BusinessWare® Connector Programming Guide

BusinessWare Connector SDK Version 1.0

Last updated November 29, 2000

VITRIA TECHNOLOGY, INC.

Vitria Technology Inc
945 Stewart Drive
Sunnyvale, CA 94086-3912
Phone: (408) 212-2700
Fax: (408) 212-2720

TABLE OF CONTENTS

PREFACE

Welcome to the *BusinessWare Connector Programming Guide*. This guide describes how to design and implement BusinessWare connectors.

This preface contains the following information:

Audience

This book is intended for Java and C++ programmers who will be writing software components, called connector flows, that enable external systems to be integrated with BusinessWare solutions. Before reading this book or attempting to create a custom connector or custom transformer, be sure to explore the connector flows, transformers, templates, and other facilities that Vitria provides with the core BusinessWare solution to determine if a connector flow already exists that may solve the connectivity problem at hand. See the *BusinessWare Connection Modeling Guide* for information about using the BusinessWare Console to create new connection models from connector flows and transformers.

Related Reading

For related information on topics in this manual, you may find it useful to refer to the following manuals:

- *BusinessWare Programming Guide*. Provides information about the BusinessWare application programming interface and discusses the tasks required to create Vitria BusinessWare solutions; includes Java and C++ example code.

- *BusinessWare Programming Reference*. An online, hyperlinked reference to public BusinessWare Java, C++, and IDL APIs, as well as various command-line programming tools.
- *Simple Connector Sample*. A tutorial that shows you how to write flow source and a flow target connector. It explains coding the different pieces of a connector, putting them together, and installing them in the BusinessWare Console.
- *EBank Connector Sample*. An advanced tutorial that shows you how to make a connector transactional and robust. This sample includes a client/server banking application; writing a transaction resource; and exception handling.

BusinessWare Documentation Set

The BusinessWare documentation set consists of the following manuals:

Table 1 BusinessWare documentation

Manual	Description
BusinessWare Foundations	Explains the concepts underlying the product and discusses the tools and features used across the product: BusinessWare Console, XML, access and revision control, and channels. If you are new to BusinessWare, you should read this manual first.
BusinessWare System Administration Guide	Discusses administering the BusinessWare and Communicator servers, including server configuration, start-up and shutdown, managing system files, security, federation, and so on.
BusinessWare Process Modeling Guide	Describes how to model and automate business processes using BusinessWare Automator.
BusinessWare Process Management Guide	Describes how to configure audit logging and persistence to a database. This guide provides detailed information on audit database tables, including schema and audit event source.
BusinessWare Task List: Supervisor's Guide	Describes how process and activity supervisors manage performer tasks that have exceeded their completion deadlines. Topics include reassigning, releasing, and starting tasks, and terminating processes.
BusinessWare Task List: Performer's Guide	Describes how to use the BusinessWare Task List. Topics include logging on, viewing your Task List, starting tasks, reassigning tasks, releasing tasks, completing tasks, and logging off.

Table 1 BusinessWare documentation (Continued)

Manual	Description
BusinessWare Connection Modeling Guide	Describes how to build connection models using BusinessWare Connection Modeler.
BusinessWare Metadata Guide	Describes the role metadata plays in BusinessWare. Includes topics such as how to create metadata, manipulate events using metadata, and translate XML into metadata.
BusinessWare Programming Guide	Introduces the BusinessWare application programming interface, discusses the tasks required to create Vitria BusinessWare solutions, and provides examples of C++ and Java code.
BusinessWare Programming Reference	Documents BusinessWare's Java and C++ public APIs, as well as various command-line programming tools.
Migration: BusinessWare 2.2.1 to BusinessWare 3.1	Shows how to migrate a BusinessWare 2.2.1 solution to BusinessWare 3.1.
Migration: BusinessWare 2.3 to BusinessWare 3.1	Shows how to migrate a BusinessWare 2.3 solution to BusinessWare 3.1.
Migration: BusinessWare 3.0 to BusinessWare 3.1	Shows how to migrate a BusinessWare 3.0 solution to BusinessWare 3.1

How This Guide Is Organized

This book contains 11 chapters and an appendix, as described in Table 2.

Table 2 Contents of This Guide

Section	Contents
Chapter 1, "Introduction to Connectors"	Provides an introduction to connectors and connection models, and a glossary of terms.
Chapter 2, "Architecture Overview"	Provides a description of the connection model infrastructure and a behind the scenes view of connection models.
Chapter 3, "Design and Development Guidelines"	Provides connector design considerations.
Chapter 4, "Creating Java Components for a Connector Flow"	Describes how to create the user-configurable pieces of a connector flow and how to use the Connector Generator Tool to automatically generate many of the pieces.

Table 2 Contents of This Guide (Continued)

Section	Contents
Chapter 5, “Writing Java Connectors”	How to write the Java class for a source or target connector flow. a detailed explanation of sending and receiving events, saving the state of connector and transforming events.
Chapter 6, “Writing C++ Connectors”	How to write the C++ class for a source or target connector flow and use the C++ wrapper (FlowBridge).
Chapter 7, “Defining Event Interfaces”	Describes event specification in IDL and ODL and registering metadata.
Chapter 8, “Writing a Transaction Resource”	Shows how to write a transaction resource and provides various strategies for creating robust one-phase commit transaction resources.
Chapter 9, “Handling Errors”	Explains error handling, error events, error codes, logging and log levels.
Chapter 10, “Miscellaneous Development Issues”	Discusses multi-instancing; nested router; and customized transformer methods.
Chapter 11, “Installing Connectors and Connection Models”	How to install connectors and connection models using initialization files, templates, and command-line tools.
Appendix A, “Reference”	Listing of the connector API classes and methods.
“Glossary”	Definitions of terms used throughout this guide.

Conventions

This manual uses the following conventions:

- `Monospace`

  Specifies filenames, object names, and programming code.

  Example: User names are located in the `/admin/users` folder.

- *Italics*

  Introduces new terminology, highlights book titles, and provides emphasis.

  Example: An *operation* is a defined action that is meant to be applied to a given class of software objects.

- **Bold**

  Highlights items.

  Example: From the list, select the **properties** item.

Contacting Vitria

If you have any questions regarding Vitria or any of the Vitria products, please contact Vitria using the resources listed in this section.

Headquarters

Vitria Technology, Inc.
945 Stewart Drive
Sunnyvale, CA 94086
1-408-212-2700

http://www.vitria.com

Technical Support

Vitria Technical Support provides comprehensive support for all Vitria products through our Technical Support web site:

http://help.vitria.com (login and password required)

If you need a login for help.vitria.com, please send your request to:

contractadmin@vitria.com

Each customer who has purchased a support contract has two or more designated individuals who are authorized to contact Vitria Technical Support. If you have questions about using the BusinessWare products, please have a designated person at your site open a case with Vitria Technical Support.

Documentation

If you have any comments on the documentation, please contact the Documentation group directly:

documentation@vitria.com

We'd like to hear from you!

1 INTRODUCTION TO CONNECTORS

The *Connector Programming Guide* provides conceptual and technical information for developers who want to create *connector flows*, the software components that link external systems to BusinessWare. This chapter introduces some of the key concepts developers should understand before creating custom connectors, including definitions of terms such as "connector flows" and "connection models." See *BusinessWare Foundations* for additional background information about BusinessWare.

This chapter contains:

BUSINESSWARE CONNECTORS

BusinessWare is an event-driven publish-subscribe distributed computing system. That means that for an external system to become part of a BusinessWare solution, the data with which the external system inherently deals must be provided to BusinessWare in the form that BusinessWare can understand, that is, as BusinessWare *events*.

That's essentially the role of a BusinessWare connector. A connector is a software component that enables BusinessWare and external systems to interact. Connectors are software components that can take data from an external system (external to BusinessWare) and turn the data into BusinessWare events for further processing, or can receive BusinessWare events and export them to an external system. Specifically, connectors can be one of three general types:

- A *source connector*, which obtains data from an external system, converts data into events, and sends events to other flows.
- A *target connector* receives events from other flows and passes the data to the external system.
- A *source-target connector* both sends and receives events. For the most part, source-target connectors also translate or transform events, hence they are typically referred to as *transformers*.

CONNECTORS AS A TYPE OF FLOW

At a lower level, “connectors” describes a particular type of software component called a *flow*—an object that can that can receive, process, and send events because it inherits these specific characteristics from the Vitria Flow class. The three types of connectors just described are more precisely referred to as three different types of flows.

Connector flows are not the only type of flows. *Channel sources* and *channel targets* are also flows; these flows communicate with BusinessWare *channels*—persistent BusinessWare structures that hold event information. (For more information, see the *BusinessWare Programming Guide*.) Channel target flows and channel source flows inherit their basic characteristics from the ChannelFlow class. A ChannelFlow communicates with a single channel. A ChannelFlow object either publishes to a channel (and is thus a target flow) or subscribes to a channel (and is thus a source flow), but not both.

Multi-threaded subscriber is like a channel source in that it subscribes to a channel and sends events that it receives from the channel to the next flow in a connection model, but multi-threaded subscriber enables multiple instances of a connection model to run concurrently (providing concurrent event processing); multi-threaded subscriber can dispatch events to different instances. (See *BusinessWare Connection Modeling Guide* for more information.)

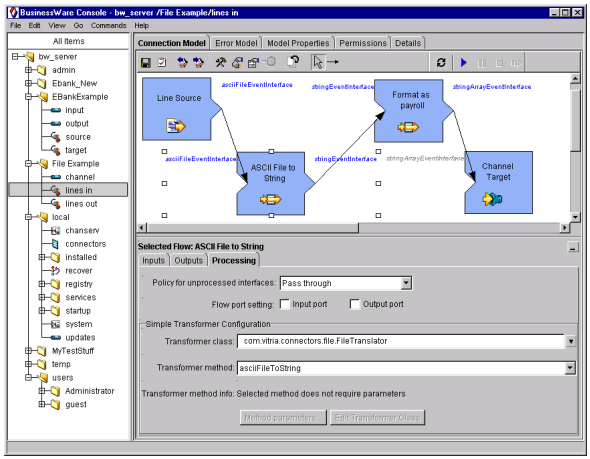
CONNECTION MODELS

Flows of all types may be connected together into a directed graph—a *connection model*—to process and distribute events. A connection model transports data, in the form of events, between end points. An end point might be a BusinessWare channel, or an external system, such as a database, a file system, or a front office application.

In fact, flows function only within the context of a connection model; as software components, they cannot run without the underlying services provided by the connection model and the associated infrastructure provided by the BusinessWare runtime environment. Flows package discrete functionality into re-usable components that business users can customize prior to runtime in the context of a specific connection model

Connection models and the connector flows and transformers that comprise them can be manipulated visually in the BusinesWare Console, as shown in the screenshot.

Connection models can include a variety of flow types needed to perform a range of tasks, from communicating with channels, transforming events, and inspecting data. In the example below, the lines out connection model includes two end points, a Line Source Flow and a Channel Target flow. In between are two transformers, ASCII to String and Format as Payroll.



Integrating an external system with BusinessWare often requires two connection models—one to handle input to BusinessWare (in the screenshot, the “lines in” connection model that’s displayed in the Console) from the external system, and one to handle input from BusinessWare to the external system (the “lines out” connection model).

CONNECTORS BUNDLED WITH BUSINESSWARE

BusinessWare is bundled with several source connector flows, target connector flows, and transformers. These are installed when BusinessWare is installed; you can access them through the BusinessWare Console. Business analysts, developers, and others can incorporate these as needed in their own connection models.

In addition, BusinessWare includes several *connection model templates*—pre-wired connection models with some basic combinations of connector flows, transformer flows, and channels that can be used as starting points for creating new connection models.

For example, the File to Channel template, available in the Console, provides a File Source flow, a Simple Data Transformer, and Channel Target flow, all wired together in the Connection Modeler, ready for configuration.

Here are brief descriptions of some of the connector flows, transformers (source-target flows), and transformation tools included with BusinessWare:

- The File Source connector is a source flow connector. The File Source Connector retrieves files from a file system and creates events using the file system data. It does not receive events from other flows.
- The File Target connector is a target flow. The File Target connector receives events from another flow and writes the event data out to files in the file system. It does not send any events to other flows.
- The Simple Data Transformer is a source-target flow. The Simple Data Transformer receives events, transforms them into different types of events, and sends the transformed events to another flow (or to other flows). (Source-target flows don't have to transform events, but they usually do.) More specifically, the Simple Data Transformer accepts a class and a method name, and calls that method when it receives events. It takes the return value from the method, and sends it to its target list of flows (the list of flows that receive events from the Simple Data Transformer).
- The RDBMS connector is a source-target flow. It can transform events, but it often just moves events through a connection model in some way. For example, the RDBMS connector can obtain a result set from an external system and then publish the result set onto a channel, without any transformation.
- BusinessWare Transformer— A comprehensive GUI tool that provides numerous built-in features. Vitria recommends using this for most all your data and event transformation needs.
- Spreadsheet Transformer — Transforms events using a spreadsheet to define transformation rules.

These are just some of the many connector flows and transformers bundled with BusinessWare. See the *BusinessWare Transformer Guide* and the *BusinessWare Connection Modeling Guide* for complete information about using these connectors and transfomers in the context of connection models.

PACKAGED CONNECTORS

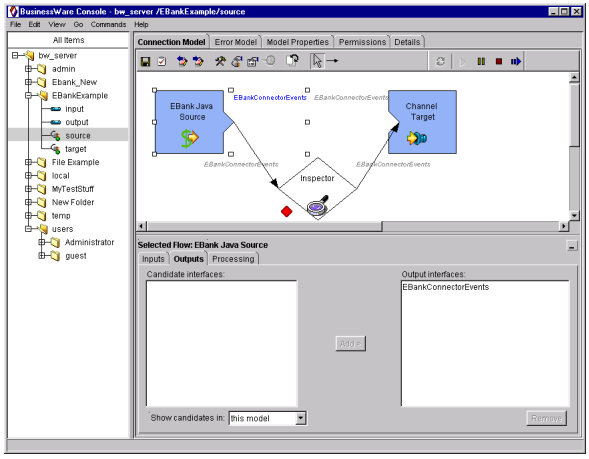
In addition to bundled connectors, Vitria also offers special optional packaged connectors designed to integrate enterprise application suites, database management systems, such as Oracle, SQL Server, PeopleSoft, SAP and a wide array of other software. New connectors are being released frequently; see Vitria's web site for a current list.

CUSTOM CONNECTORS

In addition to offering bundled and packaged connectors, Vitria exposes several interfaces and classes in BusinessWare that developers can use to create new connector flows for external systems or applications not covered by one of these other two approaches. This guide provides information primarily for creating such custom source connector flows and target connector flows; in addition, you'll find some information about transformer (source-target connector) flows.

In addition to the interfaces and classes that you'll use in your own implementations, Vitria provides several utility programs that will facilitate your development efforts. You'll find information about these topics throughout the remainder of this Guide. (See the *BusinessWare Programming Reference* for hyperlinked documentation for Java, C++, and IDL classes and interfaces.)

The EBankJavaSource connector, shown in the connection model in the screenshot below, is an example of a custom-developed connector:



NEXT STEPS

Before developing a custom connector flow, developers should review the current release of the *BusinessWare Connection Modeling Guide* and explore the options in the BusinessWare Console to see if a connector flow, connection model, or template already exists that solves the connectivity problem.

To get started developing, you should:

- Gain an understanding of BusinessWare connector architecture by reading the next chapter. Refer to the *BusinessWare Foundations* manual and the *BusinessWare Programming Reference* for additional information about the architecture.
- Install and configure BusinesWare on your development platform. See the *BusinessWare Installation Guide for Windows NT* for details. See Chapter 3, "Design and Development Guidelines" for development platform requirements.
- Install and step through the *EBank Sample*—it takes about an hour—to give yourself hands-on exposure to all the concepts covered in this guide, including: writing a flow with calls to external APIs, implementing a transaction resource, defining events in IDL, and other key topics.
- Review the *Connector Certification Guidelines for BusinessWare* for complete details on ensuring that your connector meets all Vitria requirements.

2 ARCHITECTURE OVERVIEW

Before writing a custom connector, developers should have an understanding of BusinessWare and connectors from several different frames of reference, including: how connectors fit into the BusinessWare architecture; how the classes and interfaces that comprise a connector flow are put together; and from a functional perspective, what happens at runtime. This chapter addresses these topics and includes the following sections:

- BusinessWare Basics
- Supporting Infrastructure
- Development Architecture

BUSINESSWARE BASICS

In simple terms, BusinessWare is an event-based distributed computing environment that builds on CORBA (common object request broker architecture) and other leading technology infrastructure to provide a platform for business-to-business application integration. Key components include the Communicator server, which provides high-speed asynchronous event-based communication between various BusinessWare components; and the BusinessWare server, which essentially controls all other servers and processes.

The BusinessWare server (bserv) is a CORBA server that provides naming, access control, persistence (metadata store), transactions, and activation—the BusinessWare server starts all other servers, processes, and containers, and monitors all running processes, including connection models.

The core BusinessWare architecture also includes a comprehensive framework (EMT Framework) that supports localization and internationalization. See *BusinessWare Foundations* for complete information about BusinessWare architecture.

Two areas of the BusinessWare architecture that developers must consider when they write a custom connector include the repository (metadata store) and the EMT Framework, as discussed below.

METADATA REPOSITORY

The BusinessWare repository stores (persists) metadata about events; event information is stored using IDL (interface definition language) or ODL (object definition language; see "Glossary" for definitions of terms).

The BusinessWare repository also stores information about BusinessWare objects—the flows, connection models, business process, and other objects that make up a BusinessWare solution. Developers creating new or custom connectors write these objects during the development process, and then install them in the repository.

See the *BusinessWare Metadata Guide* for more information about metadata. See Chapter 7, "Defining Event Interfaces" for information about working with metadata to create new event type definitions.

THE EMT (ERROR/MESSAGING/TRACING) FRAMEWORK

One basic principal of good software engineering is that text-based messages, including error messages and text labels, should be separate from the actual program code. Code designed with this principal in mind is much easier to maintain and to adapt to different languages or rapidly-changing business requirements.

Adhering to this principal, Vitria provides the Error, Messaging, and Tracing, or EMT Framework, a complete messaging infrastructure in BusinessWare that facilitates the efforts of both internal and external developers to deliver highly portable and easy to localize code. The EMT Framework provides the mechanism for separating text strings from the actual program code.

Briefly, implementing the EMT Framework involves creating a resource file—a text file that contains all the text strings for the system (see Figure 2-1 on page)—and then compiling the file using a special utility program provided by Vitria (rescomp, or "resource compiler"). The resource compiler generates several other files that you then compile and link with several other BusinessWare classes, ultimately producing a shared library file (a .dll or .so) called a "locale bundle." This locale bundle essentially comprises a messaging API to which you embed calls in your connector source code.

To change the text messages (display, error, label, and so on), you simply change the text in the resource file, regenerate, and re-compile. Resource files map names to strings, as shown in this example from the *EBank Connector Sample*:

Figure 2-1 Resource Text File for EMT Framework

For information about creating the text file, see the *BusinessWare Programming Guide*, Chapter 23, Error, Message, and Tracing Framework.

SUPPORTING INFRASTRUCTURE

Connector flows are software components that inherit many basic characteristics from the Vitria flow and connector APIs. A flow can receive, process, and send events; depending upon which of these functions a flow performs, it will be a source, target, or source-target flow.

Flows operate in the context of a connection model; connection models run in container servers, provided by the BusinessWare server (bserv). When a connection model is started, bserv instantiates a container server in which to run the connection model (if one is not already started); bserv also creates a Flow Manager, which is an object that controls all the flows that comprise a connection model. Flow Manager has an associated Transaction Service and an Error-handling Model. These infrastructure facilities are discussed in this section.

