
DevelopmentKits\cdk\generic_include
```

Table 106 summarizes the methods in the CxVersion class.

Table 106. Member methods of the CxVersion class

Member method	Description	Page
CxVersion()	Creates a version object.	273
compareMajor()	Compares the major version with an object or an integer.	274
compareMinor()	Compares the minor version with an object or an integer.	274
comparePoint()	Compares the point versions with an object or an integer.	275
compareTo()	Compares version with an object or an integer.	275
getDELIMITER()	Retrieves the delimiter character that separates major, minor, and point versions in an version object.	276
getLATESTVERSION()	Retrieves the latest version object.	276
getMajorVer()	Retrieves the major version.	277
getMinorVer()	Retrieves the minor version.	277
getPointVer()	Retrieves the point version.	278
setMajorVer()	Sets the major version.	278
setMinorVer()	Sets the minor version.	278
setPointVer()	Sets the point version of a version object.	279
toString()	Converts a version object or integers to a character string.	279

CxVersion()

Creates a version object from major, minor, and point numbers or from a version string.

Syntax

```
CxVersion(int major, int minor, int point);
CxVersion(char * verString);
```

Parameters

major **[in]** Is an integer that specifies the major version, which precedes the first delimiter in the version.

minor **[in]** Is an integer that specifies the minor version, which follows the first delimiter in the version.

point **[in]** Is an integer that specifies the point version, which follows the second delimiter in the version.

verString **[in]** Is a character string that specifies a version, such as “2.0.3”.

Return values

None.

Notes

Each business object definition refers to an instance of the `CxVersion` class to set its version. The version to which a `BusinessObject` constructor refers is the version of the business object definition. A business object has the same version as its business object definition.

Examples

```
CxVersion * CurrentVersion =
   new CxVersion(LATEST, LATEST, LATEST);
. . .
myVersion = new CxVersion("2.0.3");
```

See also

See also the description of the `BusObjSpec` class.

---

compareMajor()

Compares major versions or compares the major version of an version object with an integer.

Syntax

```
int compareMajor(CxVersion & other);
int compareMajor(int major);
```

Parameters

other **[in]** Is a version object.

major **[in]** Is an integer that represents the major version number.

Return values

An integer that specifies the difference between the major versions.

Notes

You can use the `compareMajor()` method to compare the major versions of business objects or business object definitions.

See also

See also the descriptions of the `compareMinor()`, `comparePoint()`, and `compareTo()` methods.

---

compareMinor()

Compares the minor versions of two version objects or compares the minor version with an integer.

Syntax

```
int compareMinor(CxVersion & other);
int compareMinor(int minor);
```

Parameters

other **[in]** Is a version object.

minor **[in]** Is an integer that represents the minor version number.

Return values

An integer that specifies the difference between the minor versions.

Notes

You can use the `compareMinor()` method to compare the minor versions of business objects or business object definitions.

See also

See also the descriptions of the `compareMajor()`, `comparePoint()`, and `compareTo()` methods.

---

comparePoint()

Compares the point versions of two version objects or compares the point version with an integer.

Syntax

```
int comparePoint(CxVersion & other);
int comparePoint(int point);
```

Parameters

other **[in]** Is a version object.

point **[in]** Is an integer.

Return values

An integer that specifies the difference between the point versions.

Notes

You can use the `comparePoint()` method to compare the point versions of business objects or business object definitions.

See also

See also the descriptions of the `compareMajor()`, `compareMinor()`, and `compareTo()` methods.

---

compareTo()

Compares the current version to another version or compares the current version to a character string.

Syntax

```
int compareTo(CxVersion & other);
int compareTo(char * verString);
```

Parameters

other **[in]** Is a version object.

verString **[in]** Is a character string that specifies a version.

Return values

Returns 0 if the versions match, or an integer that specifies the difference between the first of the following versions that do not match: major versions, minor versions, or point versions.

Notes

You can use the `compareTo()` method to determine whether a business object or business object definition has a certain version.

Examples

```
compareTo(newVersion);
```

See also

See also the descriptions of the `compareMajor()`, `compareMinor()`, and `compareTo()` methods.

getDELIMITER()

Retrieves the delimiter character that separates major, minor, and point versions in a version object.

Syntax

```
static char * getDELIMITER();
```

Parameters

None.

Return values

The current delimiter character for separating major, minor, and point versions in a version object.

Notes

You can use the `getDELIMITER()` method to determine which characters version strings might contain.

See also

See also the descriptions of the `getLATESTVERSION()` and `toString()` methods.

getLATESTVERSION()

Retrieves the latest version of a version object.

Syntax

```
static const CxVersion & getLATESTVERSION();
```

Parameters

None.

Return values

The latest version, including major, minor, and point versions separated by the current delimiter character (default period).

Notes

You can use the `compareTo()` method to determine whether a business object or business object definition has the latest version.

Examples

```
if(pObj->getVersion()->compareTo(CxVersion::getLATESTVERSION)==0)
```

See also

See also the descriptions of the `getMajorVer()`, `getMinorVer()`, and `getPointVer()` methods.

getMajorVer()

Retrieves the major version of a version object.

Syntax

```
int getMajorVer() const;
```

Parameters

None.

Return values

An integer that specifies the major version in the latest version object.

See also

See also the descriptions of the `getLATESTVERSION()`, `getMinorVer()`, and `getPointVer()` methods.

getMinorVer()

Retrieves the minor version of a version object.

Syntax

```
int getMinorVer() const;
```

Parameters

None.

Return values

An integer that specifies the minor version in the latest version.

See also

See also the descriptions of the `getLATESTVERSION()`, `getMajorVer()`, and `getPointVer()` methods.

getPointVer()

Retrieves the point version of a version object.

Syntax

```
int getPointVer() const;
```

Parameters

None.

Return values

An integer that specifies the point version in the latest version.

See also

See also the descriptions of the `getLATESTVERSION()`, `getMajorVer()`, and `getMinorVer()` methods.

setMajorVer()

Sets the major version in a version object.

Syntax

```
void setMajorVer(int newMajorVer);
```

Parameters

newMajorVer **[in]**
Is an integer that specifies the major version.

Return values

None.

See also

See also the descriptions of the `setMinorVer()` and `setPointVer()` methods.

setMinorVer()

Sets the minor version of a version object.

Syntax

```
void setMinorVer(int newMinorVer);
```

Parameters

newMinorVer **[in]**
Is an integer that specifies the minor version.

Return values

None.

See also

See also the descriptions of the `setMajorVer()` and `setPointVer()` methods.

---

setPointVer()

Sets the point version of a version object.

Syntax

```
void setPointVer(int newPointVer);
```

Examples

newPointVer **[in]**
Is an integer that specifies the point version.

Return values

None.

See also

See also the descriptions of the `setMajorVer()` and `setMinorVer()` methods.

---

toString()

Converts a version object to a character string or converts major, minor, and point versions to a character string.

Syntax

```
char * toString();
static char * toString(int major, int minor, int point);
```

Parameters

major **[in]** Is an integer that specifies a major version, which precedes the first delimiter in a version.

minor **[in]** Is an integer that specifies a minor version, which follows the first delimiter in a version.

point **[in]** Is an integer that specifies a point version, which follows the second delimiter in a version.

Return values

A character string that concatenates the major, minor, and point version numbers, from a version object or from major, minor, and point numbers that you specify. The delimiter character separates the major, minor, and point numbers in the character string.

See also

See also the description of the CxVersion constructors.

Chapter 17. GenGlobals class

The GenGlobals class is a base class for a C++ connector. From this class, a connector developer must derive a connector class and implement the virtual methods for the connector. This connector class contains the code for the application-specific component of the connector.

Important: All C++ connectors *must* extend this connector base class, which contains the following virtual methods: `init()`, `getVersion()`, `getBOHandlerforBO()`, `pollForEvents()`, and `terminate()`. In their derived connector class, developers *must* provide implementations for these virtual methods.

The header file for this class is `GenGlobals.hpp`. It resides in the following subdirectory of your product directory:

```
DevelopmentKits\cdk\generic_include
```

Table 107 summarizes the methods in the GenGlobals class.

Table 107. Member methods of the GenGlobals class

Member method	Description	Page
`GenGlobals()`	Creates an instance of the GenGlobals class.	281
`executeCollaboration()`	Send business object request to a collaboration.	282
`generateAndLogMsg()`	Generates a message from a message file and sends it to the connector's log destination.	283
`generateAndTraceMsg()`	Generates a trace message from a message file and sends it to the connector's log destination.	284
`generateMsg()`	Generates a message from a message file that you provide.	285
`getBOHandlerforBO()`	Retrieves the handler for a business object.	286
`getCollabNames()`	Retrieves a list of collaboration names that are available to process business object requests.	287
`getConfigProp()`	Retrieves a property for the connector from the repository.	287
`getTheSubHandler()`	Retrieves a subscription handler to determine which collaborations subscribe to the business object definition for the incoming business object.	290
`getVersion()`	Retrieves the version of the application-specific component of the connector framework.	290
`init()`	Establishes a connection with the application.	291
`isAgentCapableOfPolling()`	Determines whether this connector-agent process can perform polling.	292
`logMsg()`	Logs a message.	293
`pollForEvents()`	Polls an application for changes to business objects.	294
`terminate()`	Closes the connection with the application and frees allocated resources.	294
`traceWrite()`	Writes a trace message.	295

GenGlobals()

Creates an instance of the GenGlobals class. You should derive a connector class from the GenGlobals base class and implement all the virtual methods that are in the GenGlobals class.

Syntax

```
GenGlobals();
```

Parameters

None.

Return values

None.

Examples

```
XYZGlobal::XYZGlobal() : GenGlobals()
```

executeCollaboration()

Sends a business object request to the connector framework, which sends it to a business process within the integration broker. This is a synchronous request.

Syntax

```
void executeCollaboration (char * busProcName, BusinessObject & busObj,
   ReturnStatusDescriptor & rtnStatusDesc);
```

Parameters

*busProcName***[in]**
: Specifies the name of the business process to execute the business object request. If InterChange Server is your integration broker, the business-process name is the name of a collaboration.Use the `getCollabNames()` method to determine the names of collaborations that are available to process business object requests.

*busObj***[in/out]**
: Is the triggering event and the business object returned from the business process.

*retStatusDesc***[out]**
: Is the return-status descriptor containing a message and status from the business process.

Return values

None.

Notes

The `executeCollaboration()` method sends the *busObj* business object to the connector framework. The connector framework does some processing on the event object to serialize the data and ensure that it is persisted properly. It then sends the event to the *busProcName* business process in the integration broker. This method initiates a synchronous execution of an event, which means that the method waits for a response from the integration broker's business process.

WebSphere InterChange Server

If your integration broker is IBM WebSphere InterChange Server, the business process that `executeCollaboration()` invokes is a collaboration.

To receive status information about the business-process execution, pass in an instantiated return-status descriptor, *rtnStatusDesc*, as the last argument to the method. The integration broker can return status information from its business process and send it to the connector framework, which populates this return-status descriptor with it. You can use the methods of the `ReturnStatusDescriptor` class to access this status information.

Note: To initiate an asynchronous execution of an event, use the `gotApplEvent()` method. Asynchronous execution means that the calling code does *not* wait for the receipt of the event, nor does it wait for a response.

See also

See also the descriptions of the `BusinessObject` and `ReturnStatusDescriptor` classes.

---

generateAndLogMsg()

Generates a message and sends it to the connector's log destination.

Syntax

```
void generateAndLogMsg(int msgNum, int msgType, int argCount, ...);
```

Parameters

msgNum **[in]** Specifies the message number (identifier) in the message file.

msgType **[in]** Is one of the following message-type constants defined in the `CxMsgFormat` class to identify the message severity:

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

argCount **[in]** Is an integer that specifies the number of parameters within the message text.

... **[in]** Is a list of message parameters for the message text.

Return values

None.

Notes

The `generateAndLogMsg()` method combines the functionality of the `generateMsg()` and `logMsg()` methods. It generates a message from a message file and then sends it to the log destination. You establish the name of a connector's log destination through the Logging section in the Trace/Log File tab of Connector Configurator.

Important: By combining these two methods, `generateAndLogMsg()` frees up the memory required for the message string that `generateMsg()` produces. You no longer need to include the call to the `freeMemory()` method to release the memory allocated for the message string.

WebSphere InterChange Server

If `severity` is `XRD_ERROR` or `XRD_FATAL` and the connector configuration property `LogAtInterchangeEnd` is set, the error message is logged and an email notification is sent when email notification is on. See the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set for information on how to set up email notification for errors.

IBM recommends that log messages be contained in a message file and extracted with the `generateAndLogMsg()` method. This message file should be the connector message file, which contains messages specific to your connector.

Connector messages logged with `generateAndLogMsg()` are viewable using LogViewer.

Examples

The following example performs the same task as the example provides for the `generateMsg()` method:

```
ret_code = connect_to_app(userName, password);
// Message 1100 - Failed to connect to application
if (ret_code == -1) {
   msg = generateAndLogMsg(1100, CxMsgFormat::XRD_ERROR, 0, NULL);
   return BON_FAIL;
}
```

See also

generateAndTraceMsg(), generateMsg(), logMsg()

generateAndTraceMsg()

Generates a trace message and sends it to the connector's trace destination.

Syntax

```
void generateAndTraceMsg(int msgNum, int msgType, int traceLevel,
   int argCount, ...);
```

Parameters

msgNum **[in]** Specifies the message number (identifier) in the message file.

msgtype **[in]** Is one of the following message-type constants defined in the `CxMsgFormat` class:

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

traceLevel **[in]** Is one of the following trace-level constants defined in the `Tracing` class to identify the trace level used to determine which trace messages to output:

```
Tracing::LEVEL1
Tracing::LEVEL2
Tracing::LEVEL3
Tracing::LEVEL4
Tracing::LEVEL5
```

The method writes the trace message when the current trace level is greater than or equal to `traceLevel`.

Note: Do not specify a trace level of zero (`LEVEL0`) with a tracing message. A trace level of zero indicates that tracing is turned off. Therefore, any trace message associated with a *`traceLevel`* of `LEVEL0` will never print.

argCount **[in]** Is an integer that specifies the number of parameters within the message text.

... **[in]** Is a list of message parameters for the message text.

Return values

None.

Notes

The `generateAndTraceMsg()` method combines the message generating and tracing functionality of `generateMsg()` and `traceWrite()`, respectively. It generates a message from a message file and then sends it to the trace destination. You establish the name of a connector's trace destination through the Tracing section in the Trace/Log File tab of Connector Configurator.

Important: By combining these two methods, `generateAndTraceMsg()` frees up the memory required for the message string that `generateMsg()` produces. You no longer need to include the call to the `freeMemory()` method to release the memory allocated for the message string.

Connector messages logged with `generateAndTraceMsg()` are *not* viewable using LogViewer.

Examples