ERROR MODEL (ERROR GRAPH)

The connector runtime provides a built-in error model—a flow graph comparable to a connection model—with every connection model. As shown in the screenshot, the error model usually consists of an Event logger flow and a StopOn flow.

- An Event Logger flow converts error messages to events. It passes these error events to other flows that can take the appropriate error recovery action and logs messages to any specified log files.
- A StopOn flow is a Simple Data Transformer flow. By default, it is set to stop the connection model when specified errors occur.

The architecture makes no assumptions about what actions can occur within this built-in model; the model designer (business analyst, other users) must customize the error model to enforce the appropriate error-handling policies.

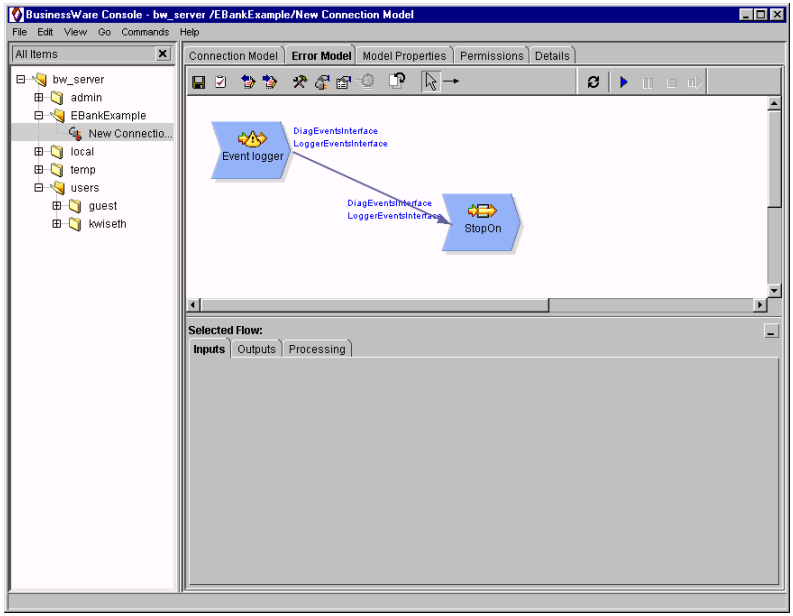


Figure 2-2 Error Model as displayed in the BusinessWare Console

Specifically, business analysts or other end-users can change the settings for Event logger and StopOn flow, delete them, or add new flows to the error model. They can set the StopOn flow to RestartOn, AbortOn, or SkipOn specific events. (RestartOn will stop the connection model for a period of time and then restart. AbortOn will abort the current transaction but will not stop the connection model. SkipOn ignores a specified event; for instance, you might use SkipOn for an innocuous event that you aren't handling in your regular model but don't want to stop or abort processing because of the event.)

Furthermore, developers can write custom methods to handle error events as they see fit. For details, see "Customizing Methods" on page 93.

To send errors to the error model from a custom connector flow, developers must write error-handling code by calling the Logger interface as discussed in "Log Levels" on page 89.

TRANSACTION ARCHITECTURE

The BusinessWare platform includes an embedded transaction service that manages and coordinates transactions. A *transaction* is a logical unit of work in which multiple changes to distributed databases (or other resources) all occur or none occur.

By definition, the resources involved in transactions must maintain their "ACID" properties; that is, the transaction is *atomic* (all or nothing, as just described). Furthermore, data on all resources must always be *consistent*, regardless of whether the transaction commits or aborts. Multiple transactions running concurrently on the same set of resources must occur as if in *isolation*. And finally, once a transaction commits, changes to the resources involved must be *durable*, regardless of any system failure after that point. (See Chapter 8, "Writing a Transaction Resource" and *BusinessWare Foundations* for additional information about transactions.)

The two-phase commit protocol is a way to coordinate the participants in a distributed transaction so that the resources involved in a transaction maintain these characteristics (atomicity, consistency, isolation, durability). Transaction services typically implement this protocol through a transaction manager that coordinates all the resources involved in a transaction by by means of a 'pre-commit' phase: in essence, all resources can tentatively make changes before committing.

Every connection model inherently has a transaction service associated with it. To use the transaction service, developers must create a *transaction resource*—a Java or C++ class associated with the flow that implements the specifics of the external API's transactional semantics.

Developers create a transaction resource for any flow that will engage in transactions. The transaction resource acts as an interface between the BusinessWare transactions and the external system's transactions. The transaction resource contains all the logic to commit or abort transactions on the external system.

Transactions in BusinessWare are implicit: Unlike other transaction processing systems that may require explicit commands to start or end a transaction, BusinessWare starts a new transaction automatically for each and every connection model whenever:

- a connection model is started
- a transaction is aborted
- a transaction is committed

The transaction service supports one-phase transactions as well as two-phase transactions, so if the external system doesn't support two-phase commit protocol, you can still include the external system in transactions.

See Chapter 8, "Writing a Transaction Resource" for detailed information about implementing support for transactions in a connector flow and for additional details about one-phase and two-phase transaction resource implementations.

FLOW MANAGEMENT

Two or more flows that are wired together comprise a connection model (also referred to as a “flow graph,” or simply “model” or “graph”). A connection model can be as simple as a single source flow and a single target, or may comprise numerous flows and transformers.

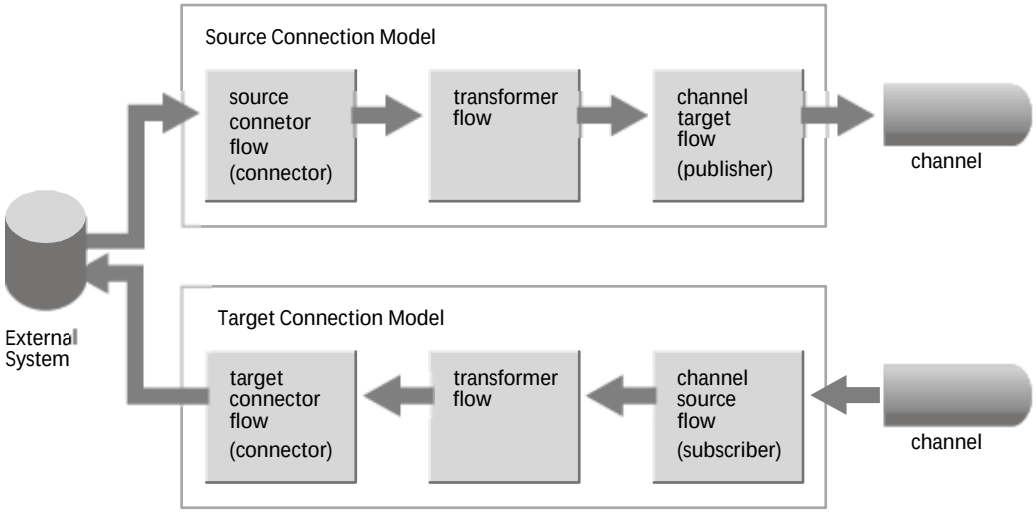


Figure 2-3 Canonical Connection Models

The figure shows two generalized connection models, one that picks up events from an external system (by means of a source flow) and one that sends events to an external system, by means of a target connector flow.

Events are moved into a connection model in one of two ways: asynchronously, using an underlying ‘push’ mechanism; or synchronously, using an underlying ‘request-reply’ mechanism. Either way, within the connection model, events are synchronously pushed from flow to flow, from the source connector flow to the ultimate target. The last flow in the model returns back to the source. Here’s a more detailed description:

1. The first flow in the model receives the event, perhaps from a channel or an external trigger. It then calls its own `doPush()` method to deliver this event to the next flow in the model.
2. The next flow receives the event, processes it, and calls its own `doPush()` method.

3. When the event reaches the last flow and it has successfully processed the event, each flow returns until the original `doPush()` call of the source flow returns.

4. At this point, because the event has successfully traversed the entire connection model, the source flow can call `commit()` on the Transaction Service (if the commitAfter_ parameter (boolean) was set to true.) This is typically done within the source flow's `doPush()` method.

The transaction service then launches into its two-phase (or one-phase, or mixed two-phase and one-phase) logistics with all the transaction resources that it has from this model. See Chapter 8, "Writing a Transaction Resource" for additional information about transaction processing in the context of a BusinessWare connector.

Also behind the scenes, some of the the key software components that work together to provide functionality across a connection model include:

- The Flow Manager, the object in charge of the connection model. Each connection model has its own flow manager.
- The Flow Environment (Flow Env), a mechanism controlled by the Flow Manager that holds references to the key services provided by BusinessWare infrastructure, including:
  - Transaction Service (described in "Transaction Architecture" on page 11)
  - Logger, which is used by flows in the connection model to log errors and messages and send them to log files and elsewhere (including the Event Model).

The Flow Manager wires the runtime flows together in the same way as the reps were wired together

- The flow adds a transaction resource to the Transaction Service. The transaction resource implements the required methods for participating in transactions.
- When the flow calls the Transaction Service to commit or abort a transaction, the Transaction Service calls the methods in the transaction resource to commit or abort.

The services provided by BusinessWare to connection models are passed as a reference in the Flow Env. (Transaction Service and Logger aren't the only services, but they're typically the only services at this time that developers need to specifically consider in the code they write.) At runtime, the Flow Manager passes the Flow Env from flow to flow inside a specific instance of a connection model, and the flow obtains the references it needs.

DEVELOPMENT ARCHITECTURE

The flow itself is implemented by means of several different files:

- The connector definition (*def*), a Java interface that contains method signatures to set and get any user-configurable properties (any information that a business analyst must enter before starting the connection model).
- The connector representation (*rep*), a Java class that handles and stores the properties listed in the def (implements the get and set methods) and creates the flow when the connection model is started. The rep also implements methods to import and export configuration information to and from the BusinessWare repository.
- The *flow*, a Java (or C++ class) in which all of the work of connecting to the external system and sending and receiving information is accomplished. The Flow implementation contains the code for starting and stopping, and for sending or receiving events from the external system. It's in the code for the flow that you'll invoke calls on the external API; add a transaction resource to the transaction service (if appropriate for the flow), and refer to the logger.
- A *BeanInfo class* that defines more user-friendly display information for the BusinessWare Console about the properties in a flow, such as property names and descriptions.

The relationships among the def, rep, rep BeanInfo, and flow files, and the Flow's location within a connection model is shown in this figure:

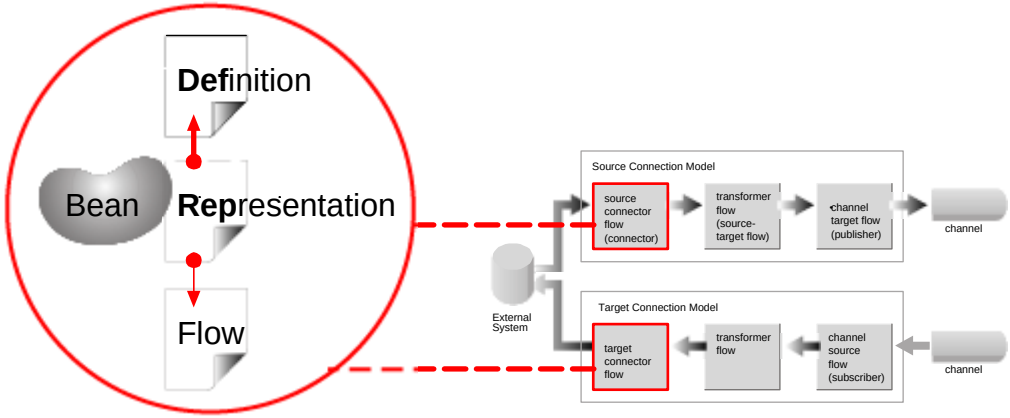


Figure 2-4 Def, Rep, BeanInfo, and Flow in context of connection model

When a connection model starts for the first time, bserv starts a Flow Manager; Flow Manager coordinates and manages the creation of all flows in any given connection model in the same general way, as described below:

1. Flow Manager creates the Reps for all flows in the connection model, starting with the Reps in the Error Model before moving to the Reps in the main model (connection model). Flow Manager sends the Flow Env to each Rep.
2. Flow Manager tells the Reps to read their propery values (import them) from the repository, which stores either the default values (for a new connection model) or the values that were last saved by a user of an existing connection model.
3. Flow Manager calls createFlow on the Rep.
4. The Rep creates the flow (passing it the Flow Env) and calls the flow's `init()` method.
5. The flow iniatializes itself, creating a transaction resource (if appropriate), and passing the resource to the transaction service.
6. The Flow Manager calls `startEvents()` on each flow, first in the error model and then in the main model. It begins with the last target flow and works backwards to the first flow source.
7. The flow connects to the external system (inside the `startEvents()` method). The flow handles events and data transfer with its external system.
8. Flow Manager calls stopEvents on each flow.
9. The flow disconnects from the external system (inside the `stopEvents()` method).

All these activities occur within the context of a connection model. If any flow in the model throws an error in `startEvents()`, the Flow Manager works back through the connection model and calls `stopEvents()` on all flows. If no errors are returned, the connection model is up and running; a green light is displayed in the Console namespace.

If there are errors, an error message displays and the small icon of the connection model (displayed in the Console namespace) turns yellow or red. (In addition, information about the error may be saved to the log files.) (See the *BusinessWare Connection Modeling Guide* and the *BusinessWare System Administration Guide* for details about runtime operations and other details from an end-user (business analyst) perspective.)

3 DESIGN AND DEVELOPMENT GUIDELINES

This chapter provides information about design considerations for connectors. Topics include:

- Design Considerations
- Getting Started

DESIGN CONSIDERATIONS

Designing a BusinessWare connector for an external system starts with a thorough analysis of the external system. You must learn everything you can about its characteristics because much of the work of creating a new connector involves making calls to the external API in your code, and turning the external data into BusinessWare events.

Here's a list of questions you should consider as you begin examining the external system:

- What kind of external system is it, and what are its basic characteristics? Is it a database, a file system, or a front-office application? Knowing the type of system to connect to will help determine the method to use in communicating with the system and the required API calls.
- What are the characteristics of the external system's API? Specifically,
  - Does the API support asynchronous or synchronous (request/reply) communications? For example, can the API notify the connector asynchronously when data has changed, or will the connector have to poll the system for changes?
  - Does the API return data (that can be used by a target connector to publish an event, making it a source-target flow)?
  - Does the API support C++ or Java?
  - Is the API thread-safe? If so, does it make sense to support multi-instancing in the connector?
  - Does the external system support transactions, and if so, does the system implement a one-phase or a two-phase commit protocol?

- How are transactions demarcated in the external system?
- How are transactions committed or aborted in the external system?

The characteristics of the external system will determine how the connector you create receives signals about changes to data, how it retrieves the changed data, how it modifies events in the third-party system, and how it will implement commits and aborts in the transaction resource.

With an understanding of the characteristics of the external system, you can start addressing questions specific to BusinessWare:

- Does the API provide access to metadata, and if it does, should you provide an IDL/ODL generator with your connector?
- How should you define the events? What type of data does the external system provide? Will the connector need to process email messages, sales orders, customer information, or raw data streams? Identify the type of data so that you can group the data type into events. Which will be the most appropriate way to model the events? IDL or ODL?
- Which type of connector is needed for your integration challenge? Do you need to create a source connector, target connector, or source-target connector? A source connector retrieves information from an external system, a target connector enters information into an external system, and a source-target connector both enters and retrieves information.
- What type or required or optional input is needed from connector users (business analysts or others who will interact with the connector)? Will users need to enter a database instance name or machine name to connect to a database? Will business users need to be able to configure polling frequency, or number of events to process before a commit?

  The information the user inputs will be turned into properties that the user sets through the BusinessWare Console on the flow property sheets.

For example, the API that's exposed to developers in the EBank example comprises three methods—balance, deposit, withdraw—as shown in this code excerpt:

```
public interface EBankCommands extends
   com.vitria.fc.object.Obj {
. . .
double balance(int account);
...
void deposit(int account, double amount);
...
```

```
void withdraw(int account, double amount);
...
```

CONNECTOR DESIGN GUIDELINES

By their very nature, external systems are beyond the control of the connector developer. To adapt easily to dynamically changing business scenarios, you should keep these guidelines in mind when you design and create connectors:

- Keep event code independent of the external system's data format—the data format of the external system could change. For instance, an event could consist of an object type, an action to be performed on the object, and a list of name/value pairs that holds the information about the object.
- Isolate code with specific tasks into functional components that you can reuse.
- Generalize the code you write as much as possible so that upgrades to the external system don't require new versions of your connector. For example, avoid hard-coding names of tables and other external system entities; if you do, when table or resource names change, you'll have to rewrite your code. Instead, reference table names dynamically in the connector code.
- Connect to the source or target only when the connection model starts (in the startEvents method); disconnect when the connection model is paused (in the stopEvents method). Doing otherwise—connecting each time a connector needs to poll—is both inefficient and non-standard.
- Don't read from and write to the external system from the same flow. Avoid writing source-target connectors that both read from and write to the external system; this sort of connector design breaks the idea of separate components and minimizes reuse.
- Although you should avoid writing source-target connectors that both read from and write to an external system, you can create source-target flows that you can embed within a specific connection model, such as these:
  - A source connector that receives events from other flows or other connection models: For example, you may want to notify a source connector of new information in the external system by sending it an event. (The Vantive source connector operates this way, receiving notification from the RDBMS connector.)
  - A target connector that sends events on to other flows or models: For example, a target connector that sends a "success" event once it has inserted data into the external system.

NAMING STANDARDS AND UI CONVENTIONS

- Capitalize all Property names (just the first letter in each property name). Property names should be short and descriptive—just a couple of words, not an entire sentence.
- Provide a short description (one to three sentences) for each property. This description should inform a knowledgeable user (a business analyst) the purpose of the property.
- Provide default values for any Property when it makes sense to do so. For example, if the third-party application for which you're creating the connector usually runs on IP port 1521, then your Rep should include this default value for the property.
- Check for invalid values when changing a property's value and throw a DiagError when appropriate.
- Implement custom property editors as needed. Users should only be presented with valid choices only; to do this, use custom property editors that are designed to accept legitimate values only.
- Use the processing tab to enable editing of properties related to processing or transformation. Properties that can change the behavior of the connector (unlike properties, such as name and password, that are more utility than function) may require a more sophisticated GUI than a property editor. Place such properties inside the processing tab, or provide modal buttons that provide dialog boxes.
- Provide Wizards when appropriate, such as for those tasks that require users to follow a predefined sequence of steps. For example, if you want a user to log in, select some objects in one place and then provide target or destination information, you can guide them through the necessary steps with a Wizard.
- Integrate connector-related GUIs with the console so that users don't have to run command-line utilities.
- Provide appropriate titles to your message boxes in the format <operation><object>. For example, "Saving Spreadsheet." Don't use generic words such as "Error," Warning, or Information.
- Keep the body of the message text in the message box to three (3) complete sentences or less. If you need more than three sentences, display a short message and tell the user to look in the logs for more detailed information.
- Create a 32x32 pixel icon (in .gif format) for each connector flow you create.
- Keep all connector code in a Java package that follows the naming convention com.*YourCompanyName*.*NameofConnector*(or application)

Event Names

Event names should be lower-case for the first word, initial-cap for every word after that. For example, dateEvent, anotherKindOfEvent, aThirdKindOfEvent.

File Names

Table 3-1 Naming Conventions for Java Files

	Source Flow	Target Flow	Source-Target Flow
Def	*name*SourceDef	*name*TargetDef	*name*SourceTargetDef
Rep	*name*SourceRep	*name*TargetRep	*name*SourceTargetRep
BeanInfo	*name*SourceRepBeanInfo	*name*TargetRepBeanInfo	*name*SourceTargetRepBeanInfo
Flow	*name*SourceFlow	*name*TargetFlow	*name*SourceTargetFlow
Transaction Resource	*name*SourceResource	*name*TargetResource	*name*SourceTargetResource

Module Names

Use initial capitals on every word; for example, *MyModule.*

GETTING STARTED

As mentioned earlier in this chapter, developers who want to create custom connector flows should first consult the *Connection Modeling Guide* and ensure that no connector flow exists to solve their connectivity problem.

For example, the BusinessWare Transformer is a powerful source-target connector flow that provides many built-in operations that you can use override to create your own transformations.

If you're a developer who wants to create a new connector flow to integrate BusinessWare with a specific third-party software package, be sure to review the *Connector Certification Guidelines for BusinessWare* for guidelines, packaging instructions, naming conventions, and the like.

Development Platform Setup

Your development environment should include the Java JDK 1.2. Minimum developer's workstation requirements include 256-Mbyte of RAM (512-MB recommended). Install the BusinessWare software on your development system according to the installation instructions. You'll need a C++ compiler, Java compiler, and Java Runtime, as well as the BusinessWare software.

The connector APIs are contained in the BusinessWare. jar file, located in the %VITRIA%\java\win32 (on Windows NT) or $VITRIA/java/sparc_solaris for Solaris; make sure your CLASSPATH is configured correctly. (If you install the core BusinessWare platform on your development machine, accepting the defaults, the CLASSPATH should be set correctly.)

CHECKLIST FOR DEVELOPING A CONNECTOR

Throughout this guide you'll find example code from the EBank Connector sample, which is installed with BusinessWare. The EBank Connector Sample includes the external system and an API for that system (a Java banking server application and a client that can deposit, withdraw, and check account balances). To facilitate your own understanding of the BusinessWare environment and connector implementation, Vitria recommends that you install and configure this sample application. See *EBank Connector Sample* for details.

Developing a connector encompasses three broad sets of development activities, starting with defining the events that your system will send or receive. To define the events, you must first examine the external API and identify the data types that the external system uses, and decide how best to define the data as BusinessWare events.

Providing transactional semantics for your connector flows will also be a big part of your development efforts. Although BusinessWare provides a built-in Transaction Service that automatically starts transactions on the connection model and that manages the pre-commit, commit, and abort activities on the flow, you as developer must determine whether your connector flow should have a transaction resource connected to it, and if so, you must write the transaction resource using the transactional semantics of the external API.

Thus, before you can do much of anything programmatically, you must focus on a thorough analysis of the external API and understand its characteristics with special attention to the concepts of events and transactions. With a firm grasp on the external API, you can begin defining the events and thinking about the overall design of the connector flows themselves.

The basic steps for developing connectors are listed below. However, note that many of these activities are concurrent or dove-tail into each other. For example, the code for your flow (step 2) will include code for handling events (step 3) and a reference to a transaction resource (step 3), so this isn't exactly a linear process. Before attempting to build your own connectors, read Chapter 2, "Architecture Overview" to learn about how connectors and other flows work within a connection model.

1. Design the connector after you analyze the external system, noting especially the characteristics of the API and data types that the system uses. See "Design and Development Guidelines" on page 17 before you get started.
2. Write the connector flow.
    a. Identify the portions of the connector that you can generate by using the Connector Generator Tool and the portions that you will need to write. See "Creating Java Components for a Connector Flow" on page 25.
    b. Implement the portions of the flow that start, pause, and connect to the external system, as described in "Introduction to Java Connector Flows" on page 45. Concurrent with this task, you should define the data interfaces and event interfaces for the connector. See "Defining Event Interfaces" on page 65.
    c. Write code to exchange data with the external system, to send and receive events, and to transform events, as described in "Writing Java Connectors" on page 45.
    d. Identify the errors and exceptions the connector needs to handle, and implement the methods suggested in "Handling Errors" on page 89.
3. To ensure reliable data transfer from the external system to the flow, write a transaction resource to handle transactions as described in "Writing a Transaction Resource" on page 75.
4. Install the connector into the BusinessWare environment as described in "Installing Connectors and Connection Models" on page 101.

The remainder of this guide provides information for performing these steps.

4 CREATING JAVA COMPONENTS FOR A CONNECTOR FLOW

This chapter guides you through the process of writing the Java code portions of a connector flow, specifically, the code for the Def, the Rep, and the BeanInfo. These elements work together to provide a visual component that end-users, such as business analysts or administrators, can configure as part of a connector model in the BusinessWare Console as shown in the screenshot on page 5.

In addition, you'll see where the actual code for the flow (which is covered in Chapter 5, "Writing Java Connectors" or Chapter 6, "Writing C++ Connectors") fits in to this scheme. Topics in this chapter include:

- Creating the Java components for the Flow
  - Step 1: Write the Flow Def
  - Step 2: Write the Flow Rep
  - Step 3: Write the BeanInfo
- Simplified Development using the Connector Generator

Unless otherwise noted, all code listings in this chapter are excerpts from the *EBank Connector Sample*.

CREATING THE JAVA COMPONENTS FOR THE FLOW

Writing the code for a connector flow involves writing several basic Java interface and class files that inherit from several other BusinessWare base classes and interfaces. You can automate the writing of these files and provide the basic structure for the flow class itself by using the Connector Generator Tool provided by Vitria; see "Simplified Development using the Connector Generator" on page 37 for information about using this tool. Even if you choose to use the Connector Generator Tool—and Vitria recommends that you do so, for simplicity's sake—you can read through this section first to see the relationships among the various files that the tool produces.

If you'll be incorporating data from an external system for which no BusinessWare event types exist in the BusinessWare repository, you must also define these events and load them into the repository in order for your code to generate or receive these events. See "Defining Event Interfaces" on page 65 for information.

STEP 1: WRITE THE FLOW DEF

Every connector flow must have a definition, or Def file, that defines the list of configurable properties that will comprise the flow. The Def file is a Java interface file that inherits from one of three specific BusinessWare class files:

- `ConnectorSourceDef`
- `ConnectorTargetDef`
- `ConnectorSourceTargetDef`

If you're creating a source flow, you'll extend the ConnectorSourceDef class; for a target flow, you'll extend the ConnectorTargetDef; for a source-target flow, you'll extend the ConnectorSourceTargetDef. All these classes are contained in the `com.vitria.connectors.common` package, so your Def file must start by importing this package.

NOTE: All the code you write for the flow def, rep, BeanInfo, and the flow itself should belong to the same Java package. As you'll see from the example code in this chapter, the package name for all Java code related to the *EBank Connector Sample* is contained in a package called "jebank" (Java EBank).

As with other Java interface files, the Def file contains abstract methods only—method headers, including parameters, without implementation. The Def file's job is to provide the list of accessor methods for setting and getting properties on the flow's property sheet. For each property that you want to expose to end-users, create a get method and a set method signature in the Def; do not use any of the reserved words in the table, however.

Table 4-1 Reserved Words

`commitAfter`	`name`	`targetEventInterfaceList`
`concurrent`	`resolver`	`targetList`
`logLevel`	`sourceEventInterfaceList`	`unknownEventFlag`

In addition, follow the naming convention for get and set methods, that is, all lower case “get” or “set” and initial capital for all other words—note “setPollRate” and “getPollRate” in the listing below.

Here’s a complete example listing of a source flow connector def. As you can see the file imports the entire com.vitria.connectors.common.* package, and then continues with the get and set method signatures:

```
package jebank;

import com.vitria.connectors.common.*;

public interface EBankJavaSourceDef extends
   ConnectorSourceDef {
   public String getIorFile();
   public void setIorFile(String iorFile);
   public int getAccount();
   public void setAccount(int account);
   public int getPollRate();
   public void setPollRate(int pollRate);
}
```

Once you compile this interface, you can implement its methods in a Rep as detailed in the next section.

STEP 2: WRITE THE FLOW REP

The Rep (or “representation”) is a Java class that implements the interface in the Def. The Rep extends the BusinessWare base class appropriate for a source, target, or source-target flow, respectively. The Rep has several other jobs, including creating the Flow object, passing the Flow Env to the Flow object, importing a list of events that the flow can receive, exporting a list of events that the flow can send, and providing a representation for the flow in the BusinessWare repository. The next several sub-sections show you how to write code that supports all these tasks.

Extend the Appropriate Connector Class

Depending on whether the flow will be a source, a target, or source-target connector flow, the code for your Rep must extend one of the following classes in the `com.vitria.connectors.common` package:

- `ConnectorSourceRep`

- `ConnectorTargetRep`
- `ConnectorSourceTargetRep`

Here's an example from the top portion of a Java class that implements the example shown in Step 1: Write the Flow Def. As you can see, this Java class extends the `ConnectorSourceRep` class and implements the `EBankJavaSourceDef` interface.

```
package jebank;

import com.vitria.diag.TextMessage;
import com.vitria.fc.io.*;
import com.vitria.fc.flow.*;
import com.vitria.connectors.common.*;

import com.vitria.fc.data.List;

public class  EBankJavaSourceRep extends ConnectorSourceRep
   implements EBankJavaSourceDef {
...
```

Next in the code you must create instance variables for each property in the Rep. For example, continuing with the EBank example:

```
...
protected String iorFile_;
protected int account_;
protected int pollRate_ = 5000;
...
```

Implement the get and set Methods

In addition to declaring the instance variables and constants in the Rep class, you must provide bodies for the methods listed in the Def; here are the implementations for the six `get` and `set` methods listed in EBankJavaSource Def interface on page 27:

```
...
public String getIorFile(){
      return iorFile_;
   }
public void setIorFile(String iorFile){
      iorFile_ = iorFile;
```

```
   }
public int getAccount(){
      return account_;
   }
public void setAccount(int account){
      account_ = account;
   }
public int getPollRate(){
      return pollRate_;
   }
public void setPollRate(int pollRate){
      pollRate_ = pollRate;
   }
...
```

Enable Properties to be Stored in the Repository

The Rep is also responsible for providing methods that will be used at runtime to store and retrieve the properties in BusinessWare. These are found in the `Exportable` interface (located in the `com.vitria.fc.io.*` package that was imported into the Rep file).

In addition, the Rep needs a way to reference the Def in the Repository. When the Flow Manager starts to create the flow, it looks in the Repository for the Def. The mechanism for this is a constant, in this example, the "EBANKJAVASOURCETAG," that locates the appropriate method to create new instances of this particular flow's Rep.

So one task is to override the getExportFormat() method and return a unique string of the form `com/my/company/NameOfDefClass:1.0`.

Here's the example from the EBank source flow Rep code:

```
...
public static final String EBANKJAVASOURCETAG =
   "EBankJavaSourceDef:1.0";
...
public String getExportFormat(){
      return EBankJavaSourceRep.EBANKJAVASOURCETAG;
   }
...
```

Override the method void exportTo(com.vitria.fc.io.ExportStream stream) by first calling super.exportTo(stream); and then exporting all the properties:

```
...
public void exportTo(ExportStream stream) throws
   ExportException {
       super.exportTo(stream);

      stream.writeString(iorFile_);
      stream.writeInt(account_);
      stream.writeInt(pollRate_);
   }
...
```

Next, you must override void importFrom(com.vitria.fc.io.ImportStream stream) by first calling super.importFrom(stream); and then importing all the properties in the same order that you exported them, as shown in this sample:

```
public void importFrom(ImportStream stream) throws
   ImportException {
       super.importFrom(stream);

      iorFile_ = stream.readString();
      account_ = stream.readInt();
      pollRate_ = stream.readInt();
}
...
```

In your code, be sure you import the properties in the same order as you exported them.

The Rep must also provide a reference to a create method; reference to the method is stored in the repository, and when an end-user clicks on the icon for your custom flow (in the Flow Palette of the Console), Flow Manager will use the method to return an instance of the flow Rep (the visual component) in the Console.

```
...
public static EBankJavaSourceDef createEBankJavaSourceDef(){
      EBankJavaSourceRep r = new EBankJavaSourceRep();
      r.init();
      return r;
   }
```

```
public static EBankJavaSourceDef
   createEBankJavaSourceDef(ImportStream s){
      EBankJavaSourceRep r = new EBankJavaSourceRep();
      r.init();
return r;
   }
...
```

Provide Lists of Event Interfaces

Another one of the Rep's tasks is to provide source and target event interface lists, if appropriate, to the end user. Target lists are used for event type checking at runtime. Event interface lists for connectors and transformers are displayed to show what types of events they can send and receive. The set of events a connector or transformer can send is called the *source list*, and the set of events it can receive is called the *target list*.

- Call createList if you want to enable users to specify different events.
- Call createListReadOnly if you want to prevent users from specifying different events.

The methods you'll override are in the `com.vitria.fc.data.List` package (which is imported in the top of the Rep file as shown in the listing.)

If the Rep is for a source flow, you must implement the `getSourceEventInterfaceList()` method to provide a list of event types that the flow can send (a source list).

Here's an example from the EBankJavaSourceRep class that creates a List object that will contain the list for the flow:

```
...
public List getSourceEventInterfaceList(){
      String [] eventNames =
   {"vtEBankConnectorModule.EBankConnectorEvents"};
      return ConnUtil.toEventInterfaceList(eventNames,
   resolver_);
   }
...
```

- For a target connector flow, you'll override com.vitria.fc.data.List `getTargetEventInterfaceList()`.
- For a source-target connector (usually a transformer) implement both methods.

To prevent users from editing event types in the Console, use the read only list.

Note the variable `resolver_` in the listing above: this handles the switch between the editable and the registered version of an event. (The registered version is used by BusinessWare at runtime; the editable version is visible in the Console.)

Instantiate the Flow

Another important tasks of the Rep is to instantiate (create the instance of) the Flow itself:

```
....
public FlowSource createFlowSource(FlowEnv env){
      EBankJavaSource out = new EBankJavaSource();
      out.init(this, env);
      return out;
   }
...
```

To specify the type of flow to create, write one of these methods in the rep:

- For a source flow, you must override com.vitria.fc.flow.FlowSource `createFlowSource`(com.vitria.fc.flow.FlowEnv) in the `FlowSourceDef`
- For a target flow, override com.vitria.fc.flow.FlowTarget `createFlowTarget`(com.vitria.fc.flow.FlowEnv) in the `FlowTargetDef` interface
- For a source-target flow, you must override com.vitria.fc.flow.Flow `createFlow`(com.vitria.fc.flow.FlowEnv) in the `FlowDef` interface

Regardless of which type of flow you're instantiating (source, target, or source-target), you instantiate the flow and then call init(this, env); on the flow in the initialization as shown in this code segment from the EBank sample (EBankJavaSourceRep.java):

```
...
public FlowSource createFlowSource(FlowEnv env){
      EBankJavaSource out = new EBankJavaSource();
      out.init(this, env);
   return out;
   }
...
```

If you'll be implementing the flow in C++ (rather than Java), you must create the flow in the CPPFlow wrapper and return the CPPFlow (you must also import com.vitria.connectors.flowbridge.*; in the Java file for a Rep that's used in conjunction with a C++ flow). Here's an example of the Java code that would instantiate a C++ version of the flow:

```
...
public FlowSource createFlowSource(FlowEnv env){
      CPPFlow cppFlow = new CPPFlow(this, env);
      cppFlow.setSharedLibName("vtEBank");
      cppFlow.setMethodName("CreateEBankSource");
      String [] params = new String[3];
      params[0] = "" + iorFile_;
      params[1] = "" + account_;
      params[2] = "" + pollRate_;
      cppFlow.setParams(params);
      cppFlow.init(env);
      return cppFlow;
   }
...
```

At runtime, this code serves to create an object of the appropriate flow type and initialize the object.

In addition, the Flow Manager will pass in the Flow Env to the object, to provide it with the references to flow registry and other infrastructure. The flow can use the `FlowEnv` object to access the Flow Controller, Flow Status Update, Transaction Service, or the Logger

The Flow Manager calls the createFlow method, with the `FlowEnv` object as a parameter, to create a new instance of the flow associated with this rep.

In this example, the EBankJavaSource is the flow that gets instantiated (you'll see the code EBankJavaSource in Chapter 5). After creating an instance of the flow object named EBankJavaSource, the Flow Env is passed to the flow, and the flow is returned to the caller (the Flow Manager).

Multi-instance Flows

If the flow will be multi-instanced, you must override the getConcurrent() method in the Rep, (return true instead of the default false):

```
public boolean getConcurrent() {
    return true;
  }
```

See “Multi-instance Connection Models” on page 99 and see *BusinessWare Connection Modeling Guide* for additional information.

Load the Locale Bundle for the EMT Framework

As discussed in Chapter 2, “Architecture Overview,” BusinessWare provides a complete messaging framework that enables developers to keep messages, errors, user-interface labels, and other text elements isolated from the body of code, ensuring that their applications can easily be localized to different languages. (See “The EMT (Error/Messaging/Tracing) Framework” on page 8 for more information.) The implementation of the framework for a specific application is called the “locale bundle,” a shared library of message text of all kinds that will be used by the application. The Rep file must include a static block that loads this locale bundle, as in this example:

```
...
static {
   com.vitria.diag.TextMessage.loadBundle("mybundle");
}
...
```

The name of the bundle is the name of the shared library with any version numbers and suffixes removed. For example, the EBank locale bundle for the Windows NT platform is a dynamic link library named vtEBankLocale3_o2r.dll; in the code for the Rep, this is referenced as simply “vtEBankLocale3,” as shown here:

```
...
static {
      TextMessage.loadBundle("vtEBankLocale3");
}
...
```

However, as with any development project, the text of various messages and other text that you might want to include in the resource file will likely change as you write and fine-tune your code. For that reason, Vitria recommends using simple placeholder messages during the development process, and leaving the implementation of the EMT framework to the end of the process, when you’ve fully debugged and completed the connector. See *BusinessWare Programming Guide* for information about creating and compiling the resource file.

STEP 3: WRITE THE BEANINFO

The BeanInfo is the last of the Java-specific files you need to create; the BeanInfo displays the properties, icons, and flow names in the Console for the Rep you create. The BeanInfo enables you to present the underlying methods to your end-users in a more readable (to non-programmers) fashion, as well as provide an icon to represent what will ultimately be a flow to the user in the Console.

As with the Java source code files you create for the Def and the Rep, the BeanInfo must also adhere to specific naming conventions. Specifically, you must prefix the name "BeanInfo" with the name of the flow Rep itself. For example, if the name of the Rep is EBankJavaSourceRep, you must name the BeanInfo class "EBankJavaSourceRepBeanInfo." Here's the top portion of the BeanInfo file for the EBank source connector:

```
package jebank;


import com.vitria.fc.data.BeanLib;
import com.vitria.diag.TextMessage;
import java.beans.SimpleBeanInfo;
import java.beans.PropertyDescriptor;
import java.beans.BeanDescriptor;
import java.beans.BeanInfo;

public class EBankJavaSourceRepBeanInfo extends
   SimpleBeanInfo {
   . . .
```

You must import the classes from the java.beans package and the two Vitria packages as shown in the listing, and then you must extend the `java.beans.SimpleBeanInfo` class.