```
if(tracePtr->getTraceLevel()>= Tracing::LEVEL3) {
   // Message 3033 - Opened main form for object
   msg = generateAndTraceMsg(3033,CxMsgFormat::XRD_FATAL,
      Tracing::LEVEL3,0, NULL);
}
```

See also

generateAndLogMsg(), generateMsg(), traceWrite()

---

generateMsg()

Generates a message from a set of predefined messages in a message file.

Syntax

```
char * generateMsg(int msgNum, int msgType, char * info,
      int argCount, ...);
```

Parameters

msgNum **[in]** Specifies the message number (identifier) in the message file.

msgType **[in]** Is one of the following message-type constants defined in the `CxMsgFormat` class:

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

info **[in]** Is an informational value, such as the name of the class for which the IBM WebSphere business integration system generated the message.

argCount **[in]** Is an integer that specifies the number of parameters within the message text.

... **[in]** Is a list of parameters for the message text.

Return values

A character pointer to the generated message.

Notes

The `generateMsg()` method allocates memory to store a generated message. When the connector has logged the message, it should call the `freeMemory()` method to release the allocated memory. This method is a member of the connector framework class `JToCPPVeneer`. The syntax of the call is:

```
void freeMemory(char * mem)
```

where `mem` is the memory allocated by `generateMsg()`. See the sample code below for an example of how to call this method.

Examples

```
char * msg;
ret_code = connect_to_app(userName, password);
// Message 1100 - Failed to connect to application
if (ret_code == -1) {
   msg = generateMsg(1100, CxMsgFormat::XRD_ERROR, NULL, 0, NULL);
   logMsg(msg);
   JToCPPVeneer::getTheHandlerStuff()->freeMemory(msg);
   return BON_FAIL;
}
```

---

getBOHandlerforBO()

Retrieves the business object handler for a business object definition.

Syntax

```
virtual BOHandlerCPP * getBOHandlerforBO(char * busObjName) = 0;
```

Parameters

busObjName **[in]**

Is the name of a business object.

Return values

A pointer to a business object handler.

Notes

The class library calls the `getBOHandlerforBO()` method to retrieve the business object handler for a business object definition.

Important: The `getBOHandlerforBO()` method is a virtual method that you *must* implement for the connector.

You can use one business object handler for multiple business object definitions or a business object handler for each business object definition.

Examples

```
BOHandlerCPP *AppGlobal::getBOHandlerforBO(char * BOName)
{
   static AppGlobal &pGlobal = NULL;
   if (NULL == pGlobal) {
      pGlobal = new AppGlobal();
   }
   return pGlobal;
}
```

See also

See also the description of the `BOHandlerCPP` class.

getCollabNames()

Retrieves the names of the collaborations that are available to process business object requests.

WebSphere InterChange Server
This method is only valid when the integration broker is InterChange Server.

Syntax

```
StringMessage & getCollabNames();
```

Return values

A `StringMessage` object containing a list of collaboration names.

Notes

The `getCollabNames()` method returns the collaboration names in a `StringMessage` object. Use the methods of this class to access the collaboration names. For more information, see Chapter 20, “StringMessage class,” on page 307

getConfigProp()

Retrieves a connector configuration property from the repository.

Syntax

```
int getConfigProp(char * property, char * val, int nMaxCount);
```

Parameters

property **[in]** Is the name of the property to retrieve.

val **[out]** Is a pointer to a buffer to which the method can write the property value.

nMaxCount **[in]**
Is the number of bytes in the value buffer.

Return values

An integer that specifies the number of bytes that the method copied to the value buffer.

Notes

When you call `getConfigProp("ConnectorName")` in a parallel-process connector (one that has the `ParallelProcessDegree` connector property set to a value greater than 1), the method always returns the name of the connector-agent master process, regardless of whether it is called in the master process or a slave process.

Examples

```
if (getConfigProp("LoginId", val, 255) == 0);
   {
   logMsg("Invalid LoginId");
   traceWrite(Tracing::LEVEL3, "Invalid LoginId", NULL);
   }
```

---

getEncoding()

Retrieves the character encoding that the connector framework is using.

Syntax

```
char * getEncoding();
```

Parameters

None.

Return values

A string containing the connector framework's character encoding.

Notes

The `getEncoding()` method retrieves the connector framework's locale, which defines cultural conventions for data according to language, country (or territory), and a character encoding. The connector framework's character encoding should indicate the character encoding of the connector application. The connector framework's locale is set using the following hierarchy:

- The `CharacterEncoding` connector configuration property in the repository

WebSphere InterChange Server

If a local configuration file exists, the setting of the `CharacterEncoding` connector configuration property in this local file takes precedence. If no local configuration file exists, the setting of the `CharacterEncoding` property is one from the set of connector configuration properties downloaded from the InterChange Server repository at connector startup.

- The character encoding from the Java environment, which is Unicode (UCS-2).

This method is useful when the connector needs to perform character-encoding processing, such as character conversion.

See also

getLocale()(in this class), getLocale() (in the `BusinessObject` class)

---

getLocale()

Retrieves the locale of the connector framework.

Syntax

```
char * getLocale();
```

Parameters

None.

Return values

A string containing the connector framework's locale setting.

Notes

The `getLocale()` method retrieves the connector framework's locale, which defines cultural conventions for data according to language, country (or territory), and a character encoding. The connector framework's locale should indicate the locale of the connector application. The connector framework's locale is set using the following hierarchy:

- The `Locale` connector configuration property in the repository

WebSphere InterChange Server

If a local configuration file exists, the setting of the `Locale` connector configuration property in this local file takes precedence. If no local configuration file exists, the setting of the `Locale` property is the one from the set of connector configuration properties downloaded from the InterChange Server repository at connector startup.

- The locale from the Java environment, which is the locale from the operating system.

This method is useful when the connector needs to perform locale-sensitive processing.

See also

getEncoding(), getLocale() (in the `BusinessObject` class)

getTheSubHandler()

Retrieves a pointer to the subscription manager. The calling routine can use this pointer to determine whether any subscriptions to a particular business object definition exist for the business object.

Syntax

```
SubscriptionHandlerCPP * getTheSubHandler() const;
```

Parameters

None.

Return values

A pointer to the subscription manager.

Notes

Through the subscription manager, the connector keeps track of the subscribers for every verb of each business object definition that the connector publishes, in a consolidated list of all active subscriptions.

Examples

```
if (getTheSubHandler->isSubscribed(theObj->getName(), "Create"){
}
```

See also

See also the description of the SubscriptionHandlerCPP, BusinessObject, and BusObjSpec classes.

getVersion()

Retrieves the version of the connector.

Syntax

```
CxVersion * getVersion() = 0;
```

Parameters

None.

Return values

A pointer to a character string indicating the version of the connector's application-specific component.

Examples

```
char * getVersion()

{
return (char *) CX_CONNECTOR_VERSIONSTRING;
}
```

Notes

The connector framework calls the `getVersion()` method to retrieve the version of the connector.

Important: The `getVersion()` method is a virtual method that you *must* implement for the connector.

See also

See also the description of the CxVersion class.

---

init()

Initializes the connector's application-specific component.

Syntax

```
virtual int init(CxVersion * version) = 0;
```

Parameters

*version***[in]** Is the version object of the connector framework.

Return values

An integer that indicates the status of the initialization operation. Typical return values are:

`BON_SUCCESS` Initialization succeeded.

`BON_FAIL` Initialization failed.

`BON_UNABLETOLOGIN`
The connector is unable to log in to the application.

For other return values, see "doVerbFor()" on page 232.

Notes

The class library calls the `init()` method when the connector comes up. Be sure to implement all of the initialization for the connector, such as logging on to an application, in the `init()` method.

Important: The `init()` method is a virtual method that you *must* implement for the connector.

As part of the initialization, the `init()` method can optionally compare the version of the connector framework with the version that it expects and return success if the versions match, or failure if the connector cannot work with the version of the connector framework.

See also

See also the description of the GenGlobals class.

isAgentCapableOfPolling()

Determines whether a connector-agent process is capable of polling.

WebSphere InterChange Server
This method is only valid when the integration broker is InterChange Server.

Syntax

```
boolean isAgentCapableOfPolling();
```

Parameters

None.

Return values

A `boolean` value that indicates whether the connector is capable of polling. This return value depends on the type of connector:

Connector type	Return value
Master (serial processing)	`true`
Master (parallel processing)	`false`
Slave (request)	`false`
Slave (polling)	`true`

Notes

If a connector is configured to run in a single-process mode (with `ParallelProcessDegree` equal to 1, which is the default), the `isAgentCapableOfPolling()` method always returns `true` because the same connector process performs both event polling and request processing.

If a connector is configured to run in parallel-process mode (`ParallelProcessDegree` is greater than 1), it consists of several processes, each with a particular purpose, as shown in Table 108.

Table 108. Purposes of processes of a parallel-process connector

Connector process type	Purpose of connector process
Connector-agent master process	Receives the incoming event from ICS and determines to which of the connector's slave processes to route the event
Request-processing slave process	Handles requests for the connector
Polling slave process	Handles polling and event delivery for the connector

The return value of `isAgentCapableOfPolling()` depends on the purpose of the connector that makes the call to this method. For a parallel-process connector, this method returns `true` *only* when called from a connector whose purpose is to serve

as a polling slave. For more information on parallel-process connectors, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set.

logMsg()

Logs a message to the connector's log destination. Log messages must be contained in a message file that you provide for your connector.

Syntax

```
void logMsg(char * msg);
void logMsg(char * msg, int severity);
```

Parameters

msg **[in]** Is a pointer to the message.

*severity***[in]** Is one of the following message types:

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

Return values

None.

Notes

The `logMsg()` method sends the specified *msg* text to the log destination. You establish the name of a connector's log destination through the Logging section in the Trace/Log File tab of Connector Configurator.

IBM recommends that log messages be contained in a message file and extracted with the `generateMsg()` method. This message file should be the connector message file, which contains messages specific to your connector. The `generateMsg()` method generates the message string for `logMsg()`. It retrieves a predefined message from a message file, formats the text, and returns a generated message string.

Note: You can use the `generateAndLogMsg()` method to combine the message generation and logging steps.

WebSphere InterChange Server

If `severity` is `XRD_ERROR` or `XRD_FATAL` and the connector configuration property `LogAtInterchangeEnd` is set, the error message is logged and an email notification is sent when email notification is on. See the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set for information on how to set up email notification for errors.

Connector messages logged with `logMsg()` are viewable using LogViewer if the message strings were generated with `generateMsg()`.

Examples

```
if ((form = CreateMainForm(conn, getFormName(theObj))) < 0) {
   msg = generateMsg(10, CxMsgFormat::XRD_FATAL, NULL, 0, NULL);
   logMsg(msg);
}
```

See also

See the description of the `GenGlobals::generateMsg()` utility.

---

pollForEvents()

Polls an application for changes to business objects.

Syntax

```
virtual int pollForEvents() = 0;
```

Parameters

None.

Return values

An integer that indicates the outcome status of the polling operation. The following return codes are typically used by the `pollForEvents()` method.

`BON_SUCCESS` The poll action was successful.

`BON_FAIL` The method failed in polling.

`BON_APPRESPONSETIMEOUT`
The application is not responding.

For other return values, see “doVerbFor()” on page 232..

Notes

The connector infrastructure calls the `pollForEvents()` method, at a time interval that you can configure, so that the connector can detect any event in the application that is interesting to a subscriber. The frequency at which the class library calls this method depends on the poll frequency value that is configured by the `PollFrequency` connector configuration property.

Important: The `pollForEvents()` method is an abstract method that you must implement for the connector.

Note: If your connector executes in parallel-process mode, it uses a separate polling slave process to handle polling.

See also

See also the description of the `SubscriptionHandlerCPP` class.

---

terminate()

Performs clean-up operations when the connector is shutting down.

Syntax

```
virtual int terminate() = 0;
```

Parameters

None.

Return values

An integer that indicates the status value of the `terminate()` operation. Typical return values are:

`BON_SUCCESS` Termination succeeded.

`BON_FAIL` Termination failed.

For other return values, see "doVerbFor()" on page 232.

Notes

The connector framework calls the `terminate()` method when the connector is shutting down. In your implementation of this method, be sure to free all the memory and log off from the application.

Important: The `terminate()` method is a virtual method that you *must* implement for the connector.

---

traceWrite()

Writes a trace message to the log destination. This is a utility method for connector developers to use.

Syntax

```
void traceWrite(int traceLevel, char * info,
      char * filterName);
```

Parameters

traceLevel **[in]** Is one of the following trace-level constants to identify the trace level used to determine which trace messages are output:

```
Tracing::LEVEL1
Tracing::LEVEL2
Tracing::LEVEL3
Tracing::LEVEL4
Tracing::LEVEL5
```

The method writes the trace message when the current trace level is greater than or equal to *traceLevel*.

Note: Do not specify a trace level of zero (`LEVEL0`) with a tracing message. A trace level of zero indicates that tracing is turned off. Therefore, any trace message associated with a *traceLevel* of `LEVEL0` will never print.

info **[in]** Is a pointer to the message text to use for the trace message.

filterName **[in]** Is a pointer to a filter to use for writing the message. Specify `NULL` for this parameter.

Return values

None.

Notes

You can use the `traceWrite()` method to write your own trace messages for a connector. Tracing is turned on for the connector when the `TraceLevel` connector configuration property is set to a nonzero value (any trace-level constant *except* `LEVEL0`).

The `traceWrite()` method sends the specified *`msg`* text to the trace destination when the current trace level is greater than or equal to *`level`*. You establish the name of a connector's trace destination through the Tracing section in the Trace/Log File tab of Connector Configurator.

Because trace messages are usually needed only during debugging, whether trace messages are contained in a message file is left at the discretion of the developer:

- If non-English-speaking users need to view trace messages, you need to internationalize these messages. Therefore, you must put the trace messages in a message file and extract them with the `generateMsg()` method. This message file should be the connector message file, which contains message specific to your connector. The `generateMsg()` method generates the message string for `traceWrite()`. It retrieves a predefined trace message from a message file, formats the text, and returns a generated message string.

  Note: You can use the `generateAndTraceMsg()` method to combine the message generation and logging steps.
- If only English-speaking users need to view trace messages, you do not need to internationalize these messages. Therefore, you can include the trace message (in English) directly in the call to `traceWrite()`. You do *not* need to use the `generateMsg()` or `generateAndTraceMsg()` method.

Connector messages logged with `traceWrite()` are *not* viewable using LogViewer.

Examples