Next, override the java.beans.PropertyDescriptor[] getPropertyDescriptors() and make the appropriate setDisplayName, setShortDescription method calls for each configurable property that you want to expose to end-users. (Note that in the code listing below, the names are being handled by the locale bundle.)

```
public PropertyDescriptor[] getPropertyDescriptors(){
       if (display_ == null){
         display_ = (BeanLib.getInterfaceProperties(EBankJavaSourceRep.class));

         for  (int i = 0; i<display_.length; i++){
            if (display_[i].getName().equals("iorFile")){
```

```
display_[i].setDisplayName(TextMessage.formatMessage(ebanklocale.CommonEBankConn
ectorMessages.IORShort));

display_[i].setShortDescription(TextMessage.formatMessage(ebanklocale.CommonEBan
kConnectorMessages.IORLong));
          }
          else if (display_[i].getName().equals("account")){

display_[i].setDisplayName(TextMessage.formatMessage(ebanklocale.EBankSourceConn
ectorMessages.AccountShort));

display_[i].setShortDescription(TextMessage.formatMessage(ebanklocale.EBankSourc
eConnectorMessages.AccountLong));
          }
          else if (display_[i].getName().equals("pollRate")){

display_[i].setDisplayName(TextMessage.formatMessage(ebanklocale.EBankSourceConn
ectorMessages.PollRateShort));

display_[i].setShortDescription(TextMessage.formatMessage(ebanklocale.EBankSourc
eConnectorMessages.PollRateLong));
 . . .
```

To set the display name for the flow, you must also override `java.beans.BeanDescriptor getBeanDescriptor()` in your BeanInfo class:

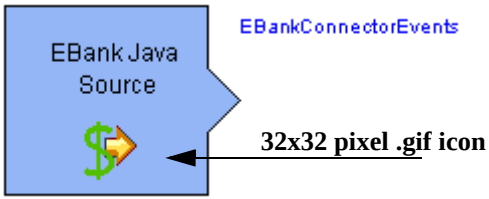
```
...
public BeanDescriptor getBeanDescriptor(){
      if (beanDescriptor_ == null){
         beanDescriptor_ = new
   BeanDescriptor(EBankJavaSourceRep.class);

   beanDescriptor_.setDisplayName(TextMessage.formatMessage(
   ebanklocale.EBankSourceConnectorMessages.JavaDisplayName)
   );
      }

      return beanDescriptor_;
...
```

To display an icon in the Console, you must override the java.awt.Image getIcon(int iconKind) method (only 32x32 icons are supported).

```
...
public java.awt.Image getIcon(int iconKind){
      if (iconKind == BeanInfo.ICON_COLOR_32x32){
         return BeanLib.getImage(this, "ebsource.gif");
      }
      else {
```

```
        return null;
...
```

Without this code snippet above, the dollar sign in source connector shown below would not exist:



Once you've created the Def, the Rep, and the BeanInfo, you can load these Java components into the BusinessWare repository to see how things are shaping up, (as long as you also implement some skeleton code for the Flow). To do so, you must create an .ini file that loads this flow type in the repository; see "How to Use the Connector Generator Tool (Wizard)" on page 38.

JAVA FILE SUMMARY

Table 4-2 Naming Conventions for Developer-created (or generated) Java Files

	Source Flow	Target Flow	Source-Target Flow
Def	*name*SourceDef	*name*TargetDef	*name*SourceTargetDef
Rep	*name*SourceRep	*name*TargetRep	*name*SourceTargetRep
BeanInfo	*name*SourceRepBeanInfo	*name*TargetRepBeanInfo	*name*SourceTargetRepBeanInfo
Flow	*name*SourceFlow	*name*TargetFlow	*name*SourceTargetFlow

SIMPLIFIED DEVELOPMENT USING THE CONNECTOR GENERATOR

The code requirements for a Def, Rep, and BeanInfo are highly templated and standardized, which means that they lend themselves easily to being automatically generated by a programming tool. Vitria provides such a tool, the Connector Generator Tool, that lets you create these basic Java source code files easily and quickly. There are two versions of the tool that provide slightly different functionality:

- Connector Wizard, a graphical user interface wizard (com.vitria.connectors.generator.ConnectorWizard)

- Connector Generator, a command line version of the tool (com.vitria.connectors.generator.Connector Generator.

The Connector Wizard enables you to enter the names for your Java files and numerous other settings in a graphical user interface; it then generates the necessary Java (or Java and C++) files for you. In addition, you can save your entries as a special text-based file (a .gen file), which you can open again later in the tool (or in a text editor).

Once you have a .gen file, you can use the command-line version of the tool, using the name of the .gen file as an input argument, and generate all files or selected files. See "Using the Connector Generator Tool and the .gen File" on page 42 for additional details about the .gen file.

Specifically, both the Connector Generator Tool can automatically generate:

- The flow def (the Java interface created in Step 1: Write the Flow Def)
- The flow rep (the Java class file created in Step 2: Write the Flow Rep)
- The flow's BeanInfo object created in Step 3: Write the BeanInfo which controls property display in the BusinessWare Console property sheet
- A base class for the flow and a companion helper class (you'll need to fill-in the specifics as discussed in Introduction to Java Connector Flows in the next chapter)
- An initialization (`.ini`) file for the connector (to load the connector flow into the Console; see Chapter 11, "Installing Connectors and Connection Models.")

Once you've generated the helper files and the skeleton of the flow, you can then complete the implementation of the flow as described in Chapter 5, "Writing Java Connectors" (or in Chapter 6, "Writing C++ Connectors") and work on events, transactions, and other issues.

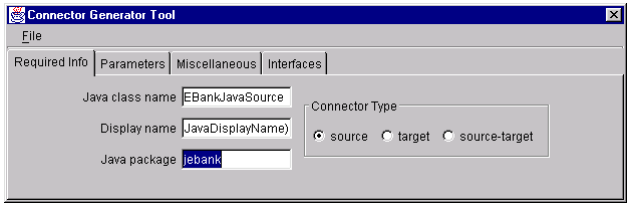
How to Use the Connector Generator Tool (Wizard)

The Connector Generator Tool is located in the %VITRIA%\bin\win32 or %VITRIA%\bin\TK sub-directory. For Windows NT, a batch file (connwiz.bat) is provided to launch the program; for Solaris, you'll find a script (connwiz.csh) that you can use to start the tool from a shell.

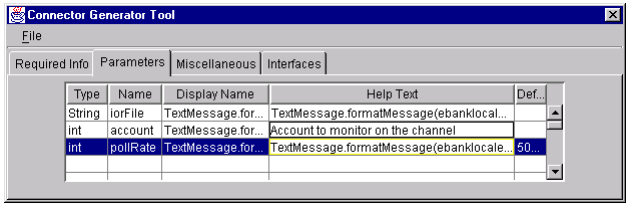
1. To start the wizard-version of the Connector Generator Tool from a Windows NT command line, enter this command:

   ```
   java com.vitria.connectors.generator.ConnectorWizard
   ```

In a few seconds, the Connector Generator Tool displays; by default, the Required Info tab is in the forefront of the display and the fields will be blank. The screenshot below shows settings as displayed after opening the .gen file for the EBankSource connector (from the EBank Sample application). You can open an existing .gen file by selecting File-->Open and then browsing the local directory to find the file.

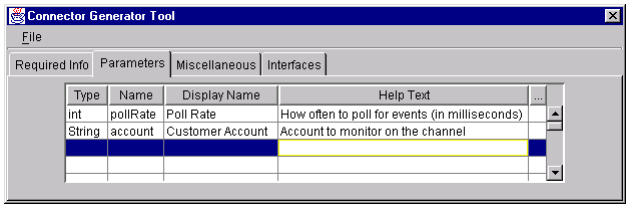


2. On the Required Info panel, enter a meaningful name for the Java class name; this name will comprise the first part of the name of each generated file.
3. The **Display name** will be used as the name of the flow in the RepBeanInfo file and shown in the Console. The **Java package** name is the name of the package to which all the generated files will belong. On this panel you should also select the radio button for the type of connector you are creating.
4. Click on the Parameters tab to display the Parameters tab. On this tab you can enter the properties for the Flow's property sheet. Each property for the property sheet is specified with a Java type and a name.



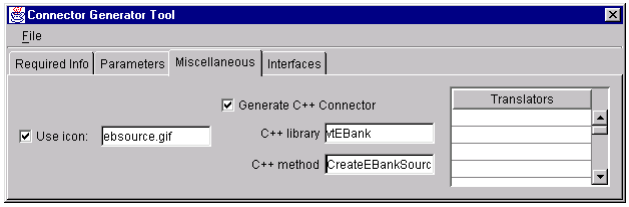
As you can see, this example uses the EMT Framework; rather than having a hard-coded Display Name and Help Text, methods created using the EMT framework are listed. If you want the property to have a default value (the far-right column of the display that's not fully displayed) you can enter it on this panel as well.

Whether hard-coded or referenced in this manner, the Display Name and Help Text will display in the property sheet through the Console to the end-user. As another example, here's another screenshot showing some textual parameters that are hard-coded and don't take advantage of the EMT Framework:



Note that the "Type" column on the Parameters tab refers to Java types, so you must be careful to enter only valid Java types. For example, the correct type for Java strings `String` —with a capital S—which is how you must enter the type name.

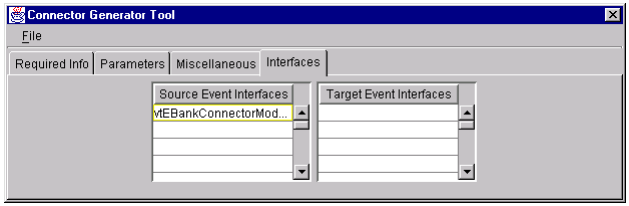
5. After entering all the parameters you want to be available to this Flow, click on the Miscellaneous tab. Here you can enter the name of the icon, if any, that you want your connector flow to use in the Flow palette on the Console.



6. If you will be implementing a C++ connector, select "Generate C++ Connector" and enter the names of the C++ library and createMethod that the connector flow will use (the names `yourLibrary` and `yourMethod` display in the class and method fields when you select Generate C++ Connector; replace these names with the C++ library and method you'll use to create the Flow.) The code generated by this tool will include the appropriate C++ wrapper calls.

   - To register SimpleTranslator classes inside the .ini (see "Generating an .ini File By Using Connector Generator" on page 102) you can enter the Translator class (or classes) on this tab.

7. Click on the Interfaces tab. Enter the type of event interfaces that will be accepted as input to the flow or sent as output from this flow; you can only specify event interfaces (not individual events). This example from the EBankJavaTarget connector shows the name of the Event interface `vtEBankConnectorModule.EBankConnectorEvents`:



8. Once you've completed all the tabs of the Wizard with the relevant information for the connector you're building, you can save the entries by selecting Save from the File menu. You'll be prompted to enter a filename; use .gen as the filename extension. (You can open this file later using the Wizard and change any settings; open it in a text file and do the same; or generate individual files at the command line.)

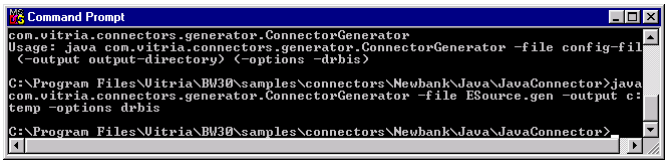
9. Finally, generate the files by selecting Generate from the File menu. You'll be prompted for a sub-directory into which to place the generated files.

Even if you don't have all the other components in place—events, message text, icons for the BeanInfo—you can still generate these Java elements and get a sense of how the process works and what the code looks like.

Note that one difference between writing your own code manually and using this tool is that the tool generates two class files rather than one for the Flow implementation: a Base class and the flow that extends the Base. Essentially, though, the code for the flow is the same.

Using the Connector Generator Tool and the .gen File

A .gen file is a text file formatted in a specific way that contains information used to generate the requisite interface, class, and other files for a specific connector flow. You can create a .gen file manually, using a text editor by following the guidelines below, or as a by-product when you use the Connector Generator Wizard tool, as described earlier in "How to Use the Connector Generator Tool (Wizard)." Once you have a .gen file, you use it as an input argument for the Connector Generator (command-line) tool, as shown here, to generate all files or select files:



Specifically, a .gen file includes:

- The connector name and name to display in the Console
- The connector type (Source, Target, or SourceTarget)
- The Java package to which the connector will belong
- The names of the .dll and the library function to execute in the constructor (if the connector contains a C++ flow)
- The property sheet names, types, and descriptions
- The filename of an icon to display in the Console, if any
- The list of translator classes that will be available in the processing tab in the Console for a source-target flow (if you implement transformer methods in the flow)
- The source and target interface list

Here are the first few lines of an example .gen file from the EBank sample:

```
classname EBankJavaSource
displayname
   TextMessage.formatMessage(ebanklocale.EBankSourceConnecto
   rMessages.JavaDisplayName)
kind source
package jebank
icon ebsource.gif
```

```
# sourceinterface nameofsource.eventinterface
sourceinterface vtEBankConnectorModule.EBankConnectorEvents
```

As you can see, this is a simple text file format that begins with an identifier name as the first item on a line followed by a value. The entries in the text above provide the classname for the flow (class EBankJavaSource); the name that will display in the console (picked up from the EMT Framework); the name for the Java package (package jebank); identifies that this will be a source connector flow (kind source); and identifies the icon that will be used (icon ebsource.gif). Also in the file is a reference to the type of events that the flow will handle (using the sourceinterface identifier).

A .gen file also contains listings for all the parameters defined in the EBankSource class; parameter descriptions in the .gen file are in the format:

```
   param type name "short description" "long description"
```

with the label “param” at the beginning of the line. For example, this flow will have a String parameter named “iorFile” that will get its short description label from the locale bundle’s IORShort variable and its long description from the IORLong variable.

```
param String iorFile
   TextMessage.formatMessage(ebanklocale.CommonEBankConnecto
   rMessages.IORShort)
   TextMessage.formatMessage(ebanklocale.CommonEBankConnecto
   rMessages.IORLong)
...
```

The reference to the locale bundle from which all the text derives is at the end of the .gen file, as follows:

```
   bundle vtEBankLocale3
```

Here’s another example .gen file, however, one that doesn’t implement the EMT Framework; the comments (#) describe what’s going on in this listing:

```
# parameter descriptions format:
# name of the connector class
classname javaSampleS
# name to be displayed in the gui
displayname "sampleS"
# kind of connector (source, target, or sourcetarget)
kind source
# java package this connector belongs to
package sampleS
# icon for display in the gui
icon ebsource.gif
```

```
# parameter descriptions format:
# param type name "short description" "long description"
param String firstName "First Name" "customer name"
# translator name.of.translator.class
translator SimpleTranslatorInterface
# sourceinterface nameofsource.eventinterface
sourceinterface SimpleTranslatorInterface
```

5 WRITING JAVA CONNECTORS

The flow's def, rep, and BeanInfo work together to provide access to the flow as a visual component (via the Console) so that end-users, such as business analysts, can set parameters and configure the specifics of the implementation at runtime. It's the flow itself that does all the work of a connector, however, and that's the subject of this chapter.

The code you write must make calls to the external API and process events, which means that you should have an understanding of the external API before starting and that you've given some thought to how you will specify the events in IDL or ODL; see "Defining Event Interfaces" for details.

If the flow you're writing will participate in transactions, you must also write the code for the transaction resource; see "Writing a Transaction Resource" on page 75 for details. Topics in this chapter include:

- Introduction to Java Connector Flows
- Writing Code for a Java Connector

Unless otherwise noted, all code listings in this chapter are excerpts from the *EBank Connector Sample*.

INTRODUCTION TO JAVA CONNECTOR FLOWS

All Java connector flows (source, target, and source-target) inherit some basic behavior (characteristics) from the BaseConnector class. Generally speaking, the common activities from a high level include starting the connector; interacting with the external system; and stopping the connector.

From a lower level, common functions include initializing the connector, retrieving configuration information from the repository, obtaining environmental information (from the Flow Env), and obtaining (and subsequently releasing) a connection to the external system. Source and target connectors diverge in how they process events (receive or send) and whether they can initiate a commit (source connector flows should initiate).

This chapter guides you through the process of writing the Java code for a connector to accomplish all these tasks, highlighting some of the differences between source and target connector flows that affect programming choices.

WRITING CODE FOR A JAVA CONNECTOR

If the external or third-party API for which you're creating the flow has been implemented in Java, you will implement the Flow in Java.

Your code should start with the package name; use the same name as the package for the Def, Rep, and BeanInfo. Here's the top portion of the code from the EBank Java source flow:

```
package jebank;

import com.vitria.fc.flow.*;
import com.vitria.fc.meta.*;
import com.vitria.fc.object.*;
import com.vitria.jct.JctLib;
import com.vitria.fc.io.*;
import com.vitria.fc.diag.*;
import com.vitria.connectors.common.BaseConnector;

import java.net.*;
import java.io.*;

import EBankAPI.*;
import vtEBankConnectorModule.*;
import ebanklocale.*;
import com.vitria.msg.ConnectorMessages;
...
```

In the sample above, the imports for EBankAPI.*, vtEBankConnectorModule.*, and ebanklocale.* are specific to the EBank connector, but all other imports at the top of the file are standard for any connector. Additionally, for a transactional flow, you would also import the ConnUtil class:

```
import com.vitria.connectors.common.ConnUtil;
```

After the imports, the code for any type of flow starts by extending the BaseConnector class from com.vitria.connectors.common.BaseConnector:

```
...
public class EBankJavaSource extends BaseConnector
   implements Runnable {
   private EBankConnection connection_;
...
```

As you can see, the code in the example above also implements Java's `Runnable` interface as a simple way of implementing its thread logic. Here's another example, from the target (rather than a source) connector; this one doesn't implement Java threads, as a point of comparison:

```
public class EBankJavaTarget extends BaseConnector
   implements EBankConnectorEvents {
...
```

In this example, the EBankJavaTarget flow class implements the event interface from the third-party system; you'll see more about this later in this chapter.

A final note about the code sample on page 46: The private EBankConnection object (variable) will be used to connect to the EBank server at runtime. Depending upon the external system, you may or may not need to setup a connection within which to interact with the external system. For example, database and message-queuing systems typically require a connection within which to handle transactions; on the other hand, file systems don't.

For this specific example, the connection object is used later in the code, in "Start and Stop Processing in the Flow" on page 48.

INITIALIZE THE CONNECTOR FLOW

Next, you must provide code to initialize the Flow; this will be called when the flow is created. As shown in the example, the init() method gets passed the Def, which contains the configurable parameters, and a Flow Env; the Flow Env is not used directly elsewhere in the code you write, but you must pass it in the initialization method so that the flow can obtain the environmental information it needs (pointer to the transaction resource, for example). The standard fully-qualified signature for the init() method is as follows:

```
init(com.vitria.connectors.common.ConnectorBaseDef, com.vitria.fc.flow.FlowEnv)
```

Here's the relevant code from *EBank Connector Sample*:

```
...
public void init(EBankJavaSourceDef d, FlowEnv env){
// STANDARD call super.init
   super.init(d, env);

   iorFile_ = d.getIorFile();
   account_ = d.getAccount();
   pollRate_ = d.getPollRate();
...
```

As shown in the listing above, within the init() method you must first call super.init(def, env) and next obtain the configuration parameters from the Def. (At runtime, this block serves to bring in either the default parameters or the parameters saved to the repository by an end-user.) With the basic flow initialized, the next task the code must handle is starting and stopping the flow from processing events.

START AND STOP PROCESSING IN THE FLOW

Once the flow is initialized, you must next provide methods that start and stop event processing in the flow; these methods will be invoked whenever an end-user starts, pauses, or stops a connection model from the BusinessWare Console, for example.

The first task is to override the void startEvents() method (inherited from the BaseConnector class); this gets called when the model is started:

```
...
public void startEvents(){
    if (logLevel_ >= DiagLogger.ERROR){
      ConnectorMessages.logStarting(logger_, EBankSourceConnectorMessages.baseid,
  name_);
    }

    try {
      connection_ = new EBankConnection(new File(iorFile_));
    }
..
```

The code above will attempt to connect to the EBank server. If the connection to the banking server is unsuccessful, a `DiagError` is thrown; the Flow Manager catches the `DiagError` and shuts down the connection model (not all the error-handling code is shown; see the *EBank Connector Sample* for complete code listings.)

Frequent re-connections (and disconnections) to an external system are inefficient, which is why the standard approach to connecting to the external system in any flow should be within the `startEvents` method (as shown in the code above) and disconnection should be within `stopEvents` (stopEvents is covered on page 49).

After connecting to the external system, you must next call super.startEvents():

```
super.startEvents();
   running_ = true;
   thread_ = new Thread(this);
   thread_.start();
```

```
if (logLevel_ >= DiagLogger.ERROR){
   ConnectorMessages.logStarted(logger_,EBankSourceConnectorMessages.baseid,
  name_);
    }
  }

...
```

The next block of code overrides `void stopEvents()`; this method will get called when the model is stopped. After checking for errors, the first thing you must do inside stopEvents method is to call `super.stopEvents`—you must do this before actually disconnecting. Here's the sample code:

```
...
public void stopEvents(){
   if (logLevel_ >= DiagLogger.ERROR){
   ConnectorMessages.logStopping(logger_, EBankSourceConnectorMessages.baseid,
  name_);
    }
// STANDARD call stopEvents before doing stop work
    super.stopEvents();

    if (thread_ != null){
      running_ = false;
      thread_.interrupt(); // in case we are sleeping
    }

    if (logLevel_ >= DiagLogger.ERROR){
      ConnectorMessages.logStopped(logger_, EBankSourceConnectorMessages.baseid,
  name_);
    }
  }
...
```

You must also stop the polling thread (for source connector flows that are polling a data source, as in this example). If the flow uses a connection, the code in stopEvents would include the necessary calls to disconnect from the third-party application (although in this specific example, there is no disconnect call; that's handled automatically when the process terminates.)

After taking care of initialization and providing mechanisms to start and stop the connector, the next major task is providing the code that actually does the work of processing the events that ultimately should move through the flow, as discussed in the next section.

CODE THAT SENDS AND RECEIVES EVENTS

You must implement code in your flow to send or receive events (or both) and move them to subsequent flows in the model. Source flows send events; target flows receive events; source-target flows can do both. For target (and source-target) flows, events are typically processed in the flow's own doPush() method, while source (and source-target flows) do this work typically in a thread and call super.doPush to pass events to subsequent flows.

The specific implementations vary depending on how the event information is available, specifically, whether event information is accessed from the metadata system or from marshaling stubs generated from IDL files, as discussed below.

See Chapter 7, "Defining Event Interfaces," and *BusinessWare Programming Guide* for additional information about events.

Event Handling (Target or Source-Target Flow Only)

Target and source-target flows receive events and typically process them in their own doPush() method. The method signature for a target flow's doPush() is:

```
public void doPush(EventBody evt){...
```

An `EventBody` is the structure (both Java and C++) that represents events in BusinessWare; an `EventBody` contains information, such as definition of its event interface, event name, and other specifications and parameters, needed to construct or deconstruct an event. You can construct or deconstruct the events at runtime in several way, including the approach shown in the sample code for *EBank Connector Sample*, discussed briefly in this section.

The standard approach for handling events in target connector flows (and source-targets) is to override doPush(). Within this method you'll put your event processing code specific to your application needs, such as adding records to a target system or updating database tables.

```
...
public void doPush(EventBody evt){
if (!evt.tryDispatch(this) && logLevel_ >= DiagLogger.WARNING){
     EBankTargetConnectorMessages.logUnknownEvent(logger_, evt.getEventSpec());
    }
  }
```

A target flow decomposes the EventBody that is passed to it in the doPush(). The EBankTarget connector flow is interested in (has subscribed to) two different types of event only—`withdraw`, `deposit`— it will drop any events of type `balance` (but will write a message to the log file).

In this example, the EBankTarget connector flow must take deposit events and withdrawal events and push them through a connection to the EBank server, which means they must take the incoming event and construct it as the type of data that EBank server can accept.

```
public void balance(int account, double balance){
    if (logLevel_ >= DiagLogger.WARNING){
      EBankTargetConnectorMessages.logBalanceNotSupported(logger_);
    }
  }
public void withdraw(int account, double amount){
...
   connection_.withdraw(account, amount);
    }
...
}

public void deposit(int account, double amount){
...
connection_.deposit(account, amount);
}
..
```

The code sample above uses `tryDispatch()` method; tryDispatch looks at the event, and if this flow implements the appropriate event interface, it will call the method that corresponds to the event interface. EventBody gets the information it needs about the event from its metadata, when it does the tryDispatch routine.

Vitria recommends using this approach when stubs are available. To use this technique, your code must also implement the interface in which the events are defined. In this example, the `EBankConnector.idl` was used (with the `jstubgen` tool; see *BusinessWare Programming Reference* for details) to generate several Java class and helper files (for marshalling and unmarshalling objects of the type specified in the IDL), which ultimately produced an interface that's implemented in this EBankJavaTarget, as shown here:

```
public class EBankJavaTarget extends BaseConnector implements EBankConnectorEvents
  {
```

Here's the complete text of `EBankConnector.idl`, which defines the events:

```
#ifndef _ebank_events_idl
#define _ebank_events_idl 1

// STANDARD Connector modules begin with vt
module vtEBankConnectorModule {
        // STANDARD interface names capitilize each word
    interface EBankConnectorEvents {
        // STANDARD event names lower case first word, upper case others
        event void balance(in long account, in double balance);
        event void withdraw(in long account, in double amount);
        event void deposit(in long account, in double amount);
     };
};
```

```
#endif
```

See Chapter 7, "Defining Event Interfaces" or the *BusinessWare Programming Guide* for additional information about IDL, ODL, and BusinessWare events.

Event Handling (Source or Source-Target Flow)

For a source flow, events are typically handled by constructing them from metadata and data (when new data is found in the external system during regular polling), and pushing them to the next flows by calling the super.doPush(com.vitria.fc.flow.EventBody) method. The event gets sent to the source connector's target list.

You must also wrap the doPush() calls inside a mutex (semaphore or lock that ensures mutually exclusive use); this transactional mutex ensures that none of the following occur at the same time:

- push
- commit
- abort
- start
- stop

This example steps through the source connector flow from *EBank Connector Sample* to highlight one approach to creating events. In the example, the source connector flow monitors EBank server for a change to the balance of a specified account and pushes the event when there's a change.

The listing only details the specific event-related code; the key tasks are creating the `EventBody` and then calling `doPush()` to send the event to the connector's target list. Unlike the EBankTarget connector flow, the source connector flow does not implement the event interfaces as compiled stubs. This uses an object called `EventDef` to hold the definition of the balance event, which it obtains with the `getMetaObject()` method below.

```
...
import com.vitria.jct.JctLib;
..
import com.vitria.connectors.common.BaseConnector;
...
..
  private EventDef balanceDef_;  // eventdef used to create our balance event
...

//Initialization code
/* As a minor optimization, we can cache the event def used to publish
       our balance events.
```

```
       */
    balanceDef_ = (EventDef)
  EBankConnectorEventsHelper.getMetaObject().findDef("balance");
   }
// startEvents(), stopEvents(), disconnect, other general routines not shown
...
```

This is the main polling loop (executed by the the thread). EBank Server is polled continuously for the balance; when the balance changes (it knows when it compares the `newBalance` to `lastBalance`), it calls sendNewBalance, passing the balance as a parameter.

```
public void run(){
    while (running_){
      double newBalance = lastBalance_;
// error and logging code
...

// get the balance
   newBalance = connection_.balance(account_);
      }
// error-handling code
       }
      }
// looks for a new balance and sends the event; does nothing if no balance change
if (newBalance != lastBalance_){
   sendNewBalance(newBalance);
      }
      else {
// error and logging code, checking the pollrate, threads, not shown here
...
```

Here's the code for the sendNewBalance() method that gets called; the key task is to create a new event for the balance EventBody and push it along. Within the sendNewBalance method, the event is constructed from the array of the parameters that comprise it. The EventDef provides information (metadata) about how these parameters should be used (by `createEventBody`) to construct the EventBody from the parameters.

```
...
/* Create a new event for the new balance, and send to the next flow */
  private void sendNewBalance(double newBalance){
    if (logLevel_ >= DiagLogger.VERBOSE){
      EBankSourceConnectorMessages.logNewBalance(logger_, newBalance);
    }
//new balance gets processed in here; create event parameters for the event here
//put account (int) first, followed by the balance (dbl)
Object [] parameters = new Object[2];
    parameters[0] = new Integer(account_);
    parameters[1] = new Double(newBalance);

    // create the event body and push it along
    EventBody out = JctLib.createEventBody(balanceDef_, parameters);
      if (logLevel_ >= DiagLogger.VERBOSE){
      EBankSourceConnectorMessages.logPushing(logger_);
    }
```

```
// STANDARD make all stateful changes before calling doPush or commit
    lastBalance_ = newBalance;
...
```

Once the EventBody contains the appropriate data, the super.doPush() can send it on its way. As mentioned earlier in this section, a true source connector flow must do this within the context of the mutex. Only true sources—not source-targets or targets—should acquire this mutex. Here's the code showing the mutex in the *EBank Connector Sample*:

```
...
Object transmutex = service_.waitForCommit();
synchronized (transmutex){
// STANDARD call super.doPush to push events to the next flow
super.doPush(out);
}
// commitAfter_ for source connector flows goes here
...
```

Source connector flows have one final task to perform—checking the value of the commitAfter_status flag, as discussed in the next section.

CHECKING VALUE OF COMMITAFTER_

Source connector flows must always check the value of the commitAfter_ boolean at the conclusion of the code, as shown in the listing. The commit must take place outside the transmutex lock (shown above):

```
...
if (commitAfter_){
      if (logLevel_ >= DiagLogger.VERBOSE){
        EBankSourceConnectorMessages.logCommitting(logger_);
      }

      service_.commit();
      }
   }
}
```

As mentioned elsewhere in this guide, source connector flows must perform this check because of the processing lifecycle within a connection model: after an event has been pushed through the entire connection model, the doPush (from the source connector flow that initated the process) is returned to the source flow. If the commitAfter_ boolean has been set to "true," (by a business analyst or other end-user), then the source connector flows calls `commit()` on the transaction service.

See "Java API Summary" on page 1 for a recap of methods used.

6 WRITING C++ CONNECTORS

Developers can write all the elements of the connector code in Java, not only the visual components that enable access through the Console—the Def, the Rep, and BeanInfo—but the Flow code itself. However, if the external system provides only C/C++ APIs, you'll need to write the flow in C++. You still create the def, rep, and BeanInfo classes in Java, but you will tie the flow to these entities using a wrapper class provided by Vitria, that allows C++ flows and Java flows to interact within the same connection model. This chapter tells you how; it provides information about implementing a connector flow in C++; topics include:

- Introduction to C++ Connector Flows and Flowbridge
- Implementing a Flow in C++

Unless otherwise noted, all code listings in this chapter are excerpts from the *EBank Connector Sample* that ships with BusinessWare.

INTRODUCTION TO C++ CONNECTOR FLOWS AND FLOWBRIDGE

Flowbridge is a complete Java package (com.vitria.connectors.flowbridge) and C++ library (vtFlowBridge) that provide several utility classes for working with C++ connector flows in the context of BusinessWare. Because much of the runtime implementation of BusinessWare, including the Console, is Java specific, when working with C++ flows, you'll need a way to let the flow objects you expose and control in the Console communicate with the flow itself. You'll also need a way to log messages and the like. FlowBridge provides mechanisms for communicating across the Java-C++ boundary.

One of the FlowBridge facilities is the C++ wrapper, a Java class that essentially functions as the representative for the C++ flow (in the context of a runing connection model). The wrapper (CPPFlow) extends the BaseConnector class, so it picks up standard flow properties, such as a list of flows to which it should deliver events.

Internally, the C++ wrapper implements a C++ wrapper flow def with the help of a C++ wrapper flow rep. The wrapper lets C++ flows work with the Java-based components of BusinessWare, as discussed in the next section.

How the C++ Wrapper Works

In a typical Java connection model, calls to doPush(), move synchronously through the connection model from flow to flow. Behind the scenes, a lower-level mechanism (not exposed to developers), does the work of pushing events from flow to flow; the push returns in the reverse order, back to the source flow.

When a C++ flow is included in a connection model, the same general process occurs, but a Java wrapper class (com.vitria.connectors.flowbridge.CPPFlow) intercepts the push on behalf of the C++ flow, and negotiates all processing on its behalf.

The CPPFlow acts as a Java representative for the C++ flow in the model. The CPPFlow intercepts any pushes to a C++ flow in the model; it processes the event, sending it the next flow in the model if appropriate.

Behind the scenes, when CPPFlow receives an event (in doPush()), it serializes the event into raw data which it then sends across the Java/C++ barrier using JNI (Java Native Interface) to the actual C++ flow.

When the data is received in C++, the event is deserialized (inflated) back into its regular state as an event, to be processed by the flow; push() gets called on the C++ flow with the event. If this push() call causes the C++ flow to push events of its own (if it is a source-target), these events are passed from C++ back to the Java wrapper using CORBA, where the events are temporarily stored in an event buffer.

After the JNI call has returned, the CPPFlow wrapper checks to see if any events are in the event buffer; if so, it pushes these events to the next flow or flows in the model.

Calls to the logger from within the C++ flow must also make use of this mechanism. These are all routed through a dispatcher that makes CORBA calls to cross the C++/Java barrier. The actual calls are made inside the Java framework by the Java wrapper code.

Implementing a Flow in C++

Creating a C++ flow and using the wrapper class involves the following steps:

- Create the Java components. In the Rep (EBankCPPSourceRep.java or EBankCPPTargetRep.java, for example), make a call to the C++ wrapper to create the flow, as discussed in "Call the Wrapper in the Java Code for the Rep."

- Write the C++ creation method using a standard CreateFlow method signature.
- Write the C++ code for the flow. As with a Java flow, the C++ flow must implement initialization, startup, shutdown, and other standard routines, as well as do the work of the flow itself; that is, either send or receive specified events.

These steps are discussed in the next several sub-sections.

CREATE THE JAVA COMPONENTS

Writing the Java components for a C++ flow is the same as writing a Java flow, with the exception of how the flow gets created by the Rep (as discussed in "Call the Wrapper in the Java Code for the Rep.") For example, here's the Java Def for the source connector:

```
package cppebank;

import com.vitria.connectors.common.*;

public interface EBankCPPSourceDef extends
   ConnectorSourceDef {
    public String getIorFile();
    public void setIorFile(String iorFile);
    public int getAccount();
    public void setAccount(int account);
    public int getPollRate();
    public void setPollRate(int pollRate);
}
...
```

See"Creating Java Components for a Connector Flow" in Chapter 4 on page 25 for additional information. One administrative detail, however: be sure to name all file names related to a C++ consistently, using the naming conventions for the Java file names. For example, the filenames associated with the C++ version of EBank source connector flow are:

- EBankCPPSourceDef.java
- EBankCPPSourceRep.java
- EBankCPPSourceBeanInfo.java
- EBankCPPSource.hxx
- EBankCPPSource.cxx

CALL THE WRAPPER IN THE JAVA CODE FOR THE REP

The Java Rep for the C++ flow must make the call to create the flow through the C++ wrapper. Here's an excerpt from the EBankCPPSourceRep.java (from the *EBank Connector Sample*) that shows how to bracket calls to create the flow source with the CPPFlow:

```
...
public FlowSource createFlowSource(FlowEnv env){
      CPPFlow cppFlow = new CPPFlow(this, env);
      cppFlow.setSharedLibName("vtEBank");
      cppFlow.setMethodName("CreateEBankSource");
      String [] params = new String[3];
      params[0] = "" + iorFile_;
      params[1] = "" + account_;
      params[2] = "" + pollRate_;
      cppFlow.setParams(params);
      cppFlow.init(env);
      return cppFlow;
   }
```

The parameters are put into a string array, which then gets passed to the create method.

WRITE A C++ FUNCTION THAT GETS CALLED BY THE WRAPPER

Write a create function using the standard signature (shown below); this create method is used to wrap the C++ flow inside the Java wrapper. Here's the standard signature:

```
#ifdef _WIN32
  int __declspec(dllexport) CreateFlowNameHere(const char** params,int iNumParams)
#else
  int CreateFlowNameHere(const char** params,int iNumParams)
#endif
```

The example C++ file listed below contains the methods used to instantiate both the source and target connectors for the EBank sample. The `CreateEBankSource` function shows standard method signature to use for methods that create C++ flows.

Note that the parameters are being passed as string values and converted to the appropriate types later in the code. You'll have to do this for the create function for your flow as well, because the parameters are always passed in as string values.

At runtime, both the EBankCPPSourceRep and EBankCPPTargetRep will create CPPFlow objects that will call the create methods using JNI (Java native interface).

```
/* This CreateFlow.cxx file contains the methods used to instantiate the source
  and target flows.  The Java Reps (generated by the connector wizard) create
    CPPFlow objects that call these create methods using JNI */

#include "IOP/vtiioporb.hxx"
#include "bw/flow.hxx"

#include "EBankSource.hxx"
#include "EBankTarget.hxx"

#ifdef __cplusplus
extern "C" {
#endif

/* This is the standard method signature to use for methods that create C++ flows.
  Parameters are always passed to us as string values, so you will have to convert
  as necessary. */
#ifdef _WIN32
  int __declspec(dllexport) CreateEBankSource(const char** params,int iNumParams)
#else
  int  CreateEBankSource(const char** params,int iNumParams)
#endif
  {
    const char * ior;
    int account;
    int pollRate;

    // convert the string parameters to their appropriate types
    ior = params[0];
    sscanf(params[1], "%d", &account);
    sscanf(params[2], "%d", &pollRate);

    // instantiate the source flow
    EBankSource * flow = new EBankSource(ior, account, pollRate);

    return (int)(vtFlow::Flow*)flow;

  }

#ifdef _WIN32
  int __declspec(dllexport) CreateEBankTarget(const char** params,int iNumParams)
#else
  int CreateEBankTarget(const char** params,int iNumParams)
#endif
  {
    EBankTarget * flow = new EBankTarget(params[0]);

    return (int)(vtFlow::Flow*)flow;
  }

#ifdef __cplusplus
}
#endif
```

WRITE THE CODE FOR THE C++ FLOW

The connector flow must provide routines to enable initialization, starting and stopping the flow, and event processing. A C++ flow is typically implemented as source file and a header file that contains several function signatures. (For *EBank Connector Sample* source connector flow, these are EBankSource.cxx and EBankSource.hxx, respectively.)

Within the C++ code for your flow, you must extend the `vtFlow::FlowImpl` class. Here's a sample code listing from EBankSource.hxx:

```
#ifndef EBankSource_h
#define EBankSource_h 1

#include "EBankConnection.hxx"

// STANDARD: Extend vtFlow::FlowImpl
class EBankSource : public vtFlow::FlowImpl {
private:
  char iorFileName_[1024]; // the name of the file containing the ior
  int account_; // the account we are polling
  int pollRate_; // the poll rate in milliseconds
  EBankConnection * connection_;
  int running_; // used for start/stop of our thread
  double lastBalance_; // what is the last balance we have received?
public:
  /* Constructor. Takes the ior filename, account, and poll rate. */
  EBankSource(const char *, int, int);
...
```

Next, you must override void startEvents(_Ix_Env& _env):

```
/* In our start code, we connect to the EBank Server, and start
     our polling thread. If there is  a problem, we set an exception.
     */
void EBankSource::startEvents(_Ix_Env& _env) {
  VTDiagnostic* diag;

  diag = ConnectorMessages::createStarting(EBankSourceConnectorMessages::baseid,
  "EBankSource");
  flowbridge_logmessage(VTTRACE_ERROR, this, diag);
```

Make any necessary calls to connect to the third-party application. Include code to prevent startup if there's an error (note the `_env.exception(err)` in the code below).

```
// try connecting
  if (!connection_->connect()){
    // could not connect!
    diag =
  CommonEBankConnectorMessages::createErrorConnecting(EBankSourceConnectorMessages
  ::baseid);
    flowbridge_logmessage(VTTRACE_ERROR, this, diag);
```

```
    // set exception so we don't start up
    VTDiagError * err = VTDiagError::create(__FILE__, __LINE__,
  CommonEBankConnectorMessages::createErrorConnecting(EBankSourceConnectorMessages
  ::baseid));
    _env.exception(err);

    return;
  }
```

After connecting (or creating a connection to) the external system, you must call FlowImpl::startEvents(_env); if you need to start a thread for polling (as in a source flow), you can do it in this function:

```
// STANDARD threads that push events must be started after FlowImpl::startEvents
  running_ = 1;

  /* This call creates a thread that will call the pollAccount method
     with this flow as an argument. */
  VTThreCreate(0, EBankSource::pollAccount, this);

  diag = ConnectorMessages::createStarted(EBankSourceConnectorMessages::baseid,
  "EBankSource");
  flowbridge_logmessage(VTTRACE_ERROR, this, diag);
}
...
```

Your flow should also provide the implementation for stopEvents(_Ix_Env& _env), as shown in this listing. Within this function, you should also stop any threads, disconnect from the external system, and handle any errors:

```
...
void EBankSource::stopEvents(_Ix_Env& _env){
  VTDiagnostic* diag;

diag = ConnectorMessages::createStopping(EBankSourceConnectorMessages::baseid,
  "EBankSource");
  flowbridge_logmessage(VTTRACE_ERROR, this, diag);
```

Before disconnecting, you must call FlowImpl::stopEvents(_env) before stopping processing:

```
// STANDARD call FlowImpl::stopEvents before stop work is done
  FlowImpl::stopEvents(_env);

  running_ = 0;

  diag = ConnectorMessages::createStopped(EBankSourceConnectorMessages::baseid,
  "EBankSource");
  flowbridge_logmessage(VTTRACE_ERROR, this, diag);
}
```

Calls to disconnect from the third-party application would be in this section of the code as well (in the example code shown above, disconnection is automatic, so there's no code for it).