```
traceWrite(Tracing::LEVEL3, "Invalid LoginId", NULL);
```

See also

generateAndTraceMsg(), generateMsg()

See also the description of the Tracing class.

Deprecated methods

Some methods in the `GenGlobals` class were supported in earlier versions but are no longer supported. These *deprecated methods* will not generate errors, but IBM recommends that you avoid their use and migrate existing code to the new methods. The deprecated methods might be removed in a future release.

Table 109 lists the deprecated methods for the `GenGlobals` class. If you are writing a new connector (not modifying an existing connector), you can ignore this section.

Table 109. Deprecated methods of the GenGlobals class

Former method	Replacement
`consumeSync()`	`executeCollaboration()`

Chapter 18. ReturnStatusDescriptor class

The `ReturnStatusDescriptor` class enables connectors to return error and informational messages in a return-status descriptor. This descriptor provides additional status information is usually returned as part of the request response sent to the integration broker.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector framework returns the return-status descriptor to the collaboration that initiated the request. The collaboration can access the information in this return-status descriptor to obtain the status of its service call request.

The header file for this class is `ReturnStatusDescriptor.hpp`. It resides in the following subdirectory of your product directory:

```
DevelopmentKits\cdk\generic_include
```

Table 110 summarizes the methods in the `ReturnStatusDescriptor` class.

Table 110. Member methods of the ReturnStatusDescriptor class

Member method	Description	Page
`getErrorMsg()`	Retrieves an error message string from a return-status descriptor.	299
`getStatus()`	Retrieves the status value from a return-status descriptor.	299
`seterrMsg()`	Sets a string containing an error or informational message in the return-status descriptor.	300
`setStatus()`	Sets the status value in the return-status descriptor.	300

getErrorMsg()

Retrieves an error message string from a return-status descriptor.

Note: This method is used *only* by the connector framework.

Syntax

```
char * getErrorMsg();
```

Parameters

None.

Return values

A string containing an error or informational message for the integration broker.

getStatus()

Retrieves the status value from a return-status descriptor.

Syntax

```
int getStatus();
```

Parameters

None.

Return values

An integer containing status value for the integration broker.

seterrMsg()

Sets a string containing an error or informational message in the return-status descriptor.

Syntax

```
void seterrMsg(char * errMsg);
```

Parameters

*errMsg***[in]** Is the message string.

Return values

None.

Notes

You can use `seterrMsg()` to return a string containing a message to an integration broker.

Examples

```
int ExampleBOHandler::doVerbFor(BusinessObject &theObj,
   ReturnStatusDescriptor *rtnObj)
{
int status = BON_SUCCESS;
char *verb = theObj.getVerb();

if (strcmp(verb, CREATE) == 0)
   status = doCreate(theObj);
else if (strcmp(verb, Verb) == 0)
   // Check for other verbs and call verb routines
   else
   {
      // Send the collaboration a message that
      // this verb is not supported.
      char errorMsg[512];
      sprintf(errorMsg, "The verb '%s' is not supported ", verb);
      rtnObj->seterrMsg(errorMsg);
      status = BON_FAIL;
   }

return status;
}
```

setStatus()

Sets the status value in a return-status descriptor.

Syntax

```
void setStatus(int status);
```

Parameters

*status***[in]** Is the status value to store in the return-status descriptor.

Return values

None.

Chapter 19. SubscriptionHandlerCPP class

The SubscriptionHandlerCPP class represents subscription managers, which you can use to determine whether the integration broker is interested in a business object. You can also use a subscription manager method to send business objects to the integration broker.

The header file for this class is SubscriptionHandlerCPP.hpp. It resides in the following subdirectory of your product directory:

```
DevelopmentKits\cdk\generic_include
```

Table 111 summarizes the methods in the SubscriptionHandlerCPP class.

Table 111. Member methods of the SubscriptionHandlerCPP class

Member method	Description	Page
SubscriptionHandlerCPP()	Creates a subscription manager. In general, you do not call this method to create a subscription manager. The connector framework usually creates a subscription manager for the connector.	303
gotApplEvent()	Sends a business object to InterChange Server.	304
isSubscribed()	Determines whether a subscription exists for a business object definition.	306

SubscriptionHandlerCPP()

Creates a subscription manager, which is an instance of the SubscriptionHandlerCPP class.

Syntax

```
SubscriptionHandlerCPP();
```

Parameters

None.

Return values

None.

Notes

A business object handler uses a subscription manager to determine which whether subscriptions exist to a business object. In general, you do not use this method to create a subscription manager. The connector class framework creates a subscription manager for the connector.

See also

See also the description of the BOHandlerCPP Class.

gotApplEvent()

Sends a business object to the connector framework. This is an asynchronous request.

Syntax

```
int gotApplEvent(BusinessObject busObj);
```

Parameters

busObj **[in]** Is the business object being sent to the integration broker.

Return values

An integer that indicates the outcome status of the event delivery. Compare this integer value with the following outcome-status constants to determine the status:

`BON_SUCCESS` The connector framework successfully delivered the business object to the connector framework.

`BON_FAIL` The event delivery failed.

`BON_CONNECTOR_NOT_ACTIVE`
The connector is paused and therefore unable to receive events.

`BON_NO_SUBSCRIPTION_FOUND`
No subscriptions exist for the event that the business object represents.

Notes

The `gotApplEvent()` method sends the *busObj* business object to the connector framework. The connector framework does some processing on the event object to serialize the data and ensure that it is persisted properly. It then makes sure the event is either sent to the ICS through IIOP or written to a queue (if you are using queues for event notification).

Before sending the business object to the connector framework, `gotApplEvent()` checks for the following conditions and returns the associated outcome status if these conditions are not met:

Condition	Outcome status
Is the status of the connector active; that is, it is not in a "paused" state? When the connector's application-specific component is paused, it no longer polls the application.	`BON_CONNECTOR_NOT_ACTIVE`
Is there any subscription for the event?	`BON_NO_SUBSCRIPTION_FOUND`

Note: Because `gotApplEvent()` makes sure that the business object and verb to be sent have a valid subscription, you do *not* need to call `isSubscribed()` immediately before calling `gotApplEvent()`.

WebSphere InterChange Server

Usually, you call the `gotApplEvent()` method from the `pollForEvents()` thread. InterChange Server uses the `pollForEvents()` method to request the connector to send subscribed events to it. The connector uses the `gotApplEvent()` method to send business objects to the connector framework, which in turn routes them to InterChange Server in response.

The poll method should check the return code from `gotApplEvent()` to ensure that any errors that are returned are handled appropriately. For example, until the event delivery is successful, the poll method should *not* remove the event from the event table.

The `gotApplEvent()` method initiates an asynchronous execution of an event. Asynchronous execution means that the method does *not* wait for receipt of the event, nor does it wait for a response.

Note: To initiate a synchronous execution of an event, use the `executeCollaboration()` method. Synchronous execution means that the calling code waits for the receipt of the event, and for a response.

Examples

```
SubscriptionHandlerCPP * theSubHandler =
      GenGlobals::getTheSubHandler();

// Determine whether there are subscribers to the event
if (theSubHandler->isSubscribed(obj_name, obj_verb) != TRUE) {
   // log message
   // delete event from event table
   // add event to archive table
   continue;
}

// Prepare to retrieve data into the business object
pObj = new BusinessObject(obj_name);
pObj->setVerb("Retrieve");

// Set key in business object

// Call the business object handler doVerbFor()
// to retrieve data
if (pObj->doVerbFor() == BON_FAIL) {
   // Log error message if retrieve fails
   retcode = BON_FAIL;
   break;
}

// Call gotApplEvent() to send the business object
pObj->setVerb(obj_verb);
theSubHandler->gotApplEvent(*pObj);
if ((theSubHandler->gotApplEvent(*pBusObj)) == BON_FAIL) {
   // Log error message
   retcode = BON_FAIL;
   break;
}
```

See also

See also the description of the `BusinessObject` Class and the `pollForEvents()` method.

isSubscribed()

Determines whether the integration broker has subscribed to a particular business object with a particular verb.

Syntax

```
int isSubscribed(char * busObjName, char * verb);
```

Parameters

busObjName **[in]**
Is the name of a business object.

verb **[in]** Is the active verb for the business object.

Return values

Returns 1 for True if the integration broker is interested in receiving the specified business object and verb; otherwise, returns 0 for False.

Notes

WebSphere InterChange Server

If your business integration system uses InterChange Server, the poll method can determine if any collaboration subscribes to the *busObjName* business object with the specified *verb*. At initialization, the connector framework requests its subscription list from the connector controller. At runtime, the poll method can use `isSubscribed()` to query the connector framework to verify that some collaboration subscribes to a particular business object. The poll method can send the event only if some collaboration is currently subscribed.

Other integration brokers

If your business integration system uses WebSphere MQ Integrator Broker or WebSphere Application Server, the connector framework assumes that the integration broker is interested in *all* the connector's supported business objects. If the application-specific component uses the `isSubscribed()` method to query the connector framework about subscriptions for a particular business object, the method returns `0` (True) for *every* business object that the connector supports.

Examples

```
SubscriptionHandlerCPP &theSubHandler =
   GenGlobals::getTheSubHandler();
if (theSubHandler->isSubscribed(theObj->getName(), theObj->getVerb())) {
   theSubHandler->gotApplEvent(theObj);
}
```

Chapter 20. StringMessage class

The StringMessage class provides methods for accessing the contents of a StringMessage object. The header file for this class is StringMessage.hpp. It resides in the following subdirectory of your product directory:

```
DevelopmentKits\cdk\generic_include
```

Table 112 summarizes the methods in the StringMessage class.

Table 112. Member methods of the StringMessage class

Member method	Description	Page
hasMoreTokens()	Indicates whether a StringMessage object has more string tokens.	307
nextToken()	Returns the next string or NULL.	307

hasMoreTokens()

Determines whether there are more strings in a StringMessage object. You can use this method to loop through a StringMessage object.

Syntax

```
unsigned char hasMoreTokens();
```

Parameters

None.

Return values

Returns 1 if the StringMessage object has more strings and 0 if not.

nextToken()

Returns the next string (token) in a StringMessage object.

Syntax

```
char * nextToken();
```

Parameters

None.

Return values

Returns the next string in a StringMessage object or NULL if there are no more strings.

Deprecated methods

Methods in the `StringMessage` class were supported in earlier versions but are no longer supported. These *deprecated methods* will not generate errors, but IBM recommends that you avoid their use and migrate existing code to the new methods. The deprecated methods might be removed in a future release.

Table 113 lists the deprecated methods for the `StringMessage` class. If you are writing a new connector (not modifying an existing connector), you can ignore this section.

Table 113. Deprecated methods of the StringMessage class

Deprecated method	Replacement
`getCurrentSize()`	None
`initTokenizer()`	None

Chapter 21. Tracing class

The Tracing class provides tracing services for connectors. The header file for this class is Tracing.hpp. It resides in the following subdirectory of your product directory:

```
DevelopmentKits\cdk\generic_include
```

This class contains the following:

- "Trace-level constants"
- "Methods"

Trace-level constants

The Tracing class defines a the trace-level constants shown in Table 114..

Table 114. Trace-level constants defined in the Tracing class

Trace-level constant	Meaning
LEVEL0	Level 0 of tracing (tracing is off)
LEVEL1	Level 1 of tracing
LEVEL2	Level 2 of tracing
LEVEL3	Level 3 of tracing
LEVEL4	Level 4 of tracing
LEVEL5	Level 5 of tracing

Methods

Table 115 summarizes the methods in the Tracing class.

Table 115. Member methods of the Tracing class

Member method	Description	Page
Tracing()	Creates an instance of the Tracing class for a connector. You never call this method to create an instance of the Tracing class. The connector class framework creates an instance of the Tracing class for the connector.	
getIndent()	Retrieves a character value that specifies the indent for tracing messages.	309
getName()	Retrieves the name of the business object for which to write tracing messages.	310
getTraceLevel()	Retrieves the current tracing level.	310
setIndent()	Sets the indent for messages.	310
write()	Writes a tracing message.	311

getIndent()

Retrieves a character string that specifies the indent for trace messages.

Syntax

```
static char * getIndent();
```

Parameters

None.

Return values

A character string that specifies the indent for trace messages.

Examples

```
tempStr = theObj::getIndent();
```

getName()

Retrieves the name of the subsystem (connector name) to use in trace messages.

Syntax

```
char * getName() const;
```

Parameters

None.

Return values

A character string that contains the name of the subsystem being traced.

getTraceLevel()

Retrieves the current trace level. To set the tracing level, you can use the `TraceLevel` connector configuration property.

Syntax

```
int getTraceLevel() const;
```

Parameters

None.

Return values

An integer that indicates the current trace level:

```
Tracing::LEVEL0
Tracing::LEVEL1
Tracing::LEVEL2
Tracing::LEVEL3
Tracing::LEVEL4
Tracing::LEVEL5
```

Examples

```
if(getTraceLevel() > Tracing::LEVEL0)
   write(Tracing::LEVEL1, "Connector failed to initialize.", NULL);
```

setIndent()

Sets the indent that tracing uses to write trace messages.

Syntax

```
static void setIndent(char * newIndent);
```

Parameters

newIndent **[in]** Is a character string that specifies the indent for tracing messages.

Return values

None.

---

write()

Writes a trace message for the connector.

Note: For most trace messages, you can simply use the `traceWrite()` utility methods provided in the `GenGlobals` and `BOHandlerCPP` classes.

Syntax

```
void write(int traceLevel, char * info);

void write(int traceLevel, char * info,
   char * filterName);
```

Parameters

traceLevel **[in]** Is one of the following tracing levels, to use for writing the message:

```
Tracing::LEVEL1
Tracing::LEVEL2
Tracing::LEVEL3
Tracing::LEVEL4
Tracing::LEVEL5
```

Note: Do not specify a trace level of zero (`LEVEL0`) with a tracing message. A trace level of zero indicates that tracing is turned off. Therefore, any trace message associated with a *traceLevel* of `LEVEL0` will never print.

info **[in]** Is a character string that contains the text of the tracing message.

filterName **[in]** Is the name of a tracing filter.

Return values

None.

Examples