Code that Sends and Receives Events

For a target (or source-target) flow, you should override push(vtFlow::EventBody * event, _Ix_Env& env). This gets called when your flow receives an event; the EventBody is passed into the flow through this entry point. As shown in the listing, event processing code also goes within this function.

To handle the events, your C++ flow should use the tryDispatch method (if stubs are present) which looks at the event and calls the appropriate method (if the event interfaces have been implemented); in this example, event interfaces are for balance, withdraw, or deposit.

If a balance event is being pushed, it's not processed, but an error message gets logged. The error handling is passed through calls to FlowBridge; see "C++ API Summary" on page 5 for function listing.

```
void EBankTarget::push(vtFlow::EventBody * event, _Ix_Env& env){
if (!event->tryDispatch(this)){
    VTDiagnostic * diag =
  EBankTargetConnectorMessages::createUnknownEvent(EBankTargetConnectorMessages::b
  aseid, event->getEventSpec());
    flowbridge_logmessage(VTTRACE_ERROR, this, diag);
  }
}
// This gets called by tryDispatch
void EBankTarget::balance(CORBA::Long account, CORBA::Double amount, _Ix_Env&
  env){
  VTDiagnostic * diag =
  EBankTargetConnectorMessages::createBalanceNotSupported(EBankTargetConnectorMess
  ages::baseid);
  flowbridge_logmessage(VTTRACE_WARNING, this, diag);
}
// This gets called by tryDispatch
void EBankTarget::deposit(CORBA::Long account, CORBA::Double amount, _Ix_Env&
  env){
  VTDiagnostic * diag =
  EBankTargetConnectorMessages::createDepositing(EBankTargetConnectorMessages::bas
  eid, account, amount);
  flowbridge_logmessage(VTTRACE_VERBOSE, this, diag);

  connection_->deposit(account, amount);
}
// This gets called by tryDispatch
void EBankTarget::withdraw(CORBA::Long account, CORBA::Double amount, _Ix_Env&
  env){
  VTDiagnostic * diag =
  EBankTargetConnectorMessages::createWithdrawing(EBankTargetConnectorMessages::ba
  seid, account, amount);
  flowbridge_logmessage(VTTRACE_VERBOSE, this, diag);
connection_->withdraw(account, amount);
}
```

(The parsing of event data and metadata to construct or deconstruct an EventBody is conceptually the same process as discussed in “Code that Sends and Receives Events” in Chapter 5 on page 50.)

If your flow is a Source or SourceTarget flow, you must make calls to FlowImpl::push(vtFlow::EventBody *, _Ix_Env& env) to push events on to the next flow (or flows). Here’s a sample code listing:

```
...
// get the balance
  double newBalance = connection_->balance(account_);

  // If we have a new balance, publish an event!
  if (newBalance != lastBalance_){
    lastBalance_ = newBalance;

    diag =
  EBankSourceConnectorMessages::createNewBalance(EBankSourceConnectorMessages::bas
  eid, newBalance);
    flowbridge_logmessage(VTTRACE_VERBOSE, this, diag);

    // construct the event body
    vtCORBA::Double corbaDouble = (vtCORBA::Double) newBalance;
    vtCORBA::Long corbaLong = (vtCORBA::Long) account_;
    OcsStubImpl::ParamValue params[3]; // leave 0 open
    params[1].addr = &corbaLong;
    params[2].addr = &corbaDouble;

    OcsInterfaceHelperElem * op =
  vtEBankConnectorModule_EBankConnectorEvents_getHelper()->elems()[0];
    vtFlow::EventBody * event = vtChanFlow::ChanFlowLib::createEventBody(op,
  params);

    // push it
    diag =
  EBankSourceConnectorMessages::createPushing(EBankSourceConnectorMessages::baseid
  );
    flowbridge_logmessage(VTTRACE_VERBOSE, this, diag);

    _Ix_Env env;
    FlowImpl::push(event, env);

  }
  else {
    diag =
  EBankSourceConnectorMessages::createNoNewBalance(EBankSourceConnectorMessages::b
  aseid);
    flowbridge_logmessage(VTTRACE_VERBOSE, this, diag);
  }
}
```

See “C++ API Summary” on page 5 for more information about FlowBridge. See *BusinessWare Programming Guide* and *BusinessWare Programming Reference* for additional information.

7 DEFINING EVENT INTERFACES

This chapter provides an introduction to BusinessWare events and how they are defined (using IDL) and stored in the repository as metadata. Topics in this chapter include:

- Overview of Events and Metadata
- Registering and Using Metadata
- Programming Techniques for Working with Events

OVERVIEW OF EVENTS AND METADATA

The common purpose of any BusinessWare connector flow is typically to either send or receive an “event”—a structure that contains both data and metadata that is used in the context of an application. For example, the *EBank Connector Sample* handles three different types of events: a withdraw event, a deposit event, and a balance event.

Metadata is data about data; it provides information about data itself, such as type of data, name, length, and so forth. In the context of BusinessWare and connector flows, metadata is data that provides information about an event, including its name, data type, and parameters.

BusinessWare uses the same hierarchy of metadata as is used by CORBA (Common Object Request Broker Architecture); the BusinessWare repository provides a hierarchical namespace for handling metadata. BusinessWare maintains a dictionary of all event metadata in the namespace; the system uses this dictionary at runtime to determine system behavior as events flow through the system. (The registry holds metadata about all other BusinessWare objects as well, in addition to events. See *BusinessWare Metadata Guide* for additional information.)

Developers must define the events that will be sent from or received by a custom connector. The structure of any event is exposed to other objects through an interface. Depending on the specific characteristics of the external system, developers can use IDL (interface definition language) to define event interfaces; ODL (object definition language) to define data or object interfaces; or both, as discussed in the next section.

OVERVIEW OF IDL AND ODL

In BusinessWare, IDL is used to define event interfaces, while ODL is typically used to define data interfaces.

Both IDL and ODL are vendor-neutral definition languages. IDL is a CORBA-compliant description language promulgated by the Object Management Group; ODL is a specification language (from the Object Data Management Group, formerly the Object Database Management Group). ODL is used to define object-oriented data, specifically, object types that conform to the ODMG Object Model.

Both IDL and ODL are *descriptive* languages; an IDL (or ODL) file defines an interface and fully specifies the parameters of each operation provided through the interface.

Vitria has extended these description languages by adding the keyword “event” to model the type of data specific to BusinessWare. (As an aside, you can use BusinessWare strictly as a CORBA ORB—bypassing the publish-subscribe and channel architecture—by not defining events.)

CREATING DEFINITION FILES THAT DESCRIBE METADATA

As mentioned previously, developers typically will use IDL to define event interfaces. To specify data interfaces that represent an object in an external system (or define a data interface for a persistent data store, such as an object-based database or collections), you can use ODL. In most cases, however, connector developers will be creating IDL files.

An IDL file is a text file formatted in a particular way that uses specific keywords to define a containment structure of objects that comprise metadata. An IDL file includes a definition of a module name that contains one or more interface definitions; each interface also contains one or more event definitions, as shown in this sample:

```
module vtFileConnectorEvents {
    typedef sequence <string> stringArray;
    typedef sequence <octet> octetArray;
```

```
    interface dataFileEventInterface {
        event void dataFileEvent(in octetArray data, in string fileName);
    };

    interface asciiFileEventInterface {
        event void asciiFileEvent(in stringArray data, in string fileName);
    };

    interface fileNotificationEventInterface {
        event void fileChanged(in string fileName);
        event void fileDeleted(in string fileName);
        event void fileCreated(in string fileName);
    };
...
```

For an external system that has many objects or events, you can nest modules within each other, as shown in the listing on page 68; each module defines a separate object.

These identifiers—module, interface, event—combine to provide a fully-qualified reference to an event in the hierarchical namespace (the repository).

If objects in the external system are modeled for create, read, update and deletes, you can use ODL to define data interfaces (rather than IDL). Like IDL, an ODL file can contain modules, interfaces, and events. Unlike IDL, ODL includes a keyword `persistent` that you can use as part of an interface definition. For example, the the data interface for `TABLE1` in the example below uses this keyword:

```
interface TABLE1 : persistent {
```

In addition, ODL automatically generates certain events (in IDL, events must always be specifically defined) when you ‘grind’ persistent interfaces (generate stubs from ODL by running the ODL file through the BusinessWare pre-compiler), specifically:

- add
- change
- delete
- addOrChange
- changeWithValue
- deleteWithValue

Here’s an excerpt of an ODL file:

```
module externalSystemConnectorEvents
{
```

```
    struct nameValueStruct {
        string name;
        any    value;
    };

    typedef sequence <nameValueStruct> nvSeq;

    interface externalSystemGenericEvents
    {
       event void ObjectChange(in string keyfield, in long objID, in nvSeq attrs);
       event void ObjectAdd(in string keyfield, in long objID, in nvSeq attrs);
       event void ObjectDelete(in string keyfield, in long objID);
    };

    module TABLE1_Module {

        /* TABLE1 structs go here */

        interface TABLE1 : persistent {
            attribute string field1;
            attribute double field2;
            attribute double field3;
            attribute string field4;
        };

    };

};
...
```

In this example, all the fields in `TABLE1` are primitive types (in this case, strings or doubles), but an external system may comprise many other types as well (in addition to primitives).

To represent defined types in IDL and ODL, you can use C/C++ style structs, as in the example above which shows the definition of `nameValueStruct` for storing name-value pairs; (a location for defining structs local to the module `TABLE1_Module` is noted in the comments of the ODL sample above).

Naming Conventions and Guidelines

Although IDL and ODL are both case-sensitive, don't use mixed upper-and-lower-case names as an attempt to give uniqueness to your event, interface, or module names: Many operating systems, including Windows NT Server, are not case sensitive at the lowest levels. To avoid potential naming conflicts, make each event name unique.

In addition, when defining IDL for events, connector developers should:

- Group related events into the same interface
- Reuse events and types whenever possible, rather than creating new ones. For example, use the standard events and types provided by Vitria found in basicEvents.idl.

Creating Stub Files

An IDL file provides the basis for creating Java or C++ stubs which can be used by a flow at runtime to derive the metadata needed to create an event. To create the stubs, developers can use the utility programs (`cstubgen` and `jstubgen`), which basically function as pre-compilers on the IDL to create various class files, helper files, and other programming-language-specific source code (at least four files are usually generated.)

As discussed in "Event Handling (Target or Source-Target Flow Only)" on page 50, you can implement the interface in your flow and use tryDispatch() deconstruct the EventBody before pushing it onward.

See *BusinessWare Programming Guide* and *BusinessWare Programming Reference* for more information about cstubgen and jstubgen.

Registering and Using Metadata

Custom metadata is avaialable to the system after it has been imported and registered with BusinessWare repository. To import and register metadata, you can:

- create an .ini that includes a call to `addTypeToRegister` method, passing in the name of the IDL file
- use the importidl command (if the IDL or ODL has been generated automatically, you must use this approach rather than using an .ini file).

See *BusinessWare Programming Reference* for information about using `importidl`. See "Creating the Initialization (.ini) File" on page 101 for additional information about creating and using .ini files.

Once the IDL (or ODL) metadata is loaded into BusinessWare it will become available on the Console.

PROVIDING AN IDL (OR ODL) GENERATOR WITH CUSTOM CONNECTORS

Defining events using IDL or ODL for external systems that have hundreds of objects is best done programmatically, by a generator tool that can query the external system's metadata and convert it to IDL or ODL automatically. As a connector developer, you might want to provide such a tool with your connector, in which case you should think about the topics covered in this section.

Creating a User Interface Tool for the IDL/ODL Generator

If an external system contains a large amount of metadata that must be imported, you should create a user interface (UI) tool to simplify the process of selecting subsets of metadata from among a potentially large number of objects.

For instance, an SAP system exposes over 200 BAPI objects and 400 IDOC types; importing every object into BusinessWare would be inefficient if you only need to use a few of them. In cases like this, you can create a wizard-like UI that enables business analysts to select the subset of metadata for which they want to generate events.

Be sure to provide some sensible options: making end-users choose just a few items from an extensive list is just as cumbersome as making end-users list hundreds of items. In short, connector developers should create a simple user interface tool to let end-users simply and easily choose components from the external system that should be input to the IDL/ODL generator.

Mapping External Datatypes to IDL (or ODL)

Data types in an external system do not always neatly map to the primitive types natively supported in IDL or ODL; for example, DateTime is a native RDBMS data type, but not a native IDL type.

Connector developers should provide a mapping from the external system's data type to an IDL or ODL data type (in the case of the example, creating an IDL/ODL struct to represent the DateTime format).

Extracting Metadata From the External System

If the external system exposes a native metadata API, you should use it whenever possible to programmatically extract metadata from the external system. If the system doesn't expose an API, you'll have to use another approach to getting the metadata. For example:

- If the external system is ORB based, it may expose a CORBA IFR (Interface Repository) object that you can use to query metadata.
- For an RDBMS based system, try extracting catalog information.
- For a mainframe based system, write a parser to extract metadata information from COBOL copy books.

Creating Metadata for BusinessWare

Developers can create metadata for BusinessWare in one of two ways: by using the BusinessWare metadata API, or by generating an IDL/ODL file. (End-users or developers can also use the Console to create metadata; see *BusinessWare Metadata Guide* for additional information.) Generating a file is easier to implement and debug than using the BusinessWare metadata API, so Vitria recommends that only advanced developers use BusinessWare metadata API.

Implementing the Generator

You can implement the generator as a command-line program or as a pluggable BusinessWare UI component. If the generator uses pluggable UI components such as event panels, the generator can be launched from within the BusinessWare Console. Otherwise, for stand-alone programs, the user must run the program from the command prompt or a Unix shell. See *BusinessWare Programming Reference* for additional information.

Programming Techniques for Working with Events

As mentioned throughout this guide, one of the major tasks a connector flow must perform is constructing events from (or deconstructing an event into) its constituent elements (as they exist outside of BusinessWare).

If stubs are available, you should use tryDispatch when receiving events (see "Creating Stub Files" on page 69 and "Event Handling (Target or Source-Target Flow Only)" on page 50); the *EBank Connector Sample* shows you how to use this recommended approach.

The *Simple Connector Sample* provides an overview of two approaches for sending an event by using methods from the JctLib (Java channel toolkit library) class (com.vitria.jct.JctLib) and from the meta object library (com.vitria.fc.meta.EventDef), as discussed briefly in the remainder of this chapter.

Here's the complete IDL for the SampleEvents interface (from the *Simple Connector Sample*). This IDL file defines an event called `dateEvent` that comprises three IDL types—`string`, `long`, and `long long`; the equivalent Java types are `String`, `int`, and `long`, respectively.

```
module myEvents {

interface SampleEvents {
    event void dateEvent(in string name, in long id, in long long date);
};

};
```

The listing below shows one approach to sending events of the type defined in myEvents.SampleEvents. An encoder is constructed by given an interface name and the connector flow ("this") ; it then returns an object that implements all the event methods in that interface, calling these methods (for example, dataEvent()) automatically pushes the event:

```
...
private SampleEvents sampleEvents_;
...
public void init(SampleSourceDef def, FlowEnv env) {
    super.init(def, env);
...
    sampleEvents_ = (SampleEvents)(JctLib.encoder(this, "myEvents.SampleEvents"));
...
}
...

public void doWork() {
    sampleEvents_.dateEvent(name_, id_, System.currentTimeMillis());
}
...
```

The listing above shows excerpts of the encoder being initialized in the init() method with all the events defined in `myEvents.SampleEvents`. This encoder is then called in doWork to create a dateEvent with the parameters name_ and id_; the current date is obtained using a method from the standard Java class library. Once the encoder creates the event, it automatically passes it to the target list of the connector.

To create the event directly, you can pass the data and the metadata (definition of the data) for the event to the `createEventBody()` method (in `com.vitria.jct.JctLib`), as shown in the listing below.

```
...
private void sendEvent() {
    Object[] params = new Object[3];
    params[0] = name_;
    params[1] = new Integer(id_);
    params[2] = new Long(System.currentTimeMillis());
EventDef eDef = (EventDef)
  (SampleEventsHelper.getMetaObject().findDef("dateEvent"));

    EventBody eb = JctLib.createEventBody(eDef, params);

    doPush(eb);
}
...
```

BusinessWare events are objects created from the `EventBody` class (`com.vitria.fc.flow.EventBody`). Metadata is used to create an EventBody; metadata can also be used to get an event's specification in the format of a Java String.

In the listing above, an EventBody `eb` is created by using the metadata stored in the `eDef` variable; the data itself is stored in the `params` variable (the array); the variable `eDef` is an `EventDef` object containing metadata.

The `findDef` method, called on a meta object, returns the metadata for an event. The meta object is obtained by calling the `getMetaObject` method in the `SampleEventsHelper.`(When you run this IDL file through the pre-compiler (jstubgen), the result is four files, including `SampleEventsHelper.java`, a helper class that is used for marshalling and unmarshalling objects of `myEvents.SampleEvents` type.)

To use the EventDef object, you must import the `EventDef` class as follows:

```
import com.vitria.fc.meta.EventDef;
```

Here's a summary of the steps involved with this approach:

- Create an object array that holds data to send in the event
- Create an EventDef to encapsulate the metadata about the object
- Use JctLib to wrap the data and metadata in an EventBody

Review the *Simple Connector Sample* and *EBank Connector Sample* for complete code listings and discussions of alternative ways of obtaining metadata and using it to create or deconstruct events. Also see *BusinessWare Programming Guide* and *BusinessWare Programming Reference*.

8 WRITING A TRANSACTION RESOURCE

This chapter tells you how to create a transaction resource for a connector flow. In the first part of the chapter, you'll find information about transactions and programming strategies; in the second part of the chapter, you'll find the details about how to write the actual code for a transaction resource. The chapter includes the following topics:

Unless otherwise noted, all code listings in this chapter are excerpts from the *EBank Connector Sample*.

OVERVIEW OF TRANSACTIONS

In simple terms, a *transaction* is a *logical unit of work;* it may encompass a single task or several tasks, but as a logical unit, all tasks—whether one task or dozens—must complete in their entirety, or none should complete at all.

For example, when a bank transfers funds from a savings account to a checking account, it must decrease the savings account *and* increase the checking account, or leave both accounts alone; otherwise, the banks books—and the customer's accounts—will be out of balance.

In a distributed computing environment, in which accounts (and resources in general) reside on completely different systems, ensuring that transactions always complete successfully is challenging for many reasons, but a solid approach to ensuring that they do is by using a *transaction service* that implements the *two-phase commit protocol*.

As discussed in "Supporting Infrastructure" on page 9, Vitria provides such a transaction service to manage transactions for connectors and connection models. If you want to ensure that a transaction within any flow is managed by this Transaction Service, you must create a transaction resource for the flow, as detailed in this chapter.

THE TRANSACTION SERVICE

Each connection model has an associated Transaction Service that handles transactions across all flows in the connection model; the Transaction Service is necessary because the individual flows that comprise a connection model are unaware of each other. (Flows are reusable components, which means a specific flow type may be configured and used differently in different connection models.)

When a connection model is started, each flow that participates in transactions creates an object called a *transaction resource* and passes it to the Transaction Service. Thus, the Transaction Service is aware of all the flows in a connection model that need to participate in transactions.

The Transaction Service for the connection model coordinates transactions across the model; however, the responsibility for ensuring that data is saved lies with each connector's transaction resource.

THE TRANSACTION RESOURCE

Every connector that participates in transactions must have a transaction resource that calls the appropriate methods to handle transactions in the external system at the direction of the transaction service.

The code you write for the transaction resource must include methods that the transaction service invokes at runtime to instruct the resource to get ready to commit (precommitResource or prepareToCommit, as shown in Table 8-2 and Table 8-3 on page 88), and to commit or abort the transaction, depending on the outcome of the precommitResource or the prepareToCommit.

The Transaction Service relies on the transaction resource of each flow in a connection model to implement the appropriate methods on the external system's API to commit data or abort processing. However, the transaction resource can only guarantee data to the extent that the API can guarantee it, and that depends on whether the external system provides API for one-phase or a two-phase commit protocol, as discussed in the next section.

ONE-PHASE AND TWO-PHASE COMMIT PROTOCOLS

As introduced in "Transaction Architecture" on page 11, the transaction service associated with each connection model can support both one- and two-phase transactions.

To recap:

- Two-phase transaction resources, which support a two-phase commit protocol, can guarantee that a commit will succeed because of a ‘pre-commit’ stage. If an external resource is two-phase, the transaction service relies on the transaction resource to guarantee successful commits once a transaction resource has successfully passed a “pre-commit” stage.
- One-phase transaction resources, which do not have a pre-commit method, cannot guarantee that a commit will succeed. At most, a single one-phase resource can participate in a transaction.

The two-phase approach is preferred because it can guarantee that a transaction will complete after a ‘pre-commit’ phase has been passed (discussed in more detail in the next section); it ensures *once-and-only-once* data delivery. If the external system supports two-phase commit protocol, you should create a transaction resource that implements the protocol.

If the external resource supports a one-phase protocol, there are techniques you can use to ensure once-and-only-once data delivery, or at-least-once data delivery, as discussed in “Robust One-Phase Transaction Resources” on page 80.

How the Transaction Service Processes Transactions

As mentioned earlier, when the Flow Manager instantiates the flows in a connection model at runtime, each flow that has a transaction resource associated with it adds the resource to the transaction service. The flows begin processing events through the model in whatever manner they’ve been configured to do so, and the transaction service does little until one of the flows signals a commit.

Given the nature of the underlying BusinessWare connection model architecture, the source flow should initiate the commit: processing in the context of a connection model traverses from flow to flow, and it’s only when the underlying push() returns to the source flow that processing (in the context of the entire model) is complete; see Chapter 2, “Architecture Overview.”

A standard source flow includes a check on the value of the `commitAfter_` boolean; if set to “True” (by the end-user through the Console or programmatically), the source flow calls commit on the transaction service. The commitAfter_ is a member of the BaseConnector class, from which all connector flows inherit basic characteristics; the default value is false.

(See “Checking value of commitAfter_” on page 54 for additional information about including this call in a source connector flow.) The sequence of steps involved with committing a transaction is as follows:

1. A source flow with `commitAfter_` property set to true calls `commit()` on the Transaction Service.
2. The Transaction Service calls `prepareToCommit()` on the one-phase transaction resource (if a one-phase resource exists in the model). This tells it to get ready for a commit and determines whether a commit is likely to succeed. If `prepareToCommit()` returns the active transaction ID, a commit will probably work, and the Transaction Service goes to the next step. If `prepareToCommit()` returns null, the Transaction Service gives up and the transaction is aborted.
3. The Transaction Service calls `precommitResource()` on all the two-phase transaction resources. If any of two-phase resource returns null (or doesn’t return anything) the Transaction Service gives up and the transaction is aborted.
4. The Transaction Service then stores all active transaction IDs in the repository.
5. The Transaction Service calls `commitResource()` on the one-phase transaction resource. If an error occurs, the Transaction Service gives up and the transaction is aborted.
6. The Transaction Service calls `commitResource()` on all the two-phase transaction resources and the commit is complete.

This process is summarized in the table below:

Table 8-1 Process Initiated when Source Flow with “commitAfter_” Set to “True” calls commit on Transaction Service

Transaction Service	Two-Phase Resource (*n*)	One-Phase Resource (1)
1. Calls prepareToCommit() on one-phase resource		2. Returns null or active transaction ID
3. Stops if receives null, or continues.		
4. Calls preCommit() on all two-phase resources	5(*n*) Each two-phase resource returns an active transaction ID or a null	
6. Halts processing if any null received (or no response); otherwise, continues processing		
7. Stores all active transaction IDs		

Table 8-1 Process Initiated when Source Flow with "commitAfter_" Set to "True" calls commit on Transaction Service (Continued)

8. Calls commitResource() on one-phase resource using its transaction ID		9. Commits (returns true to transaction service if successful)
10. If one-phase has committed successfully, then calls commitResource() on each two-phase resources using their respective transaction IDs	11(*n*) all commit	

Failures can occur at any stage of processing, so another major role of the transaction service is recovering from a system failure in the midst of a transaction and determining how to proceed, as discussed in the next section.

How the Transaction Service Handles Failures

When there's a system failure, the specifics of how the transaction service handles the recovery process will vary, depending upon whether the specific connection model includes only two-phase resources, or includes a one-phase resource.

- If only two-phase resources are involved, and if not all two-phase resources have pre-committed, the transaction service will let the transaction resources simply time-out; the distributed resources will be returned to a consistent state automatically.
- If a connection model includes a flow with a one-phase transaction resource (in addition to two-phase resources), and the system crashes after the prepareToCommit was called on the one-phase resource but before the precommitResource had been called on the two-phase resources, then the transaction service recovers by first calling getPrepareStatus (on the one-phase resource using the stored active transaction ID).
  - If the one-phase resource returns COMMITTED, then the transaction service can continue with the precommitResource on the two-phase resources.
  - If the getPrepareStatus returned PREPARED, then the transaction service will abort the transaction.

In this way, the one-phase transaction resource and the two-phase transaction resources can be involved in a transaction across the same connection model. However, since one-phase resources do not guarantee that commits will occur after the prepareToCommit method has been called, there techniques you should consider using in your one-phase transaction resources to ensure data delivery, as discussed in the next section.

ROBUST ONE-PHASE TRANSACTION RESOURCES

The two-phase commit protocol is the preferred way to handle transactions because it guarantees once-and-only-once data delivery. But not all external systems support the two-phase commit protocol. If you must write a one-phase resource, you can ensure once-and-only once data delivery by using some of the techniques discussed below (if the external API provides a transaction ID or you can add data in the context of a transaction). If the external system doesn't support a transaction ID or other marker, or let you add data in the context of an in-process transaction, you can still ensure at-least-once data delivery by using the technique described in "One-Phase Transaction Resources without Status Information" on page 83.

ONE-PHASE RESOURCES THAT IMPLEMENT STATUS INFORMATION

If the resource provides a transaction ID, you should use the transaction ID (or some other identifier as part of its commit protocol), you can use this transaction ID to ensure once-and-only-once data delivery. For example, the external API for the *EBank Connector Sample* supports one-phase transactions and provides an xid, as shown below:

```
...
public EBankServer() {
    accounts_ = new Hashtable();
    xids_ = new Hashtable();
    xidCounter_ = 0;
...
private synchronized int nextXID(){
    xidCounter_++;
return xidCounter_;
  }
...
// transactions
  public int prepare(){
```

```
    int xid = nextXID();
    xids_.put(new Integer(xid), commands_);
   commands_ = new ArrayList();
   return xid;
  }
   public boolean committed(int xid) {
      Object transaction = xids_.get(new Integer(xid));
   if (transaction == null){
      return true;
    }
    else {
      return false;
    }
  }

  public EBankServerModule.Result commit(int xid){
    // all or none - here we go
    List l = (List) xids_.get(new Integer(xid));
   if (l == null){
      return EBankServerModule.Result.UNKNOWN_TRANSACTION;
    }
...
```

As the listing above shows, the EBank server exposes `prepare()`, `commit()`, and `committed()` API, which include an `xid` parameter; this xid is used in the EBank transaction resource (as shown in the next code listing).

The transaction resource returns a byte array that corresponds to the xid (the code that converts the xid into and out of this array is not shown below). To signal a failure, the transaction resource returns null. It provides valuable information about the status of the current transaction, and will be used by the transaction service to perform the actual commit (or abort the transaction if there's a failure).

Here's an excerpt showing the transaction ID being passed as part of the prepareToCommit logic from the *EBank Connector Sample*.

```
public byte[] prepareToCommit(){
    if (logLevel_ >= DiagLogger.VERBOSE){
     ConnectorMessages.logPrepareToCommit(logger_,
  EBankTargetConnectorMessages.baseid);
    }
ByteArrayOutputStream baos;

    try {
// call prepare on the ebank connection and grab the xid
      EBankConnection conn = connector_.getConnection();
      int xid = conn.prepare();
...
```

```
catch(Exception iox) {
      if (logLevel_ >= DiagLogger.ERROR){
        EBankTargetConnectorMessages.logErrorPreparing(logger_, iox);
      }
        return null;
// convert byte array back to xid
...

...
```

When the transaction service calls `prepareToCommit` on the transaction resource, the transaction resource calls the external API (`prepare`) on the EBank connection. When the Transaction Service calls this transaction resource's `commitResource()` method, the transaction is committed. (See the code for the transaction resource (`ETargetResource.java` or `EBankTarget.cxx` and `EBankTarget.hxx`) from *EBank Connector Sample* for complete details.)

The *EBank Connector Sample* transaction resource also uses this transaction ID to determine if a COMMIT actually did succeed or not, as shown in the next excerpt. The getPrepareStatus method gets called (by the transaction service) for purposes of determining the status of the last transaction; the method passes the byte[] that was returned in prepareToCommit (some of the conversion and logging code is not shown).

```
public int getPrepareStatus(byte[] id){
    if (logLevel_ >= DiagLogger.VERBOSE){
      ConnectorMessages.logGetPrepareStatus(logger_,
   EBankTargetConnectorMessages.baseid);
    }
// convert the byte array back into an xid
...
if (logLevel_ >= DiagLogger.TRIVIA){
...
try {
      EBankConnection conn = connector_.getConnection();
// get the status of the xid
      if (conn.committed(xid)){
// we committed the last transaction successfully
        out =  OnePhaseResource.COMMITTED;
      }
      else {
// we did not commit the last transaction successfully
        out =  OnePhaseResource.PREPARED;
      }
    }
...
```

If the external API doesn't provide this level of support for transactions, you may still be able to ensure robust once-and-only-once data delivery if the external system lets you add data in the context of the current transaction, as discussed in the next section.

No Transaction ID API in External System

If the external API doesn't provide a transaction ID or other status flag in the context of a commit method, but the external system lets you add data within a transaction, you may still be able to ensure once-and-only-once data delivery for a one-phase resource. For example, Vitria's RDBMS source connector stores the database transaction ID in a `CONNECTORCOMMITS` table, as part of the transaction that's in-process.

If a system crash occurs during the commit, the Transaction Service calls the transaction resource's `getPrepareStatus()` method during recovery. The Transaction Service checks for the piece of data saved in the `prepareToCommit()`and if the data exists, the Transaction Service knows that the transaction committed—if it hadn't the data wouldn't exist, because all the work in a transaction succeeds or fails.

If the data doesn't exist, the transaction service knows the transaction failed, and the event is processed again.

ONE-PHASE TRANSACTION RESOURCES WITHOUT STATUS INFORMATION

If the external API doesn't provide a transaction ID and you cannot add data concurrent with an in-process transaction, you won't be able to determine if a commit succeeded or not in the event of a system crash during the commit stage. In such cases, you can code your one-phase resources to return default status values, as follows:

- Always return COMMITTED for a source connector flow
- Always return PREPARED for a target connector flow

These guidelines ensure that you'll provide *at-least-once* data delivery for flows that have one-phase resources (which are unable to use the techniques discussed in the previous section). The rationale for this approach is discussed in the next sub-section.

Transaction Service and getPrepareStatus()

Between the time the Transaction Service issues a commit() and the time all resources actually commit, there's always the possibility of a system failure. When the system restarts, the transaction service calls `getPrepareStatus()` method on the one-phase resource in the model for which it was processing a transaction. The `getPrepareStatus()` method returns only one of two values—COMMITTED or PREPARED—and the transaction service takes its next step based on this information.

- If the transaction service receives a `COMMITTED` from a resource, it assumes the data (event) was committed successfully, and it continues processing the current transaction; the event is not dropped.
- If the transaction service receives `PREPARED` from a transaction resource, it assumes that the commit did not succeed and it aborts the current transaction; drops the event; and attempts to re-process the transaction again.

In the case of a source connector flow, which picks up data from an external system and sends data in event form into the model, returning PREPARED if the commit completed can result in data loss if the commit actually succeeded because when a commit completes, the data in the external system is deleted and the event containing the data is dropped. So when the transaction service receives a PREPARED and it aborts the current transaction and attempts to process the transaction again, the data is no longer available on the external system. That's why you must return COMMITTED for a source flow; if the commit really failed, the data still exists and will simply be re-processed.

The opposite is true of a target connector flow, which receives events either directly (or indirectly, through other flows in a model) from a BusinessWare channel and ultimately inserts them into an external system. In the case of a target connector flow, you should always return PREPARED. When you return COMMITTED for a target flow, the transaction service assumes the event has traversed the connection model succesfully, so it notifies the channel server, which then delivers the next event in the list.

If the commit had in fact failed, however, and the data wasn't saved to the external system, then the event is gone. That's why you must return PREPARED for a target flow; if the commit had succeeded, the data is saved in the external system and event will be re-processed, possibly resulting in a duplication of data—but again, data loss is anathema to a transaction processing system: at-least-once delivery is preferred over data loss.

WRITING A TRANSACTION RESOURCE FOR A FLOW

To create a transaction resource for a connector flow, you must first analyze the external system's API and determine if it supports transactions; if it does, you must next determine if the transactions are one-phase or two-phase. With an understanding of the external transaction API, you can then write a Java class for the transaction resource (or a C++ class, if your connector flow is C++ based) that implements the transaction logic, using one of the approaches discussed earlier in this chapter.

Once you have the transaction resource written, you'll associate the resource with the flow by making the appropriate call in your Flow source's initialization section (see Chapter 5, "Writing Java Connectors" or Chapter 6, "Writing C++ Connectors" for additional details about writing code for the Flow.) These tasks are detailed in the next two sections.

WRITE THE TRANSACTION RESOURCE

The transaction resource is a runtime object that gets instantiated by the flow and then passed to the BusinessWare transaction service.

Code for the transaction resource includes calls that implement the external system's transaction API, that is, calls to commit or abort transactions. The connection to the external system is made in the flow itself (in the startEvents() method), and this connection is passed to the transaction resource in the constructor, as you'll see in the code below.

Import Transaction APIs

Here's a step-through of the Java version of the transaction resource for the *EBank Connector Sample*; this transaction resource is added to the transaction services from the Java target flow. The code starts with the package name (your transaction resource should be part of the same package as the def, rep, BeanInfo, and the flow itself). The code begins with these standard imports:

```
import com.vitria.fc.trans.*;
import com.vitria.fc.io.*;
import com.vitria.fc.data.*;
import com.vitria.fc.diag.*;
import java.io.*;
```

Next, the API of the external system is imported; in addition, this transaction resource takes full advantage of the EMT Framework, so those classes are imported as well:

```
import EBankAPI.*;
import ebanklocale.*;
```

```
import com.vitria.msg.ConnectorMessages;
```

The external API supports only a one-phase commit, so the class implements this interface, as shown in the code below. Also note the calls to the logger in this listing. (See “Methods Available to Transaction Resources” on page 87 for discussion and comparison of the one-phase resource methods and the two-phase methods.)

```
public class ETargetResource implements OnePhaseResource{
  private DiagLogger logger_; // for logging
  private int logLevel_; // for logging
```

Connect to the External System

Some external systems, such as databases, implement transactions within the context of a connection, in which case the transaction resource must share the connector’s current connection to the external system; this is the case in the code below (the `connector_` variable is used to access the connection to the EBank server).

```
...
private EBankJavaTarget connector_;
...
  public ETargetResource(DiagLogger l, int logLevel,
   public byte[] prepareToCommit(){
    if (logLevel_ >= DiagLogger.VERBOSE){
     ConnectorMessages.logPrepareToCommit(logger_,
   EBankTargetConnectorMessages.baseid);
    }

    ByteArrayOutputStream baos;

    try {
// call prepare on the ebank connection and grab the xid
      EBankConnection conn = connector_.getConnection();
      int xid = conn.prepare();

      if (logLevel_ >= DiagLogger.TRIVIA){
       EBankTargetConnectorMessages.logXID(logger_, xid);
      }
...
```

See *EBank Connector Sample* to review the full code listing for the transaction resource. Again, the majority of the code in any transaction resource will be specific to the external transactional API and how commits and aborts are handled. See “Robust One-Phase Transaction Resources” on page 80 for information about ensuring that one-phase resources provide once-and-only-once data delivery, or, if need be, at-least-once data delivery.

INITIALIZING THE TRANSACTION RESOURCE IN THE FLOW

To add a transaction resource for your connector to the Transaction Service, you must make a call in the `init()` method in your flow code, as shown in this excerpt from EBankJavaTarget.java, the *EBank Connector Sample*:

```
. . .
public void init(EBankJavaTargetDef d, FlowEnv env){
     super.init(d, env);
     iorFile_ = d.getIorFile();

ETargetResource res = new ETargetResource(logger_,
   logLevel_, this);
    ConnUtil.addResource(res, service_);
  }
. . .
```

See Chapter 5, “Writing Java Connectors” (or Chapter 6, “Writing C++ Connectors”) for more information about referencing the transaction resource in the code for your flow.

METHODS AVAILABLE TO TRANSACTION RESOURCES

Methods to use for a Java transaction resource (from the `com.vitria.fc.trans` package) are shown in the two tables below.

Table 8-2 One-phase Methods

Method	Description
prepareToCommit()	Returns transaction ID
commitResource()	Commits the transaction
abortResource()	Kills the transaction
getPrepareStatus()	Returns a COMMITTED flag if the transaction has committed and returns a PREPARED flag if the transaction has not yet committed

Note that although the one-phase `prepareToCommit()` method and the two-phase `precommitResource()` method return the current transaction ID to the transaction service, these methods do not provide an equal measure of assurance that the transaction will complete after these respective calls are made. By definition, a two-phase commit API provides a guarantee that after a successful precommitResource, the commit calls will succeed, even in spite of a system failure. If the external system provides a two-phase commit protocol, then it provides this guarantee.