```
write(Tracing::LEVEL4, "Connector failed to initialize.", NULL);
```

See also

See also the description of the `traceWrite()` method under the `BOHandlerCPP` and `GenGlobals` classes.

Appendix A. Standard configuration properties for connectors

This appendix describes the standard configuration properties for the connector component of WebSphere Business Integration adapters. The information covers connectors running on the following integration brokers:

- WebSphere InterChange Server (ICS)
- WebSphere MQ Integrator, WebSphere MQ Integrator Broker, and WebSphere Business Integration Message Broker, collectively referred to as the WebSphere Message Brokers (WMQI).
- WebSphere Application Server (WAS)

Not every connector makes use of all these standard properties. When you select an integration broker from Connector Configurator, you will see a list of the standard properties that you need to configure for your adapter running with that broker.

For information about properties specific to the connector, see the relevant adapter user guide.

Note: In this document, backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes and follow the conventions for each operating system.

New and deleted properties

These standard properties have been added in this release.

New properties

- XMLNameSpaceFormat

Deleted properties

- RestartCount

Configuring standard connector properties

Adapter connectors have two types of configuration properties:

- Standard configuration properties
- Connector-specific configuration properties

This section describes the standard configuration properties. For information on configuration properties specific to a connector, see its adapter user guide.

Using Connector Configurator

You configure connector properties from Connector Configurator, which you access from System Manager. For more information on using Connector Configurator, refer to the Connector Configurator appendix.

Note: Connector Configurator and System Manager run only on the Windows system. If you are running the connector on a UNIX system, you must have a Windows machine with these tools installed. To set connector properties

for a connector that runs on UNIX, you must start up System Manager on the Windows machine, connect to the UNIX integration broker, and bring up Connector Configurator for the connector.

Setting and updating property values

The default length of a property field is 255 characters.

The connector uses the following order to determine a property's value (where the highest number overrides other values):

1. Default
2. Repository (only if WebSphere InterChange Server is the integration broker)
3. Local configuration file
4. Command line

A connector obtains its configuration values at startup. If you change the value of one or more connector properties during a run-time session, the property's **Update Method** determines how the change takes effect. There are four different update methods for standard connector properties:

- **Dynamic**
  The change takes effect immediately after it is saved in System Manager. If the connector is working in stand-alone mode (independently of System Manager), for example with one of the WebSphere message brokers, you can only change properties through the configuration file. In this case, a dynamic update is not possible.
- Component restart
  The change takes effect only after the connector is stopped and then restarted in System Manager. You do not need to stop and restart the application-specific component or the integration broker.
- Server restart
  The change takes effect only after you stop and restart the application-specific component and the integration broker.
- Agent restart (ICS only)
  The change takes effect only after you stop and restart the application-specific component.

To determine how a specific property is updated, refer to the **Update Method** column in the Connector Configurator window, or see the Update Method column in the Property Summary table below.

Summary of standard properties

Table 116 on page 315 provides a quick reference to the standard connector configuration properties. Not all the connectors make use of all these properties, and property settings may differ from integration broker to integration broker, as standard property dependencies are based on `RepositoryDirectory`.

You must set the values of some of these properties before running the connector. See the following section for an explanation of each property.

Table 116. Summary of standard configuration properties

Property name	Possible values	Default value	Update method	Notes
AdminInQueue	Valid JMS queue name	*CONNECTORNAME* /ADMININQUEUE	Component restart	Delivery-Transport is JMS
AdminOutQueue	Valid JMS queue name	*CONNECTORNAME*/ADMINOUTQUEUE	Component restart	Delivery-Transport is JMS
AgentConnections	1-4	1	Component restart	Delivery-Transport is MQ or IDL: Repository-Directory is <REMOTE>
AgentTraceLevel	0-5	0	Dynamic	
ApplicationName	Application name	Value specified for the connector application name	Component restart	
BrokerType	ICS, WMQI, WAS			
CharacterEncoding	ascii7, ascii8, SJIS, Cp949, GBK, Big5, Cp297, Cp273, Cp280, Cp284, Cp037, Cp437 **Note:** This is a subset of supported values.	ascii7	Component restart	
ConcurrentEventTriggeredFlows	1 to 32,767	1	Component restart	Repository-Directory is <REMOTE>
ContainerManagedEvents	No value or JMS	No value	Component restart	Delivery-Transport is JMS
ControllerStoreAndForwardMode	true or false	True	Dynamic	Repository-Directory is <REMOTE>
ControllerTraceLevel	0-5	0	Dynamic	Repository-Directory is <REMOTE>
DeliveryQueue		*CONNECTORNAME*/DELIVERYQUEUE	Component restart	JMS transport only
DeliveryTransport	MQ, IDL, or JMS	JMS	Component restart	If Repository-Directory is local, then value is JMS only
DuplicateEventElimination	True or False	False	Component restart	JMS transport only: Container-ManagedEvents must be <NONE>
FaultQueue		*CONNECTORNAME*/FAULTQUEUE	Component restart	JMS transport only
jms.FactoryClassName	CxCommon.Messaging.jms .IBMMQSeriesFactory or CxCommon.Messaging .jms.SonicMQFactory or any Java class name	CxCommon.Messaging. jms.IBMMQSeriesFactory	Component restart	JMS transport only

Table 116. Summary of standard configuration properties (continued)

Property name	Possible values	Default value	Update method	Notes
jms.MessageBrokerName	If FactoryClassName is IBM, use crossworlds.queue.manager. If FactoryClassName is Sonic, use localhost:2506.	crossworlds.queue.manager	Component restart	JMS transport only
jms.NumConcurrentRequests	Positive integer	10	Component restart	JMS transport only
jms.Password	Any valid password		Component restart	JMS transport only
jms.UserName	Any valid name		Component restart	JMS transport only
JvmMaxHeapSize	Heap size in megabytes	128m	Component restart	Repository-Directory is <REMOTE>
JvmMaxNativeStackSize	Size of stack in kilobytes	128k	Component restart	Repository-Directory is <REMOTE>
JvmMinHeapSize	Heap size in megabytes	1m	Component restart	Repository-Directory is <REMOTE>
ListenerConcurrency	1- 100	1	Component restart	Delivery-Transport must be MQ
Locale	en_US, ja_JP, ko_KR, zh_CN, zh_TW, fr_FR, de_DE, it_IT, es_ES, pt_BR **Note:** This is a subset of the supported locales.	en_US	Component restart	
LogAtInterchangeEnd	True or False	False	Component restart	Repository-Directory must be <REMOTE>
MaxEventCapacity	1-2147483647	2147483647	Dynamic	Repository-Directory must be <REMOTE>
MessageFileName	Path or filename	InterchangeSystem.txt	Component restart	
MonitorQueue	Any valid queue name	*CONNECTORNAME*/MONITORQUEUE	Component restart	JMS transport only: DuplicateEvent-Elimination must be True
OADAutoRestartAgent	True or False	False	Dynamic	Repository-Directory must be <REMOTE>
OADMaxNumRetry	A positive number	1000	Dynamic	Repository-Directory must be <REMOTE>
OADRetryTimeInterval	A positive number in minutes	10	Dynamic	Repository-Directory must be <REMOTE>
PollEndTime	HH:MM	HH:MM	Component restart	

Table 116. Summary of standard configuration properties (continued)

Property name	Possible values	Default value	Update method	Notes
PollFrequency	A positive integer in milliseconds no (to disable polling) key (to poll only when the letter p is entered in the connector's Command Prompt window)	10000	Dynamic	
PollQuantity	1-500	1	Agent restart	JMS transport only: Container-ManagedEvents is specified
PollStartTime	HH:MM(HH is 0-23, MM is 0-59)	HH:MM	Component restart	
RepositoryDirectory	Location of metadata repository		Agent restart	For ICS: set to <REMOTE>; For WebSphere MQ message brokers and WAS: set to C:\crossworlds\repository
RequestQueue	Valid JMS queue name	*CONNECTORNAME*/REQUESTQUEUE	Component restart	Delivery-Transport is JMS
ResponseQueue	Valid JMS queue name	*CONNECTORNAME*/RESPONSEQUEUE	Component restart	Delivery-Transport is JMS: required only if Repository-Directory is <REMOTE>
RestartRetryCount	0-99	3	Dynamic	
RestartRetryInterval	A sensible positive value in minutes: 1 - 2147483547	1	Dynamic	
RHF2MessageDomain	mrm, xml	mrm	Component restart	Only if Delivery-Transport is JMS and WireFormat is CwXML.
SourceQueue	Valid WebSphere MQ name	*CONNECTORNAME*/SOURCEQUEUE	Agent restart	Only if Delivery-Transport is JMS and Container-ManagedEvents is specified
SynchronousRequestQueue		*CONNECTORNAME*/SYNCHRONOUSREQUESTQUEUE	Component restart	Delivery-Transport is JMS
SynchronousRequestTimeout	0 - any number (millisecs)	0	Component restart	Delivery-Transport is JMS

Table 116. Summary of standard configuration properties (continued)

Property name	Possible values	Default value	Update method	Notes
SynchronousResponseQueue		*CONNECTORNAME*/SYNCHRONOUSRESPONSEQUEUE	Component restart	Delivery-Transport is JMS
WireFormat	CwXML, CwBO	CwXML	Agent restart	CwXML if Repository-Directory is not <REMOTE>: CwBO if Repository-Directory is <REMOTE>
WsifSynchronousRequest Timeout	0 - any number (millisecs)	0	Component restart	WAS only
XMLNameSpaceFormat	short, long	short	Agent restart	WebSphere MQ message brokers and WAS only

Standard configuration properties

This section lists and defines each of the standard connector configuration properties.

AdminInQueue

The queue that is used by the integration broker to send administrative messages to the connector.

The default value is CONNECTORNAME/ADMININQUEUE.

AdminOutQueue

The queue that is used by the connector to send administrative messages to the integration broker.

The default value is CONNECTORNAME/ADMINOUTQUEUE.

AgentConnections

Applicable only if RepositoryDirectory is <REMOTE>.

The AgentConnections property controls the number of ORB connections opened by orb.init[].

By default, the value of this property is set to 1. There is no need to change this default.

AgentTraceLevel

Level of trace messages for the application-specific component. The default is 0. The connector delivers all trace messages applicable at the tracing level set or lower.

ApplicationName

Name that uniquely identifies the connector's application. This name is used by the system administrator to monitor the WebSphere business integration system environment. This property must have a value before you can run the connector.

BrokerType

Identifies the integration broker type that you are using. The options are ICS, WebSphere message brokers (WMQI, WMQIB or WBIMB) or WAS.

CharacterEncoding

Specifies the character code set used to map from a character (such as a letter of the alphabet, a numeric representation, or a punctuation mark) to a numeric value.

Note: Java-based connectors do not use this property. A C++ connector currently uses the value `ascii7` for this property.

By default, a subset of supported character encodings only is displayed in the drop list. To add other supported values to the drop list, you must manually modify the `\Data\Std\stdConnProps.xml` file in the product directory. For more information, see the appendix on Connector Configurator.

ConcurrentEventTriggeredFlows

Applicable only if `RepositoryDirectory` is <REMOTE>.

Determines how many business objects can be concurrently processed by the connector for event delivery. Set the value of this attribute to the number of business objects you want concurrently mapped and delivered. For example, set the value of this property to `5` to cause five business objects to be concurrently processed. The default value is `1`.

Setting this property to a value greater than `1` allows a connector for a source application to map multiple event business objects at the same time and deliver them to multiple collaboration instances simultaneously. This speeds delivery of business objects to the integration broker, particularly if the business objects use complex maps. Increasing the arrival rate of business objects to collaborations can improve overall performance in the system.

To implement concurrent processing for an entire flow (from a source application to a destination application), you must:

- Configure the collaboration to use multiple threads by setting its `Maximum number of concurrent events` property high enough to use multiple threads.
- Ensure that the destination application's application-specific component can process requests concurrently. That is, it must be multi-threaded, or be able to use connector agent parallelism and be configured for multiple processes. Set the `Parallel Process Degree` configuration property to a value greater than `1`.

The `ConcurrentEventTriggeredFlows` property has no effect on connector polling, which is single-threaded and performed serially.

ContainerManagedEvents

This property allows a JMS-enabled connector with a JMS event store to provide guaranteed event delivery, in which an event is removed from the source queue and placed on the destination queue as a single JMS transaction.

The default value is No value.

When ContainerManagedEvents is set to JMS, you must configure the following properties to enable guaranteed event delivery:

- PollQuantity = 1 to 500
- SourceQueue = CONNECTORNAME/SOURCEQUEUE

You must also configure a data handler with the MimeType, DHClass, and DataHandlerConfigMOName (optional) properties. To set those values, use the **Data Handler** tab in Connector Configurator. The fields for the values under the Data Handler tab will be displayed only if you have set ContainerManagedEvents to JMS.

Note: When ContainerManagedEvents is set to JMS, the connector does *not* call its pollForEvents() method, thereby disabling that method's functionality.

This property only appears if the DeliveryTransport property is set to the value JMS.

ControllerStoreAndForwardMode

Applicable only if RepositoryDirectory is <REMOTE>.

Sets the behavior of the connector controller after it detects that the destination application-specific component is unavailable.

If this property is set to true and the destination application-specific component is unavailable when an event reaches ICS, the connector controller blocks the request to the application-specific component. When the application-specific component becomes operational, the controller forwards the request to it.

However, if the destination application's application-specific component becomes unavailable **after** the connector controller forwards a service call request to it, the connector controller fails the request.

If this property is set to false, the connector controller begins failing all service call requests as soon as it detects that the destination application-specific component is unavailable.

The default is true.

ControllerTraceLevel

Applicable only if RepositoryDirectory is <REMOTE>.

Level of trace messages for the connector controller. The default is 0.

DeliveryQueue

Applicable only if DeliveryTransport is JMS.

The queue that is used by the connector to send business objects to the integration broker.

The default value is CONNECTORNAME/DELIVERYQUEUE.

DeliveryTransport

Specifies the transport mechanism for the delivery of events. Possible values are MQ for WebSphere MQ, IDL for CORBA IIOP, or JMS for Java Messaging Service.

- If ICS is the broker type, the value of the DeliveryTransport property can be MQ, IDL, or JMS, and the default is IDL.
- If the RepositoryDirectory is a local directory, the value may only be JMS.

The connector sends service call requests and administrative messages over CORBA IIOP if the value configured for the DeliveryTransport property is MQ or IDL.

WebSphere MQ and IDL

Use WebSphere MQ rather than IDL for event delivery transport, unless you must have only one product. WebSphere MQ offers the following advantages over IDL:

- Asynchronous communication:
  WebSphere MQ allows the application-specific component to poll and persistently store events even when the server is not available.
- Server side performance:
  WebSphere MQ provides faster performance on the server side. In optimized mode, WebSphere MQ stores only the pointer to an event in the repository database, while the actual event remains in the WebSphere MQ queue. This saves having to write potentially large events to the repository database.
- Agent side performance:
  WebSphere MQ provides faster performance on the application-specific component side. Using WebSphere MQ, the connector's polling thread picks up an event, places it in the connector's queue, then picks up the next event. This is faster than IDL, which requires the connector's polling thread to pick up an event, go over the network into the server process, store the event persistently in the repository database, then pick up the next event.

JMS

Enables communication between the connector and client connector framework using Java Messaging Service (JMS).

If you select JMS as the delivery transport, additional JMS properties such as jms.MessageBrokerName, jms.FactoryClassName, jms.Password, and jms.UserName, appear in Connector Configurator. The first two of these properties are required for this transport.

Important: There may be a memory limitation if you use the JMS transport mechanism for a connector in the following environment:

- AIX 5.0
- WebSphere MQ 5.3.0.1
- When ICS is the integration broker

In this environment, you may experience difficulty starting both the connector controller (on the server side) and the connector (on the client side) due to memory use within the WebSphere MQ client. If your installation uses less than 768M of process heap size, IBM recommends that you set:

- The LDR_CNTRL environment variable in the CWSharedEnv.sh script.

  This script resides in the \bin directory below the product directory. With a text editor, add the following line as the first line in the CWSharedEnv.sh script:

  ```
  export LDR_CNTRL=MAXDATA=0x30000000
  ```

This line restricts heap memory usage to a maximum of 768 MB (3 segments * 256 MB). If the process memory grows more than this limit, page swapping can occur, which can adversely affect the performance of your system.

- The `IPCCBaseAddress` property to a value of 11 or 12. For more information on this property, see the *System Installation Guide for UNIX*.

DuplicateEventElimination

When you set this property to `true`, a JMS-enabled connector can ensure that duplicate events are not delivered to the delivery queue. To use this feature, the connector must have a unique event identifier set as the business object's **ObjectEventId** attribute in the application-specific code. This is done during connector development.

This property can also be set to `false`.

Note: When `DuplicateEventElimination` is set to `true`, you must also configure the `MonitorQueue` property to enable guaranteed event delivery.

FaultQueue

If the connector experiences an error while processing a message then the connector moves the message to the queue specified in this property, along with a status indicator and a description of the problem.

The default value is `CONNECTORNAME/FAULTQUEUE`.

JvmMaxHeapSize

The maximum heap size for the agent (in megabytes). This property is applicable only if the `RepositoryDirectory` value is <REMOTE>.

The default value is 128m.

JvmMaxNativeStackSize

The maximum native stack size for the agent (in kilobytes). This property is applicable only if the `RepositoryDirectory` value is <REMOTE>.

The default value is 128k.

JvmMinHeapSize

The minimum heap size for the agent (in megabytes). This property is applicable only if the `RepositoryDirectory` value is <REMOTE>.

The default value is 1m.

jms.FactoryClassName

Specifies the class name to instantiate for a JMS provider. You *must* set this connector property when you choose `JMS` as your delivery transport mechanism (DeliveryTransport).

The default is `CxCommon.Messaging.jms.IBMMQSeriesFactory`.

jms.MessageBrokerName

Specifies the broker name to use for the JMS provider. You *must* set this connector property when you choose `JMS` as your delivery transport mechanism (DeliveryTransport).

The default is `crossworlds.queue.manager`.

jms.NumConcurrentRequests

Specifies the maximum number of concurrent service call requests that can be sent to a connector at the same time. Once that maximum is reached, new service calls block and wait for another request to complete before proceeding.

The default value is 10.

jms.Password

Specifies the password for the JMS provider. A value for this property is optional.

There is no default.

jms.UserName

Specifies the user name for the JMS provider. A value for this property is optional.

There is no default.

ListenerConcurrency

This property supports multi-threading in MQ Listener when ICS is the integration broker. It enables batch writing of multiple events to the database, thus improving system performance. The default value is 1.

This property applies only to connectors using MQ transport. The `DeliveryTransport` property must be set to `MQ`.

Locale

Specifies the language code, country or territory, and, optionally, the associated character code set. The value of this property determines such cultural conventions as collation and sort order of data, date and time formats, and the symbols used in monetary specifications.

A locale name has the following format:

ll_TT.codeset

where:

ll	a two-character language code (usually in lower case)
TT	a two-letter country or territory code (usually in upper case)
codeset	the name of the associated character code set; this portion of the name is often optional.

By default, only a subset of supported locales appears in the drop list. To add other supported values to the drop list, you must manually modify the

\Data\Std\stdConnProps.xml file in the product directory. For more information, see the appendix on Connector Configurator.

The default value is en_US. If the connector has not been globalized, the only valid value for this property is en_US. To determine whether a specific connector has been globalized, see the connector version list on these websites:

http://www.ibm.com/software/websphere/wbiadapters/infocenter, or
http://www.ibm.com/websphere/integration/wicserver/infocenter

LogAtInterchangeEnd

Applicable only if RespositoryDirectory is <REMOTE>.

Specifies whether to log errors to the integration broker's log destination. Logging to the broker's log destination also turns on e-mail notification, which generates e-mail messages for the MESSAGE_RECIPIENT specified in the InterchangeSystem.cfg file when errors or fatal errors occur.

For example, when a connector loses its connection to its application, if LogAtInterChangeEnd is set to true, an e-mail message is sent to the specified message recipient. The default is false.

MaxEventCapacity

The maximum number of events in the controller buffer. This property is used by flow control and is applicable only if the value of the RepositoryDirectory property is <REMOTE>.

The value can be a positive integer between 1 and 2147483647. The default value is 2147483647.

MessageFileName

The name of the connector message file. The standard location for the message file is \connectors\messages. Specify the message filename in an absolute path if the message file is not located in the standard location.

If a connector message file does not exist, the connector uses InterchangeSystem.txt as the message file. This file is located in the product directory.

Note: To determine whether a specific connector has its own message file, see the individual adapter user guide.

MonitorQueue

The logical queue that the connector uses to monitor duplicate events. It is used only if the DeliveryTransport property value is JMS and DuplicateEventElimination is set to TRUE.

The default value is CONNECTORNAME/MONITORQUEUE

OADAutoRestartAgent

Valid only when the RepositoryDirectory is <REMOTE>.

Specifies whether the connector uses the automatic and remote restart feature. This feature uses the MQ-triggered Object Activation Daemon (OAD) to restart the connector after an abnormal shutdown, or to start a remote connector from System Monitor.

This property must be set to `true`to enable the automatic and remote restart feature. For information on how to configure the MQ-triggered OAD feature. see the *Installation Guide for Windows* or *for UNIX*.

The default value is `false`.

OADMaxNumRetry

Valid only when the `RepositoryDirectory` is <REMOTE>.

Specifies the maximum number of times that the MQ-triggered OAD automatically attempts to restart the connector after an abnormal shutdown. The `OADAutoRestartAgent` property must be set to `true` for this property to take effect.

The default value is `1000`.

OADRetryTimeInterval

Valid only when the `RepositoryDirectory` is <REMOTE>.

Specifies the number of minutes in the retry-time interval for the MQ-triggered OAD. If the connector agent does not restart within this retry-time interval, the connector controller asks the OAD to restart the connector agent again. The OAD repeats this retry process as many times as specified by the `OADMaxNumRetry` property. The `OADAutoRestartAgent` property must be set to `true` for this property to take effect.

The default is `10`.

PollEndTime

Time to stop polling the event queue. The format is `HH:MM`, where *HH* represents 0-23 hours, and *MM* represents 0-59 seconds.

You must provide a valid value for this property. The default value is `HH:MM`, but must be changed.

PollFrequency

The amount of time between polling actions. Set `PollFrequency` to one of the following values:

- The number of milliseconds between polling actions.
- The word `key`, which causes the connector to poll only when you type the letter `p` in the connector's Command Prompt window. Enter the word in lowercase.
- The word `no`, which causes the connector not to poll. Enter the word in lowercase.

The default is `10000`.

Important: Some connectors have restrictions on the use of this property. To determine whether a specific connector does, see the installing and configuring chapter of its adapter guide.

PollQuantity

Designates the number of items from the application that the connector should poll for. If the adapter has a connector-specific property for setting the poll quantity, the value set in the connector-specific property will override the standard property value.

PollStartTime

The time to start polling the event queue. The format is *HH:MM*, where *HH* represents 0-23 hours, and *MM* represents 0-59 seconds.

You must provide a valid value for this property. The default value is `HH:MM`, but must be changed.

RequestQueue

The queue that is used by the integration broker to send business objects to the connector.

The default value is `CONNECTOR/REQUESTQUEUE`.

RepositoryDirectory

The location of the repository from which the connector reads the XML schema documents that store the meta-data for business object definitions.

When the integration broker is ICS, this value must be set to <REMOTE> because the connector obtains this information from the InterChange Server repository.

When the integration broker is a WebSphere message broker or WAS, this value must be set to <*local directory*>.

ResponseQueue

Applicable only if `DeliveryTransport` is JMS and required only if `RepositoryDirectory` is <REMOTE>.

Designates the JMS response queue, which delivers a response message from the connector framework to the integration broker. When the integration broker is ICS, the server sends the request and waits for a response message in the JMS response queue.

RestartRetryCount

Specifies the number of times the connector attempts to restart itself. When used for a parallel connector, specifies the number of times the master connector application-specific component attempts to restart the slave connector application-specific component.

The default is `3`.

RestartRetryInterval

Specifies the interval in minutes at which the connector attempts to restart itself. When used for a parallel connector, specifies the interval at which the master connector application-specific component attempts to restart the slave connector application-specific component. Possible values ranges from 1 to 2147483647.

The default is 1.

RHF2MessageDomain

WebSphere MQ Integrator broker only.

This property allows you to configure the value of the field domain name in the JMS header. When data is sent to WMQI over JMS transport, the adapter framework writes JMS header information, with a domain name and a fixed value of `mrm`. A connfigurable domain name enables users to track how the WMQI broker processes the message data.

A sample header would look like this:

```
<mcd><Msd>mrm</Msd><Set>3</Set><Type>
Retek_POPhyDesc</Type><Fmt>CwXML</Fmt></mcd>
```

The default value is `mrm`, but it may also be set to `xml`. This property only appears when DeliveryTransport is set to `JMS`and WireFormat is set to `CwXML`.

SourceQueue

Applicable only if `DeliveryTransport` is JMS and ContainerManagedEvents is specified.

Designates the JMS source queue for the connector framework in support of guaranteed event delivery for JMS-enabled connectors that use a JMS event store. For further information, see "ContainerManagedEvents" on page 319.

The default value is `CONNECTOR/SOURCEQUEUE`.

SynchronousRequestQueue

Applicable only if `DeliveryTransport` is JMS.

Delivers request messages that require a synchronous response from the connector framework to the broker. This queue is necessary only if the connector uses synchronous execution. With synchronous execution, the connector framework sends a message to the `SynchronousRequestQueue` and waits for a response back from the broker on the `SynchronousResponseQueue`. The response message sent to the connector bears a correlation ID that matches the ID of the original message.

The default is `CONNECTORNAME/SYNCHRONOUSREQUESTQUEUE`

SynchronousResponseQueue

Applicable only if `DeliveryTransport` is JMS.

Delivers response messages sent in reply to a synchronous request from the broker to the connector framework. This queue is necessary only if the connector uses synchronous execution.

The default is `CONNECTORNAME/SYNCHRONOUSRESPONSEQUEUE`

SynchronousRequestTimeout

Applicable only if `DeliveryTransport` is JMS.

Specifies the time in minutes that the connector waits for a response to a synchronous request. If the response is not received within the specified time, then the connector moves the original synchronous request message into the fault queue along with an error message.

The default value is 0.

WireFormat

Message format on the transport.

- If the `RepositoryDirectory` is a local directory, the setting is `CwXML`.
- If the value of `RepositoryDirectory` is <REMOTE>, the setting is`CwBO`.

WsifSynchronousRequest Timeout

WAS integration broker only.

Specifies the time in minutes that the connector waits for a response to a synchronous request. If the response is not received within the specified, time then the connector moves the original synchronous request message into the fault queue along with an error message.

The default value is 0.

XMLNameSpaceFormat

WebSphere message brokers and WAS integration broker only.

A strong property that allows the user to specify short and long name spaces in the XML format of business object definitions.

The default value is `short`.

Appendix B. Connector Configurator

This appendix describes how to use Connector Configurator to set configuration property values for your adapter.

You use Connector Configurator to:

- Create a connector-specific property template for configuring your connector
- Create a configuration file
- Set properties in a configuration file

Note:

In this document, backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes and follow the conventions for each operating system.

The topics covered in this appendix are:

Overview of Connector Configurator

Connector Configurator allows you to configure the connector component of your adapter for use with these integration brokers:

- WebSphere InterChange Server (ICS)
- WebSphere MQ Integrator, WebSphere MQ Integrator Broker, and WebSphere Business Integration Message Broker, collectively referred to as the WebSphere Message Brokers (WMQI)
- WebSphere Application Server (WAS)

You use Connector Configurator to:

- Create a **connector-specific property template** for configuring your connector.
- Create a **connector configuration file**; you must create one configuration file for each connector you install.
- Set properties in a configuration file.
  You may need to modify the default values that are set for properties in the connector templates. You must also designate supported business object definitions and, with ICS, maps for use with collaborations as well as specify messaging, logging and tracing, and data handler parameters, as required.

The mode in which you run Connector Configurator, and the configuration file type you use, may differ according to which integration broker you are running. For example, if WMQI is your broker, you run Connector Configurator directly, and not from within System Manager (see "Running Configurator in stand-alone mode" on page 330).

Connector configuration properties include both standard configuration properties (the properties that all connectors have) and connector-specific properties (properties that are needed by the connector for a specific application or technology).

Because **standard properties** are used by all connectors, you do not need to define those properties from scratch; Connector Configurator incorporates them into your configuration file as soon as you create the file. However, you do need to set the value of each standard property in Connector Configurator.

The range of standard properties may not be the same for all brokers and all configurations. Some properties are available only if other properties are given a specific value. The Standard Properties window in Connector Configurator will show the properties available for your particular configuration.

For **connector-specific properties**, however, you need first to define the properties and then set their values. You do this by creating a connector-specific property template for your particular adapter. There may already be a template set up in your system, in which case, you simply use that. If not, follow the steps in "Creating a new template" on page 331 to set up a new one.

Note: Connector Configurator runs only in a Windows environment. If you are running the connector in a UNIX environment, use Connector Configurator in Windows to modify the configuration file and then copy the file to your UNIX environment.

Starting Connector Configurator

You can start and run Connector Configurator in either of two modes:

- Independently, in stand-alone mode
- From System Manager

Running Configurator in stand-alone mode

You can run Connector Configurator independently and work with connector configuration files, irrespective of your broker.

To do so when the broker is IBM WebSphere InterChange Server:

- From **Start>Programs**, click **IBM WebSphere InterChange Server>IBM WebSphere Business Integration Toolset>Development>Connector Configurator**.
- Select **File>New>Configuration File**.
- When you click the pull-down menu next to **System Connectivity Integration Broker**, you can select ICS connectivity.

To do so when you have WebSphere Business Integration Adapters and another broker installed:

- From **Start>Programs**, click **IBM WebSphere Business Integration Adapters>Tools>Connector Configurator**.
- Select **File>New>Connector Configuration**.
- When you click the pull-down menu next to **System Connectivity Integration Broker:**, you can select WMQI or WAS connectivity, depending on your broker.

You may choose to run Connector Configurator independently to generate the file, and then connect to System Manager to save it in a System Manager project (see "Completing a configuration file" on page 335.)

Running Configurator from System Manager

You can run Connector Configurator from System Manager.

To run Connector Configurator:

1. Open the System Manager.
2. In the System Manager window, expand the **Integration Component Libraries** icon and highlight **Connectors**.
3. From the System Manager menu bar, click **Tools>Connector Configurator**. The Connector Configurator window opens and displays a **New Connector** dialog box.
4. When you click the pull-down menu next to **System Connectivity Integration Broker**, you can select ICS, WebSphere Message Brokers or WAS, depending on your broker.

To edit an existing configuration file:

1. In the System Manager window, select any of the configuration files listed in the Connector folder and right-click on it. Connector Configurator opens and displays the configuration file with the integration broker type and file name at the top.
2. Click the Standard Properties tab to see which properties are included in this configuration file.

Creating a connector-specific property template

To create a configuration file for your connector, you need a connector-specific property template as well as the system-supplied standard properties.

You can create a brand-new template for the connector-specific properties of your connector, or you can use an existing file as the template.

- To create a new template, see "Creating a new template" on page 331.
- To use an existing file, simply modify an existing template and save it under the new name.

Creating a new template

This section describes how you create properties in the template, define general characteristics and values for those properties, and specify any dependencies between the properties. Then you save the template and use it as the base for creating a new connector configuration file.

To create a template:

1. Click **File>New>Connector-Specific Property Template**.
2. The **Connector-Specific Property Template** dialog box appears, with the following fields:
   - **Template**, and **Name**

     Enter a unique name that identifies the connector, or type of connector, for which this template will be used. You will see this name again when you open the dialog box for creating a new configuration file from a template.

- **Old Template**, and **Select the Existing Template to Modify**

  The names of all currently available templates are displayed in the Template Name display.
- To see the connector-specific property definitions in any template, select that template's name in the **Template Name** display. A list of the property definitions contained in that template will appear in the **Template Preview** display. You can use an existing template whose property definitions are similar to those required by your connector as a starting point for your template.

3. Select a template from the **Template Name** display, enter that template name in the **Find Name** field (or highlight your selection in **Template Name**), and click **Next**.

If you do not see any template that displays the connector-specific properties used by your connector, you will need to create one.

Specifying general characteristics

When you click **Next** to select a template, the **Properties - Connector-Specific Property Template** dialog box appears. The dialog box has tabs for General characteristics of the defined properties and for Value restrictions. The General display has the following fields:

- **General:**
  Property Type
  Updated Method
  Description
- **Flags**
  Standard flags
- **Custom Flag**
  Flag

After you have made selections for the general characteristics of the property, click the **Value** tab.

Specifying values

The **Value** tab enables you to set the maximum length, the maximum multiple values, a default value, or a value range for the property. It also allows editable values. To do so:

1. Click the **Value** tab. The display panel for Value replaces the display panel for General.
2. Select the name of the property in the **Edit properties** display.
3. In the fields for **Max Length** and **Max Multiple Values**, make any changes. The changes will not be accepted unless you also open the **Property Value** dialog box for the property, described in the next step.
4. Right-click the box in the top left-hand corner of the value table and click **Add**. A **Property Value** dialog box appears. Depending on the property type, the dialog box allows you to enter either a value, or both a value and range. Enter the appropriate value or range, and click **OK**.
5. The **Value** panel refreshes to display any changes you made in **Max Length** and **Max Multiple Values**. It displays a table with three columns:

   The **Value** column shows the value that you entered in the **Property Value** dialog box, and any previous values that you created.

   The **Default Value** column allows you to designate any of the values as the default.

The **Value Range** shows the range that you entered in the **Property Value** dialog box.

After a value has been created and appears in the grid, it can be edited from within the table display. To make a change in an existing value in the table, select an entire row by clicking on the row number. Then right-click in the **Value** field and click **Edit Value**.

Setting dependencies

When you have made your changes to the **General** and **Value** tabs, click **Next**. The **Dependences - Connector-Specific Property Template** dialog box appears.

A dependent property is a property that is included in the template and used in the configuration file *only if* the value of another property meets a specific condition. For example, `PollQuantity` appears in the template only if JMS is the transport mechanism and `DuplicateEventElimination` is set to `True`.
To designate a property as dependent and to set the condition upon which it depends, do this:

1. In the **Available Properties** display, select the property that will be made dependent.
2. In the **Select Property** field, use the drop-down menu to select the property that will hold the conditional value.
3. In the **Condition Operator** field, select one of the following:

   == (equal to)

   != (not equal to)

   > (greater than)

   < (less than)

   >= (greater than or equal to)

   <=(less than or equal to)
4. In the **Conditional Value** field, enter the value that is required in order for the dependent property to be included in the template.
5. With the dependent property highlighted in the **Available Properties** display, click an arrow to move it to the **Dependent Property** display.
6. Click **Finish**. Connector Configurator stores the information you have entered as an XML document, under `\data\app` in the`\bin` directory where you have installed Connector Configurator.

Creating a new configuration file

When you create a new configuration file, your first step is to select an integration broker. The broker you select determines the properties that will appear in the configuration file.

To select a broker:

- In the Connector Configurator home menu, click **File>New>Connector Configuration**. The **New Connector** dialog box appears.
- In the I**ntegration Broker** field, select ICS, WebSphere Message Brokers or WAS connectivity.
- Complete the remaining fields in the **New Connector** window, as described later in this chapter.

You can also do this:

- In the System Manager window, right-click on the **Connectors** folder and select **Create New Connector**. Connector Configurator opens and displays the **New Connector** dialog box.

Creating a configuration file from a connector-specific template

Once a connector-specific template has been created, you can use it to create a configuration file:

1. Click **File>New>Connector Configuration**.
2. The **New Connector** dialog box appears, with the following fields:
   - **Name**

     Enter the name of the connector. Names are case-sensitive. The name you enter must be unique, and must be consistent with the file name for a connector that is installed on the system.

     Important: Connector Configurator does not check the spelling of the name that you enter. You must ensure that the name is correct.
   - **System Connectivity**

     Click ICS or WebSphere Message Brokers or WAS.
   - **Select Connector-Specific Property Template**

     Type the name of the template that has been designed for your connector. The available templates are shown in the **Template Name** display. When you select a name in the Template Name display, the **Property Template Preview** display shows the connector-specific properties that have been defined in that template.

     Select the template you want to use and click **OK**.
3. A configuration screen appears for the connector that you are configuring. The title bar shows the integration broker and connector names. You can fill in all the field values to complete the definition now, or you can save the file and complete the fields later.
4. To save the file, click **File>Save>To File** or **File>Save>To Project**. To save to a project, System Manager must be running.
   If you save as a file, the **Save File Connector** dialog box appears. Choose `*.cfg` as the file type, verify in the File Name field that the name is spelled correctly and has the correct case, navigate to the directory where you want to locate the file, and click **Save**. The status display in the message panel of Connector Configurator indicates that the configuration file was successfully created.

   Important: The directory path and name that you establish here must match the connector configuration file path and name that you supply in the startup file for the connector.
5. To complete the connector definition, enter values in the fields for each of the tabs of the Connector Configurator window, as described later in this chapter.

Using an existing file

You may have an existing file available in one or more of the following formats:

- A connector definition file.
  This is a text file that lists properties and applicable default values for a specific connector. Some connectors include such a file in a `\repository` directory in their delivery package (the file typically has the extension `.txt`; for example, CN_XML.txt for the XML connector).

- An ICS repository file.
  Definitions used in a previous ICS implementation of the connector may be available to you in a repository file that was used in the configuration of that connector. Such a file typically has the extension `.in` or `.out`.
- A previous configuration file for the connector.
  Such a file typically has the extension *.cfg.

Although any of these file sources may contain most or all of the connector-specific properties for your connector, the connector configuration file will not be complete until you have opened the file and set properties, as described later in this chapter.

To use an existing file to configure a connector, you must open the file in Connector Configurator, revise the configuration, and then resave the file.

Follow these steps to open a *.txt, *.cfg, or *.in file from a directory:

1. In Connector Configurator, click **File>Open>From File**.
2. In the **Open File Connector** dialog box, select one of the following file types to see the available files:
   - Configuration (`*.cfg`)
   - ICS Repository (`*.in`, `*.out`)

     Choose this option if a repository file was used to configure the connector in an ICS environment. A repository file may include multiple connector definitions, all of which will appear when you open the file.
   - All files (*.*)

     Choose this option if a `*.txt` file was delivered in the adapter package for the connector, or if a definition file is available under another extension.
3. In the directory display, navigate to the appropriate connector definition file, select it, and click **Open**.

Follow these steps to open a connector configuration from a System Manager project:

1. Start System Manager. A configuration can be opened from or saved to System Manager only if System Manager has been started.
2. Start Connector Configurator.
3. Click **File>Open>From Project**.

Completing a configuration file

When you open a configuration file or a connector from a project, the Connector Configurator window displays the configuration screen, with the current attributes and values.

The title of the configuration screen displays the integration broker and connector name as specified in the file. Make sure you have the correct broker. If not, change the broker value before you configure the connector. To do so:

1. Under the **Standard Properties** tab, select the value field for the `BrokerType` property. In the drop-down menu, select the value `ICS`, `WMQI`, or `WAS`.
2. The Standard Properties tab will display the properties associated with the selected broker. You can save the file now or complete the remaining configuration fields, as described in "Specifying supported business object definitions" on page 338..

3. When you have finished your configuration, click **File>Save>To Project** or **File>Save>To File**.

   If you are saving to file, select `*.cfg` as the extension, select the correct location for the file and click **Save**.

   If multiple connector configurations are open, click **Save All to File** to save all of the configurations to file, or click **Save All to Project** to save all connector configurations to a System Manager project.

   Before it saves the file, Connector Configurator checks that values have been set for all required standard properties. If a required standard property is missing a value, Connector Configurator displays a message that the validation failed. You must supply a value for the property in order to save the configuration file.

Setting the configuration file properties

When you create and name a new connector configuration file, or when you open an existing connector configuration file, Connector Configurator displays a configuration screen with tabs for the categories of required configuration values.

Connector Configurator requires values for properties in these categories for connectors running on all brokers:

- Standard Properties
- Connector-specific Properties
- Supported Business Objects
- Trace/Log File values
- Data Handler (applicable for connectors that use JMS messaging with guaranteed event delivery)

Note: For connectors that use JMS messaging, an additional category may display, for configuration of data handlers that convert the data to business objects.

For connectors running on **ICS**, values for these properties are also required:

- Associated Maps
- Resources
- Messaging (where applicable)

Important: Connector Configurator accepts property values in either English or non-English character sets. However, the names of both standard and connector-specific properties, and the names of supported business objects, must use the English character set only.

Standard properties differ from connector-specific properties as follows:

- Standard properties of a connector are shared by both the application-specific component of a connector and its broker component. All connectors have the same set of standard properties. These properties are described in Appendix A of each adapter guide. You can change some but not all of these values.
- Application-specific properties apply only to the application-specific component of a connector, that is, the component that interacts directly with the application. Each connector has application-specific properties that are unique to its application. Some of these properties provide default values and some do not; you can modify some of the default values. The installation and configuration chapters of each adapter guide describe the application-specific properties and the recommended values.

The fields for **Standard Properties** and **Connector-Specific Properties** are color-coded to show which are configurable:

- A field with a grey background indicates a standard property. You can change the value but cannot change the name or remove the property.
- A field with a white background indicates an application-specific property. These properties vary according to the specific needs of the application or connector. You can change the value and delete these properties.
- Value fields are configurable.
- The **Update Method** field is informational and not configurable. This field specifies the action required to activate a property whose value has changed.

Setting standard connector properties

To change the value of a standard property:

1. Click in the field whose value you want to set.
2. Either enter a value, or select one from the drop-down menu if it appears.
3. After entering all the values for the standard properties, you can do one of the following:
   - To discard the changes, preserve the original values, and exit Connector Configurator, click **File>Exit** (or close the window), and click **No** when prompted to save changes.
   - To enter values for other categories in Connector Configurator, select the tab for the category. The values you enter for **Standard Properties** (or any other category) are retained when you move to the next category. When you close the window, you are prompted to either save or discard the values that you entered in all the categories as a whole.
   - To save the revised values, click **File>Exit** (or close the window) and click **Yes** when prompted to save changes. Alternatively, click **Save>To File** from either the File menu or the toolbar.

Setting application-specific configuration properties

For application-specific configuration properties, you can add or change property names, configure values, delete a property, and encrypt a property. The default property length is 255 characters.

1. Right-click in the top left portion of the grid. A pop-up menu bar will appear. Click **Add** to add a property. To add a child property, right-click on the parent row number and click **Add child**.
2. Enter a value for the property or child property.
3. To encrypt a property, select the **Encrypt** box.
4. Choose to save or discard changes, as described for "Setting standard connector properties."

The Update Method displayed for each property indicates whether a component or agent restart is necessary to activate changed values.

Important: Changing a preset application-specific connector property name may cause a connector to fail. Certain property names may be needed by the connector to connect to an application or to run properly.

Encryption for connector properties

Application-specific properties can be encrypted by selecting the **Encrypt** check box in the **Edit Property** window. To decrypt a value, click to clear the **Encrypt**

check box, enter the correct value in the **Verification** dialog box, and click **OK**. If the entered value is correct, the value is decrypted and displays.

The adapter user guide for each connector contains a list and description of each property and its default value.

If a property has multiple values, the **Encrypt** check box will appear for the first value of the property. When you select **Encrypt**, all values of the property will be encrypted. To decrypt multiple values of a property, click to clear the **Encrypt** check box for the first value of the property, and then enter the new value in the **Verification** dialog box. If the input value is a match, all multiple values will decrypt.

Update method

Refer to the descriptions of update methods found in the *Standard configuration properties for connectors* appendix, under "Setting and updating property values" on page 314.

Specifying supported business object definitions

Use the **Supported Business Objects** tab in Connector Configurator to specify the business objects that the connector will use. You must specify both generic business objects and application-specific business objects, and you must specify associations for the maps between the business objects.

Note: Some connectors require that certain business objects be specified as supported in order to perform event notification or additional configuration (using meta-objects) with their applications. For more information, see the *Connector Development Guide for C++* or the *Connector Development Guide for Java*.

If ICS is your broker

To specify that a business object definition is supported by the connector, or to change the support settings for an existing business object definition, click the **Supported Business Objects** tab and use the following fields.

Business object name: To designate that a business object definition is supported by the connector, with System Manager running:

1. Click an empty field in the **Business Object Name** list. A drop-down list displays, showing all the business object definitions that exist in the System Manager project.
2. Click on a business object to add it.
3. Set the **Agent Support** (described below) for the business object.
4. In the File menu of the Connector Configurator window, click **Save to Project**. The revised connector definition, including designated support for the added business object definition, is saved to the project in System Manager.

To delete a business object from the supported list:

1. To select a business object field, click the number to the left of the business object.
2. From the **Edit** menu of the Connector Configurator window, click **Delete Row**. The business object is removed from the list display.
3. From the **File** menu, click **Save to Project**.

Deleting a business object from the supported list changes the connector definition and makes the deleted business object unavailable for use in this implementation of this connector. It does not affect the connector code, nor does it remove the business object definition itself from System Manager.

Agent support: If a business object has Agent Support, the system will attempt to use that business object for delivering data to an application via the connector agent.

Typically, application-specific business objects for a connector are supported by that connector's agent, but generic business objects are not.

To indicate that the business object is supported by the connector agent, check the **Agent Support** box. The Connector Configurator window does not validate your Agent Support selections.

Maximum transaction level: The maximum transaction level for a connector is the highest transaction level that the connector supports.

For most connectors, `Best Effort` is the only possible choice.

You must restart the server for changes in transaction level to take effect.

If a WebSphere Message Broker is your broker

If you are working in stand-alone mode (not connected to System Manager), you must enter the business name manually.

If you have System Manager running, you can select the empty box under the **Business Object Name** column in the **Supported Business Objects** tab. A combo box appears with a list of the business object available from the Integration Component Library project to which the connector belongs. Select the business object you want from the list.

The **Message Set ID** is an optional field for WebSphere Business Integration Message Broker 5.0, and need not be unique if supplied. However, for WebSphere MQ Integrator and Integrator Broker 2.1, you must supply a unique **ID**.

If WAS is your broker

When WebSphere Application Server is selected as your broker type, Connector Configurator does not require message set IDs. The **Supported Business Objects** tab shows a **Business Object Name** column only for supported business objects.

If you are working in stand-alone mode (not connected to System Manager), you must enter the business object name manually.

If you have System Manager running, you can select the empty box under the Business Object Name column in the Supported Business Objects tab. A combo box appears with a list of the business objects available from the Integration Component Library project to which the connector belongs. Select the business object you want from this list.

Associated maps (ICS only)

Each connector supports a list of business object definitions and their associated maps that are currently active in WebSphere InterChange Server. This list appears when you select the **Associated Maps** tab.

The list of business objects contains the application-specific business object which the agent supports and the corresponding generic object that the controller sends to the subscribing collaboration. The association of a map determines which map will be used to transform the application-specific business object to the generic business object or the generic business object to the application-specific business object.

If you are using maps that are uniquely defined for specific source and destination business objects, the maps will already be associated with their appropriate business objects when you open the display, and you will not need (or be able) to change them.

If more than one map is available for use by a supported business object, you will need to explicitly bind the business object with the map that it should use.

The **Associated Maps** tab displays the following fields:

- **Business Object Name**

  These are the business objects supported by this connector, as designated in the **Supported Business Objects** tab. If you designate additional business objects under the Supported Business Objects tab, they will be reflected in this list after you save the changes by choosing **Save to Project** from the **File** menu of the Connector Configurator window.
- **Associated Maps**

  The display shows all the maps that have been installed to the system for use with the supported business objects of the connector. The source business object for each map is shown to the left of the map name, in the **Business Object Name** display.
- **Explicit**

  In some cases, you may need to explicitly bind an associated map.

  Explicit binding is required only when more than one map exists for a particular supported business object. When ICS boots, it tries to automatically bind a map to each supported business object for each connector. If more than one map takes as its input the same business object, the server attempts to locate and bind one map that is the superset of the others.

  If there is no map that is the superset of the others, the server will not be able to bind the business object to a single map, and you will need to set the binding explicitly.

  To explicitly bind a map:

  1. In the **Explicit** column, place a check in the check box for the map you want to bind.
  2. Select the map that you intend to associate with the business object.
  3. In the **File** menu of the Connector Configurator window, click **Save to Project**.
  4. Deploy the project to ICS.
  5. Reboot the server for the changes to take effect.

Resources (ICS)

The **Resource** tab allows you to set a value that determines whether and to what extent the connector agent will handle multiple processes concurrently, using connector agent parallelism.
Not all connectors support this feature. If you are running a connector agent that

was designed in Java to be multi-threaded, you are advised not to use this feature, since it is usually more efficient to use multiple threads than multiple processes.

Messaging (ICS)

The messaging properties are available only if you have set MQ as the value of the `DeliveryTransport` standard property and ICS as the broker type. These properties affect how your connector will use queues.

Setting trace/log file values

When you open a connector configuration file or a connector definition file, Connector Configurator uses the logging and tracing values of that file as default values. You can change those values in Connector Configurator.

To change the logging and tracing values:

1. Click the **Trace/Log Files** tab.
2. For either logging or tracing, you can choose to write messages to one or both of the following:
   - To console (STDOUT):
     Writes logging or tracing messages to the STDOUT display.

     Note: You can only use the STDOUT option from the **Trace/Log Files** tab for connectors running on the Windows platform.
   - To File:
     Writes logging or tracing messages to a file that you specify. To specify the file, click the directory button (ellipsis), navigate to the preferred location, provide a file name, and click **Save**. Logging or tracing message are written to the file and location that you specify.

     Note: Both logging and tracing files are simple text files. You can use the file extension that you prefer when you set their file names. For tracing files, however, it is advisable to use the extension `.trace` rather than `.trc`, to avoid confusion with other files that might reside on the system. For logging files, `.log` and `.txt` are typical file extensions.

Data handlers

The data handlers section is available for configuration only if you have designated a value of JMS for DeliveryTransport and a value of JMS for ContainerManagedEvents. Not all adapters make use of data handlers.

See the descriptions under `ContainerManagedEvents` in Appendix A, Standard Properties, for values to use for these properties. For additional details, see the *Connector Development Guide for C++* or the *Connector Development Guide for Java.*

Saving your configuration file

When you have finished configuring your connector, save the connector configuration file. Connector Configurator saves the file in the broker mode that you selected during configuration. The title bar of Connector Configurator always displays the broker mode (ICS, WMQI or WAS) that it is currently using.

The file is saved as an XML document. You can save the XML document in three ways:

- From System Manager, as a file with a `*.con` extension in an Integration Component Library, or
- In a directory that you specify.
- In stand-alone mode, as a file with a `*.cfg` extension in a directory folder.

For details about using projects in System Manager, and for further information about deployment, see the following implementation guides:

- For ICS: *Implementation Guide for WebSphere InterChange Server*
- For WebSphere Message Brokers: *Implementing Adapters with WebSphere Message Brokers*
- For WAS: *Implementing Adapters with WebSphere Application Server*

Changing a configuration file

You can change the integration broker setting for an existing configuration file. This enables you to use the file as a template for creating a new configuration file, which can be used with a different broker.

Note: You will need to change other configuration properties as well as the broker mode property if you switch integration brokers.

To change your broker selection within an existing configuration file (optional):

- Open the existing configuration file in Connector Configurator.
- Select the **Standard Properties** tab.
- In the **BrokerType** field of the Standard Properties tab, select the value that is appropriate for your broker.
  When you change the current value, the available tabs and field selections on the properties screen will immediately change, to show only those tabs and fields that pertain to the new broker you have selected.

Completing the configuration

After you have created a configuration file for a connector and modified it, make sure that the connector can locate the configuration file when the connector starts up.

To do so, open the startup file used for the connector, and verify that the location and file name used for the connector configuration file match exactly the name you have given the file and the directory or path where you have placed it.

Using Connector Configurator in a globalized environment

Connector Configurator is globalized and can handle character conversion between the configuration file and the integration broker. Connector Configurator uses native encoding. When it writes to the configuration file, it uses UTF-8 encoding.

Connector Configurator supports non-English characters in:

- All value fields
- Log file and trace file path (specified in the **Trace/Log files** tab)

The drop list for the `CharacterEncoding` and `Locale` standard configuration properties displays only a subset of supported values. To add other values to the drop list, you must manually modify the `\Data\Std\stdConnProps.xml` file in the product directory.

For example, to add the locale en_GB to the list of values for the Locale property, open the stdConnProps.xml file and add the line in boldface type below:

```
<Property name="Locale"
isRequired="true"
updateMethod="component restart">
                <ValidType>String</ValidType>
            <ValidValues>
                                <Value>ja_JP</Value>
                                <Value>ko_KR</Value>
                                <Value>zh_CN</Value>
                                <Value>zh_TW</Value>
                                <Value>fr_FR</Value>
                                <Value>de_DE</Value>
                                <Value>it_IT</Value>
                                <Value>es_ES</Value>
                                <Value>pt_BR</Value>
                                <Value>en_US</Value>
                                <Value>en_GB</Value>
                    <DefaultValue>en_US</DefaultValue>
            </ValidValues>
    </Property>
```

Appendix C. Connector Script Generator

The Connector Script Generator utility creates or modifies the connector script for connectors running on the UNIX platform. Use this tool to do either of the following:

- To generate a new connector startup script for a connector you have added without using the WebSphere Business Integration Adapters installer.
- To modify an existing startup script for a connector to include the correct configuration file path.

To run the Connector Script Generator, do the following:

1. Navigate to the *ProductDir*/bin directory.
2. Enter the command `./ConnConfig.sh`.

   The Connector Script Generator screen appears as shown in Figure 80.

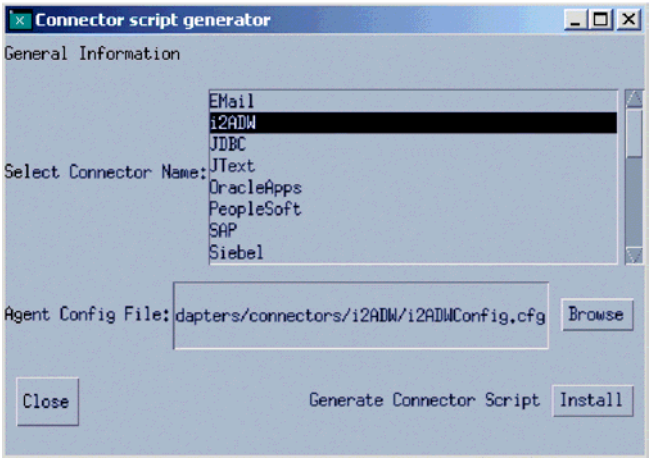

Figure 80. Connector Script Generator.

3. From the Select Connector Name list, select the connector for which the startup script is to be generated.
4. For Agent Config File, specify the connector's configuration file by entering its full-path name or by clicking **Browse** to select a file.
5. To generate or update the connector script, click **Install**.

   The connector_manager_*ConnectorName* file (where *ConnectorName* is the name of the connector you are configuring) is created in the *ProductDir*/bin directory.
6. Click **Close**.

Appendix D. Connector feature checklist

This appendix describes the connector feature checklist.

Guidelines for using the connector feature checklist

The connector feature checklist briefly describes each of the standard features for connectors. The feature list establishes a baseline for the behavior of a connector. Therefore, as you design a new connector, you can use the list as a quick reference to standard connector features.

During the implementation phase for your connector, you can use the feature list to create a specification describing the functionality of your connector. To use the list:

- Check *Full* for each feature that your connector supports.
- Check *Partial* for each feature that your connector partially supports and include notes describing the implementation.
- Check *No* for each feature that the connector does not support.
- Check *N/A* for each feature that is not relevant for the connector. For example, if your connector does not implement event notification, check N/A for all event notification features.

If a feature is not supported according to the standard behavior, check *Partial* and provide additional information.

Standard behavior for request processing

Table 117 lists the standard features for connector handling of business object requests. The table includes a brief description of each feature and a page number of the section in the book containing more information on the feature.

Table 117. Standard features for request processing

Category and name	Description	Supported?
Business Object and Attribute Naming		
Business object names	Business object names should have no semantic value to the connector. Two business objects with the same structure, data, and application-specific information but with different names should process identically in the connector.	__ Full __ Partial __ No __ N/A
Attribute names	Attribute names in a business object should have no semantic value to the connector. Values such as application table name or column name should be stored in the application-specific information field of the attribute and not in the attribute name.	__ Full __ Partial __ No __ N/A
Create		
Create Verb	The connector creates the object in the destination application. The application object includes all values in the business object, including child objects. See “Handling the Create verb” on page 80.	__ Full __ Partial __ No __ N/A
Delete		
Delete Verb	The connector supports the Delete verb, and when processing this verb, it does a true physical delete, not a logical delete. See “Handling the Delete verb” on page 98.	__ Full __ Partial __ No __ N/A

Table 117. Standard features for request processing (continued)

Category and name	Description	Supported?
Logical delete	The connector supports logical deletes operations via the Update verb only. The Delete verb is used only for physical deletes. See "Implications of business objects representing logical Delete events" on page 95.	__ Full __ Partial __ No __ N/A
Exist		
Exist Verb	The connector checks for the existence of an entity in the application database. It returns `SUCCEED` if the object passed in exists in the application database, and `FAIL` if the object does not exist in the application database. See "Handling the Exists verb" on page 99.	__ Full __ Partial __ No __ N/A
Retrieve		
Retrieve Verb	The entire hierarchical image (including all child business objects) is retrieved from application when the Retrieve verb is processed. The retrieve is based only on the key values of the business object. See "Handling the Retrieve verb" on page 83.	__ Full __ Partial __ No __ N/A
Ignore missing child object	If `IgnoreMissingChildObject` is set to true in the business object level application-specific information, the connector returns `SUCCEED` even if not all the children specified in the business object are found in the application. See "Retrieving child objects" on page 86.	__ Full __ Partial __ No __ N/A
RetrieveByContent		
RetrieveBy Content Verb	The entire hierarchical image (including all child objects), based solely on a subset of non-key values, is retrieved. See "Handling the RetrieveByContent verb" on page 89.	__ Full __ Partial __ No __ N/A
Multiple results	If more than one object is retrieved from the application, RetrieveByContent should return the first object and use the return code `MULTIPLE_HITS`. See "Handling the RetrieveByContent verb" on page 89.	__ Full __ Partial __ No __ N/A
Ignore missing child object	If `IgnoreMissingChildObject` is set to true in the business object level application-specific information, the connector returns `SUCCEED` even if not all the children specified in the business object are found in the application.	__ Full __ Partial __ No __ N/A
Update		
After-image support	The connector performs all the steps necessary to make the object in the destination application exactly match the business object received in the `doVerbFor()`call. See "Handling the Update verb" on page 91.	__ Full __ Partial __ No __ N/A
Delta support	Connector processes exactly the objects and verbs that are received in the source business object. The destination application object is updated only by processing the contents of source business object, not by making the application representation match the source business object. [Not currently an IBM standard.]	__ Full __ Partial __ No __ N/A
KeepRelations	When KeepRelations is specified, child relations are not destroyed in the target application. Otherwise, all the child relations are destroyed first, then the child objects sent in from InterChange Server are created and the relations restored. "Destroyed" means a logical or physical delete of the relation to the child, or, in some cases, deletion of the child itself, depending on the functionality of the connector and application. KeepRelations is set as application-specific information on the child array in the parent object (not as text on the child itself). The syntax should be `keeprelations=true`.	__ Full __ Partial __ No __ N/A
Verb Support		

Table 117. Standard features for request processing (continued)

Category and name	Description	Supported?
Subverb support	The connector supports processing of verbs on child objects independent of the verb on the parent object. When a verb is set in a child business object, the connector performs the operation that the child verb indicates, regardless of the verb on the top-level business object. If a verb in a child business object request is not set, the connector can either leave the child verb as `NULL`, set the child verb to the verb in the top-level business object, or set the value of the verb to the operation that the connector needs to perform. See "Verb stability" on page 78.	__ Full __ Partial __ No __ N/A
Verb Stability	Verbs in a business object should remain stable throughout the request and response cycle. When a connector receives an business object request, the hierarchical object returned to InterChange Server should have the same verb(s) as the original request, with the exception of verbs that are set on child business objects that were null in the original request	__ Full __ Partial __ No __ N/A

Standard behavior for the event notification

Table 118 lists standard features for event retrieval and notification.

Table 118. Standard features for event notification

Category and name	Description	Supported?
Connector Properties		
Event distribution	The event retrieval mechanism includes a filter that processes only events that are associated with the connector making the poll call. This feature requires adding a ConnectorId field to the event table so that multiple connectors can use the same event table. Each connector also requires a ConnectorId connector property. This property sets the identifier for a particular instance of a connector and allows the connector to pick up only the events assigned to it. See "Event distribution" on page 126.	__ Full __ Partial __ No __ N/A
PollQuantity	The connector uses the `PollQuantity` connector property to specify the maximum number of events the connector will process for each poll call. If possible, the connector should limit the number of rows retrieved in the poll call to PollQuantity. (For example, in SQL Server, use the `set rowcount` option.) See "Retrieving event records" on page 180.	__ Full __ Partial __ No __ N/A
Event Table		
Event status values	Where applicable, the values are used for event status are described in Table 118.	__ Full __ Partial __ No __ N/A
Object key	The object key column must use name-value pairs to set data in a new business object. For example, if ContractId is the name of an attribute in the business object, the object key is: ContractId=45381. The connector should support multiple name-value pairs separated by a delimiter. The delimiter is configurable (PollAttributeDelimiter) and should default to a colon (:). See "Object key" on page 111.	__ Full __ Partial __ No __ N/A
Object name	The object name field should be set to the exact business object name. See "Standard contents of an event record" on page 110.	__ Full __ Partial __ No __ N/A

Table 118. Standard features for event notification (continued)

Category and name	Description	Supported?
Priority	Priority is 0-n, with 0 being the highest priority. The connector polls and processes events in order of priority. Note that no decrementing is done, which could, in rare circumstances, lead to low priority events being shut out (not processed). See "Processing events by event priority" on page 125.	__ Full __ Partial __ No __ N/A
Miscellaneous Features		
Archiving	An event is archived once the connector has processed it, whether or not the event was successfully delivered to InterChange Server. The event status is kept in the archive table and is one of the following: • Success. The event was detected, and an object was created and sent to InterChange Server. • Unsubscribed. The event was detected, but the connector did not have a subscription for that event/verb combination, so the event was not sent to InterChange Server. • Error. The event was detected, but the connector encountered an error in trying to process the event, either in the process of building a business object or in posting the object to InterChange Server.	__ Full __ Partial __ No __ N/A
CDK method `gotApplEvent()`	The connector should call `gotApplEvent()`only from within `pollForEvents()`.	__ Full __ Partial __ No __ N/A
Delta event notification	An event can be created that represents only the changes to a hierarchical business object, such as the addition or deletion of order lines, without creating an update event for the entire business object. [Not currently an IBM Standard]	__ Full __ Partial __ No __ N/A
Future event processing	The mechanism for specifying a future date or time at which an event should be processed. The connector does not process the event until that date or time. [Not currently an IBM Standard]	__ Full __ Partial __ No __ N/A
In-Progress event recovery	When restarted, a connector checks the event table for events that have a status of `IN_PROGRESS`. If any exist, the connector does one of the following: • `PropValue = FailOnStartup`: Logs a fatal error and sends an email notification. • `PropValue = Reprocess`: Submits the events to InterChange Server. • `PropValue = LogError`: Logs an error but does not shut down. • `PropValue = Ignore`: Ignores these entries in the event table. This behavior is configurable via the `InDoubtEvents` connector property. Use the Notes field to describe exactly how the connector handles this feature.	__ Full __ Partial __ No __ N/A
Physical delete event	The connector creates an empty business object with the Delete verb, with key values populated and the rest of the attributes populated with CxIgnore, and sends the object to InterChange Server. See "Processing Delete events" on page 126.	__ Full __ Partial __ No __ N/A
RetrieveAll	The connector retrieves the entire hierarchical business object during subscription delivery. See "Retrieving application data" on page 184.	__ Full __ Partial __ No __ N/A

Table 118. Standard features for event notification (continued)

Category and name	Description	Supported?
Smart filtering	Duplicate events are not saved in the event store. Before storing a new event as a record in the event store, the event detection mechanism queries the event store for existing events that match the new event. The event detection mechanism does not generate a record for a new event in these cases: • The business object name, verb, key, status, and ConnectorId (if applicable) in a new event match those of another unprocessed event in the event store. • The business object name, key, and status for a new event match an unprocessed event in the event store. In addition, the verb for the new event is Update, and the verb for the unprocessed event is Create. • The business object name, key, and status for a new event match an unprocessed event in the event store. In addition, the verb in the unprocessed event in the event store is Create, and the verb in the new event is Delete. In this case, remove the Create record from the event store.	__ Full __ Partial __ No __ N/A
Verb stability	The connector should send a business object with the same verb that is in the event table. See "Getting the business object name, verb, and key" on page 182.	__ Full __ Partial __ No __ N/A

General standards

Table 119 lists general standards for connector behavior.

Table 119. General standards

Category and Name	Description	Supported?
Business Object		
Foreign key	There is no standard defined. If you use this property, check *Full* and describe how you use it. If you do not use this property, check *No*.	__ Full __ Partial __ No __ N/A
Foreign key attribute property	If this attribute property is set to true, the connector verifies that the value is a valid key. If the key is invalid, the connector returns `FAIL`. The connector assumes a foreign key is present in the application, and the connector should never try to create an object marked as a foreign key.	__ Full __ Partial __ No __ N/A
Key	There is no standard defined. If you use this property, check *Full* and describe how you use it. If you do not use this property, check *No*.	__ Full __ Partial __ No __ N/A
Max Length	There is no standard defined. If you use this property, check *Full* and describe how you use it. If you do not use this property, check *No*.	__ Full __ Partial __ No __ N/A
Required	There is no standard defined. If you use this property, check *Full* and describe how you use it. If you do not use this property, check *No*.	__ Full __ Partial __ No __ N/A
Metadata driven design	The connector can support new business objects without recompiling because business object processing is based on metadata in the business object definition. See "Assessing support for metadata-driven design" on page 47.	__ Full __ Partial __ No __ N/A

Table 119. General standards (continued)

Category and Name	Description	Supported?
Loss of Connection to Application		
Connection lost on request processing	The connector detects the connection error when processing a business object request and shuts down. The connector logs a fatal error and sends a return code of `APPRESPONSETIMEOUT` so that email notification can be triggered. See "Handling loss of connection to an application" on page 72.	__ Full __ Partial __ No __ N/A
Connection lost on poll	The connector detects the connection error at the time of a poll call and shuts down. The connector logs a fatal error and sends a return code of `APPRESPONSETIMEOUT` so that email notification can be triggered. See "Handling loss of connection to an application" on page 72.	__ Full __ Partial __ No __ N/A
Connection lost while idle	Connector shuts down as soon as the connection to the application is lost. The connector logs a fatal error and sends a return code of `APPRESPONSETIMEOUT` so that email notification can be triggered.	__ Full __ Partial __ No __ N/A
Connector Properties		
ApplicationPassword	The connector should use this property value as the password to log in to the application.	__ Full __ Partial __ No __ N/A
ApplicationUser Name	The connector should use this property value as the user name to log in to the application.	__ Full __ Partial __ No __ N/A
UseDefaults connector property	If this connector property is set to true, when the connector processes a business object request with a Create verb, it calls the JCDK or CDK method `initAndValidateAttributes()`.	__ Full __ Partial __ No __ N/A
Message Tracing		
General messaging	Messages that identify the business object handlers used for each object. Messages that log each time a business object is posted to Interchange Server, either from `gotApplEvent()` or `consumeSync()`. Messages that indicate each time a business object request is received. Guidelines for the trace messages at each trace level 0-5 follow. Note that the connector should deliver all the trace messages applicable at the level of tracing set and lower. See "Trace messages" on page 137.	__ Full __ Partial __ No __ N/A
Trace Level 0	0 - Message that identifies the connector version. No other tracing is done at this level.	__ Full __ Partial __ No __ N/A
Trace Level 1	1 - Status messages and identifying (key) information for each business object processed. A message is sent each time the `pollForEvents()` method is executed.	__ Full __ Partial __ No __ N/A
Trace Level 2	2 - Messages that identify the business object handlers used for each object the connector processes. Messages that log each time a business object is posted to InterChange Server, either from `gotApplEvent()`or `consumeSync()`. Messages that indicate each time a business object request is received.	__ Full __ Partial __ No __ N/A

Table 119. General standards (continued)

Category and Name	Description	Supported?
Trace Level 3	3 - Messages that identify the foreign keys being processed (if applicable). These messages appear when the connector has encountered a foreign key in a business object or when the connector sets a foreign key in a business object. Messages that relate to business object processing. Examples of this include finding a match between business objects, or finding a business object in an array of child business objects.	__ Full __ Partial __ No __ N/A
Trace Level 4	4 - Messages that identify application-specific information. Examples of this text include the values returned by the functions that process the application-specific information fields in business objects. Messages that identify entry or exit functions. These messages help trace the process flow of the connector. Messages that trace any thread-specific processing. For example, if the connector spawns multiple threads, a message should log the creation of each new thread.	__ Full __ Partial __ No __ N/A
Trace Level 5	5 - Messages that indicate connector initialization. The messages include the value of each configuration property that has been retrieved from InterChange Server. Messages that detail the status of each thread the connector spawns while it is running.The connector log file contains all statements executed in the application and the value of any variables that are substituted (where applicable). Messages for business object dumps. The connector outputs a text representation of a business object before it begins processing (showing the object the connector receives from the integration broker) and after it has processed the object (showing the object the connector returns to the integration broker).	__ Full __ Partial __ No __ N/A
Message tracing	Do not use the CDK method `generateMsg()` for tracing; instead, hard-code the message strings for trace messages.	__ Full __ Partial __ No __ N/A
Miscellaneous Features		
Java package names	All Java-based connectors should follow these package naming standards: com.CompanyName.connectors.ConnectorAgentPrefix Example: `com.crossworlds.connectors.XML`	__ Full __ Partial __ No __ N/A
Logging messages	The connector logs errors and other information that the user needs regardless of the trace level set for the system. See 135.	__ Full __ Partial __ No __ N/A
CDK method `logMsg()`	Always use the CDK method `generateMsg()` before calling `logMsg()`.	__ Full __ Partial __ No __ N/A
NT service compliance	To be NT service-compliant, do not use any method or function that points to `STDOUT`, for example, the `printf()` method in C++.	__ Full __ Partial __ No __ N/A
Transaction support	An entire business object request must be wrapped in a single transaction. All Create, Update, and Delete transactions for a top-level business object and all of its children should be wrapped in a single transaction. If any failure is detected during the life of the transaction, the whole transaction should be rolled back.	__ Full __ Partial __ No __ N/A
Special IBM CrossWorlds Values		
CxBlank processing	On a Create operation, the connector inserts an appropriate blank value for attributes with the value CxBlank. The blank value may be configurable or specific to the application. See “Handling the Blank and Ignore values” on page 173.	__ Full __ Partial __ No __ N/A

Table 119. General standards (continued)

Category and Name	Description	Supported?
CxIgnore processing	The connector does not set a value in the application for attributes that are passed in with the value CxIgnore when processing Create or Update verbs. See "Handling the Blank and Ignore values" on page 173.	__ Full __ Partial __ No __ N/A

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Burlingame Laboratory Director
IBM Burlingame Laboratory
577 Airport Blvd., Suite 800

Burlingame, CA 94010
U.S.A

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not necessarily tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Programming interface information

Programming interface information, if provided, is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States or other countries, or both:

IBM
the IBM logo
AIX
CrossWorlds
DB2
DB2 Universal Database
Domino
Lotus
Lotus Notes
MQIntegrator
MQSeries
Tivoli
WebSphere

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

MMX, Pentium, and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others. WebSphere Business Integration Adapter Framework V2.4.0

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X