Table 8-3 Two-phase Methods

Method	Description
precommitResource()	Returns transaction ID and prepares the system for a commit
commitResource()	Confirms the transaction
abortResource()	Kills the transaction

9 HANDLING ERRORS

Your connector flow must include code to handle erroneous data and unexpected situations. This chapter provides information about how to log and handle errors; topics include:

- Sending Error Messages to the Log File
- Log Levels

SENDING ERROR MESSAGES TO THE LOG FILE

All connector flows inherit from the BaseConnector class, which provides utility methods for logging and exception handling; BaseConnector class implements the DiagLogger interface. DiagLogger contains a set of six log levels, from NONE to TRIVIA, as discussed in the next section.

LOG LEVELS

The DiagLogger interface (imported in the BaseConnector) provides the following constants to decide how much information gets written to the log. The log level is a given property; end-users can set the log level for a flow on its property sheet (in the Console):

Table 9-1 Log Levels

Log Level	Description
NONE	(0) Perform no tracing at all under any circumstances.
ERROR	(1) Log serious errors and exceptions.
WARNING	(2) Log warnings and other questionable situations.
NORMAL	(3) Log major occurrences, such as startup and administrative changes (not event-based). Adds little or no performance overhead and should produce little output if the application is well designed.
VERBOSE	(4) Log every system occurrence, typically pre-transition or per-event. Adds some performance overhead.
TRIVIA	(5) Log all available information. Can negatively affect performance (by adding to overhead). Restrict use to debugging.

In the code you write, you can check the log level before writing out a message to the log or elsewhere. For example, the following statement ensures that only actual errors (in the application) or Java exceptions trigger whatever follows, specifically, the message writing.

```
if (logLevel >= DiagLogger.ERROR) ...
```

On the other hand, a logLevel >= DiagLogger.TRIVIA would capture an extensive amount of information, potentially causing a performance bottleneck, (depending on the context of the rest of code in which it's embedded).

The actual text of the messages that get generated by these log levels depends on what you provide; see "Using the EMT Framework for Log Messages" on page 92). The logLevel constants provide an arbitrary range that you must provide for in your own connector by creating the appropriate message text.

LOGGER_

The BaseConnector also provides a `logger_` object; use this object to send error and informational messages to the log file. Your code should include a check for the log level setting (of the connector at runtime, since end-users of your connector can alter this in the Console), as shown in the excerpt below:

```
...
public void startEvents(){
if (logLevel_ >= DiagLogger.ERROR){
   ConnectorMessages.logStarting(logger_, EBankSourceConnectorMessages.baseid,
  name_);
     }
...
```

Within the `if` statement in the listing above, a message will be sent to the logger when the connector flow is started and attempts to connect to the external system (the EBank server); the log will identify where the error occurred (logStarting).

Your code must import the DiagLogger interface as follows:

```
import com.vitria.fc.diag.DiagLogger;
```

The next code listing includes the entire startEvents method, with additional error checking and logging code:

```
...
if (logLevel_ >= DiagLogger.ERROR){
      ConnectorMessages.logStarting(logger_, EBankSourceConnectorMessages.baseid,
  name_);
     }

     try {
```

```
      connection_ = new EBankConnection(new File(iorFile_));
    }
    // STANDARD throw DiagError in startevents when encountering a failure
    catch (Exception e){
      Diagnostic d =
  CommonEBankConnectorMessages.createErrorConnecting(EBankSourceConnectorMessages.
  baseid);
      logger_.write(d);

      // wrap the diagnostic in a DiagError and throw it
      throw new DiagError(d);
    }
...
```

Here's the comparable code from the C++ version of the source connector flow. In this listing, you see that the FlowBridge API (the C++ wrapper that enables a C++ flow to cross the C++/Java boundary) is used to write out the log messages; for example, `flowbridge_logmessage(VTTRACE_ERROR, this, diag)` and `flowbridge_logmessage(VTTRACE_ERROR, this, diag)`

```
...
void EBankSource::startEvents(_Ix_Env& _env) {
  VTDiagnostic* diag;
diag = ConnectorMessages::createStarting(EBankSourceConnectorMessages::baseid,
  "EBankSource");
  flowbridge_logmessage(VTTRACE_ERROR, this, diag);

// try connecting
  if (!connection_->connect()){
    // could not connect!
    diag =
  CommonEBankConnectorMessages::createErrorConnecting(EBankSourceConnectorMessages
  ::baseid);
    flowbridge_logmessage(VTTRACE_ERROR, this, diag);

// set exception so we don't start up
    VTDiagError * err = VTDiagError::create(__FILE__, __LINE__,
  CommonEBankConnectorMessages::createErrorConnecting(EBankSourceConnectorMessages
  ::baseid));
    _env.exception(err);

    return;
  }
...
```

This is just a short example. Review the code listings for both C++ and Java connector flows in *EBank Connector Sample* for more details about how to use these call in your own code.

See "C++ API Summary" on page 5 for all FlowBridge methods.

USING THE EMT FRAMEWORK FOR LOG MESSAGES

The EMT (Error, Messaging, and Tracing) framework is provided in BusinessWare to handled messages of all types, including error, trace, and log messages ("The EMT (Error/Messaging/Tracing) Framework" on page 8 for additional background information).

The EMT Framework enables fast and easy localization and internationalization because it provides a complete infrastructure for generating text messages; it also provides the underlying facilities for handling all these messages in your code, by means of a "resource bundle," which includes C++ stubs (that provide the link between between your text and BusinessWare), Java, C++, or both

The resource bundle is specific to any BusinessWare application, including connectors. Developers writing custom connectors must create a resource file, generate the stubs and database of message text (which will be used at runtime), and generate C++ header files or Java class files (or both, if you're creating a connector flow that will have both types of flows). In the source code for the flow Rep you then load the resource bundle as explained in "Load the Locale Bundle for the EMT Framework" on page 34.

During the development process, you'll likely want to simply use placeholder type messages, and develop the actual resource bundle when all the other technical issues are complete. For final code, you should use the EMT Framework. See the *BusinessWare Programming Guide* for more information.

10 MISCELLANEOUS DEVELOPMENT ISSUES

This chapter discusses:

- Customizing Methods
- Multi-threaded Subscriber
- Request-reply (Synchronous) Processing

CUSTOMIZING METHODS

In most connection models, events sent by the flow source or received from a channel are not in the appropriate format. For example, ISAM (indexed sequential access method) file data isn't structured as events, and must be transformed into events usable by BusinessWare. Transformer flows modify events into the format needed. BusinessWare provides several standard transformer methods; see *BusinessWare Connection Modeling Guide* for information.

Developers can also write custom transformer methods for several of the built-in source-target flows, specifically, the Simple Data Transformer, Simple Router, Nested Router, and Multi-threaded Subscriber, as discussed in this section.

Before writing a custom transformer method, you must define:

- Event interfaces that the transformer method will handle
- Action to be taken for both expected and unexpected events
- Data to be returned after event processing.

Your code can return a single event; an array of events; or null (in which case the event is dropped).

WRITING THE METHOD IN THE JAVA CODE EDITOR

You can write the code by using the Java Code Editor, which is a feature of the BusinessWare Console. The advantage of using this editor is that issues such as class names, classpaths, saving, compiling, and registering in the namespace are taken care of for you—you write the method body only in the Java Code Editor dialog box.

The method takes an EventBody as a parameter. If the method is part of a Simple Data Transformer, it returns another EventBody, otherwise it does not.

Clicking the Register button in the Java Code Editor compiles the class containing the method and registers the class in BusinessWare. The new class will be displayed in the Simple Data Transformer's Processing tab (via the Console).

See the *BusinessWare Connection Modeling Guide* for details about using the Java Code Editor. Review the next section to gain an understanding of how to write the custom method.

WRITING THE CODE MANUALLY

To write the code externally—outside the BusinessWare Console and Java Code Editor—you must write a traditional Java class file, with package name, imports, and the like.

After you have written and tested your code, you can register it with BusinessWare, as discussed in "Registering Custom Methods", below. (As with other Java class files for flows, you must put the transformer class in the system's CLASSPATH for Java classes.)

The next several sub-sections step through the process of writing a custom method for the Simple Data Transformer.

CUSTOMIZING METHODS FOR THE SIMPLE TRANSFORMER

For the Simple Transformer, the class must have these imports:

```
com.vitria.fc.data.*;
com.vitria.fc.io.*;
com.vitria.fc.flow.*;
com.vitria.fc.diag.*;
com.vitria.msg.*;
com.vitria.fc.meta.*;
com.vitria.fc.record.*;
com.vitria.jct.JctLib;
com.vitria.connectors.datalators.simpletranslator.*;
```

The four method signatures you can use for a custom transformer method are as follows:

- `myMethod(EventBody e)`
- `myMethod(EventBody e, String [] s)`

- `myMethod(EventBody e, String [] s, SimpleTranslatorInterface sti)`
- `myMethod(EventBody e, SimpleTranslatorInterface sti)`

Each of these can return either an EventBody or an EventBody array. If the returned EventBody reference is null, the event is dropped.

Writing the Custom Class

You can create custom transformer classes that contain any number of methods that perform different data transformations. Placing multiple methods in one class groups them all in one file and separates the transformation logic from the connector flow code.

To facilitate re-use, group multiple translator methods in their own class.

Here's a walk-through of a transformer class (from the *Simple Connector Sample*) that has two methods; one converts `dateEvents` to `stringEvents`, the other converts `stringEvents` to `dateEvents`.

The following IDL code defines a `dateEvent`:

```
module myEvents {
interface SampleEvents {
    event void dateEvent(in string name, in long id, in longlong date);
};
};
```

The following IDL code defines a `stringEvent`:

```
module vtEvents {
interface stringEventInterface {
    event void stringEvent(in string data);
};
};
```

The following code is in the `MyDataTranslator` class described in the *Simple Connector Sample*:

```
public EventBody dateToString(EventBody e) {
    if (e.getEventSpec().equals(dateSpec_)) {
        Object[] p = e.getParameters();
        return JctLib.createEventBody(
            stringEvent_, new Object[]{ p[0] + “,” + p[1] + “,” + p[2] }
        );
    }
    return null;
}

public EventBody stringToDate(EventBody e) {
```

```
    if (e.getEventSpec().equals(stringSpec_)) {
        Object[] p = e.getParameters();
        String inString = (String)(p[0]);
        StringTokenizer st = new StringTokenizer(inString, ",");
        if (st.countTokens() == 3) {
            String s1 = st.nextToken();
            String s2 = st.nextToken();
            String s3 = st.nextToken();
            return JctLib.createEventBody(
                dateEvent_,
                new Object[]{ s1, new Integer(s2), new Long(s3) }
            );
        }
    }
    return null;
}
```

Registering Custom Methods

Transformer classes and methods are displayed in the BusinessWare Console when the Simple Data Transformer flow is configured. In order for the custom transformer class to be displayed along with its list of available methods, the transformer class must be registered.

Custom methods created using the Java Code Editor are automatically registered during the compile stage; see the *BusinessWare Connection Modeling Guide* for details.

Custom methods created manually, using the JDK or and IDE, must be registered in BusinessWare in one of three ways:

- From an .ini file.
  - If you create the .ini by using the Connector Generator Tool, simply supply the transformer class name in the .gen file (or in the Connector Generator Wizard GUI. See "Using the Connector Generator Tool and the .gen File" on page 42 for details.
  - If you write a Java program to generate the .ini, call `addTranslatorLib()`with the name of the transformer class and method settings. See "Adding a Transformer Class" on page 104 for details.
- From a command line, using the TransReg utility. Briefly, you simply enter `transreg -a JavaClassName1...` at the command prompt. The batch file (or .csh, in Unix) launches the Java SimpleTransReg program and registers the transformer. (See *BusinessWare Programming Reference* for details about using this command-line utility)

- From the BusinesWare Console, by entering in the class name on the Processing Panel. See the *BusinessWare Connection Modeling Guide* for details.

In addition to registering the transformer class in BusinessWare, you must also make sure the class files are located in the CLASSPATH of the machine

To use an externally-developed custom transformer method that is specified in a call to `addTranslatorLib()` in a Java program, as seen in the previous section, perform the following steps:

1. Compile the Java file containing the transformer class and method.
2. Place the resulting class files in the CLASSPATH.

Displaying Event Lists

A transformer flow is a source-target flow, and as with the source flows and target flows, the event types that a transformer can send and receive can be displayed in the Console.

types of events that a transformer flow can send and receive can be displayed in the Console. The list of events that the transformer can send is the *source list*; the events it can receive are the *target list*.

See “Creating the Java components for the Flow” in Chapter 4 for details about implementing getSourceEventInterfaceList (to display a source list) and getTargetEventInterfaceList() to display a target list.

Customizing Simple Router Methods

The SimpleRouter is a source-target flow that enables routing of events based on type or actual content. Simple Router has a list of potential targets; when an event is received by the router, a user-defined method receives the EventBody aldong with a vector of possible targets and user-configurable parameters.

The router returns a vector with a subset of targets to which it can route; the SimpleRouter can route based on type (of event) or based on content. Developers can use the Java Code Editor to modify routing methods; see *BusinessWare Connection Modeling Guide* for additional details about using Simple Router and about using the Java Code Editor.

CUSTOMIZING NESTEDROUTER

The Nested Router encompasses features of both a simple router and a nested model; that is, it receives events that it can route based on type or content, but it routes to other models (rather than specific flows or channels). If, in the context of a connection model, you need to direct output to multiple models, you can use the Nested Router flow.

Use Nested Router when you need to route to numerous multiple flows or channels. Nested Router has a "folder" property; when you place a nested router in a model, you specify the folder in which Nested Router will look for models.

For example, you could create a folder called /users/accounting and specify this as the folder property for a specific Nested Router. You can then use the Java Code Editor (in the Console) to write your own code to define the target model (for the router); the method takes in an EventBody and returns the name of a string. For example:

```
String out = "VITR";
```

would route all events to model called VITR in folder /users/accounting.

Usage rules for Nested Router are the same as for the Nested Model; see *BusinessWare Connection Modeling Guide* for complete details.

MULTI-THREADED SUBSCRIBER

The Channel Flow source is a single instance source: it subscribes to a channel and receives events, processing them one at a time, sending each event to its target.

The Multi-threaded Subscriber establishes a series of queues that are used by multple instances of a connection model to read events from the channel. The size of the queue is set by the users as is the number of instances.

The Multi-threaded Subscriber is used in multi-instance target connection models. The Multi-threaded Subscriber uses a (key) map class to determine which queue is read. The default class is round robin as follows

```
// Always return a java.lang.Integer
Object iKey = new
   Integer((int)e.getPosition()%concurrencyLevel());
```

You can override this method to implement your own processing order, using the Java Code Editor. (See *BusinessWare Connection Modeling Guide* for additional information about configuring multi-instancing.)

Multi-instance Connection Models

The typical connection model runs as a single instance. To improve throughput, end-users can configure connection models to run multiple instances, provided all flows in the model are multi-instance-enabled.

Several BusinessWare flows, such as the Multi-threaded Subscriber, Channel Target, Simple Data Transformer, RDBMS, and HTTP flows, are multi-instance enabled; end-users can combine these flows as needed into specific connection models and configure the number of instances that they'd like (by changing the concurrent instances property under Model Properties).

Not all flows can easily be multi-instanced. In addition, all flows in a connection model must be multi-instance enabled (otherwise, the createFlowManager won't return true and the connection model won't be multi-instanced after all). If you're wiring together multi-instance-enabled flows into a connection model or connection model template, be sure that all flows are multi-instanced.

REQUEST-REPLY (SYNCHRONOUS) PROCESSING

As mentioned throughout this guide, once an event exists in the context of a specific connection model, it is "pushed" from flow to flow by an underlying BusinessWare mechanism. The originating flow waits for the push to be returned before it concludes its processing loop.

When a connector flow picks up data from a channel and creates an event that gets pushed from the source flow, the communication style between channel and source flow is asynchronous.

As an alternative, you can implement a request/reply (synchronous) using the request/reply API. (BusinessWare HTTP connector has implemented this style.) See the Request/Reply Sample in the connectors samples folder in BusinessWare for additional information.

Impact on Using SimpleTranslatorInterface in a Connection Model

For example, typically when a simple translator returns an EventBody, the EventBody is pushed (usually to the next flow for processing). If you want to return the EventBody as a result (instead of pushing it to the next flow), you can use the markIsResponse(true) mthod, and the event will be returned to the requestor (instead of being pushed).

Use public EventBody[] execute(EventBody e) method to use request-reply style invocation on the flow that follows the Simple Translator. That flow will process the event and return an array of event bodies. This method enables you to write code in the SimpleTranslator that takes an event and performs multiple invocations on the subsequent flow before returning a result. To specifically name the flow on which multiple invocations should occur, use public EventBody[] executeAt(String flowName, EventBody e), where flowName is a valid flow in the connection model.

See "Simple Translator Interface" on page 5 for listing of all methods in this interface.

11 INSTALLING CONNECTORS AND CONNECTION MODELS

This chapter discusses several tasks associated with installing connectors and connection model templates. This chapter includes these topics:

CREATING THE INITIALIZATION (.INI) FILE

Before you can install a connector, you must create an initialization file (.ini) that loads and registers various components—connector flows, connection model templates, and transformers, for example—into the BusinessWare repository and makes them available through the Console. A connector's initialization (`.ini`) file contains information about the connector's name, its rep, and default values for its properties.

At a minimum, an initialization file (.ini) is needed to register a connector flow in the BusinessWare repository and install a flow into the Flow Palette on the BusinessWare Console. In addition, developers can create initialization files that:

- add connection model templates to the Console
- register transformer methods
- import and register IDL into the BusinessWare repository

You can accomplish these tasks by using the command-line tools listed in this table:

Task	Command-line Tool
Add connection model templates to the Connection Model menu in the Console	ConnUtil
Register transformer methods	TransReg
Import and register IDL into the repository	importidl

See the *BusinessWare Programming Reference* for information about using these tools.

You can write a Java program that will create the initialization file for you (as described in "Creating an .ini File From a Java Program" below), or you can use the Connector Generator tool, as described in the next sub-section.

Generating an .ini File By Using Connector Generator

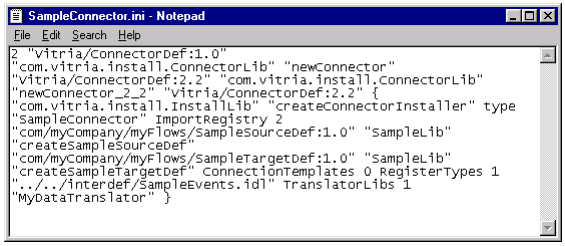
The Connector Generator Tool creates a default `.ini` file for a connector based on properties in the .gen file. You can generate the .ini file using either the wizard version of the Connector Generator or the command-line version, but to use the command-line version, an .gen file must already exist. See "Using the Connector Generator Tool and the .gen File" on page 42 for additional information about using the command line tool.

Do not modify the .ini file after it's been generated. Add the name of the initialization file (without the .ini) to the vtInstalled.cfg file, then run the loaditems script (if the BusinessWare installation has already been up and running). See "Installing a Connector" on page 108 for additional information about installation.

The generator tool can automatically generate the .ini file for the connector flow, but if you want to create an .ini file that includes a connection model or a connection model template, you must write a Java class that uses several key BusinessWare classes and methods, as described in the next section.

Creating an `.ini` File From a Java Program

Developers can create `.ini` files by writing a Java program that generates the file specifically for their connector. This section steps through two files from the *Simple Connector Sample*—`SampleLib.java` and `SampleIni.java`. When compiled and run, the `SampleIni.class` generates an .ini file (shown in the screenshot), which loads the connector, connection model templates, and the transformer into the Flow Registry:



You can adapt these two Java files as needed for your connector and any connection model templates and transformers.

NOTE: Use the fully-qualified package names for the connector code (rather than the non-qualified names shown in these code listings).

Step-through of the SampleIni.java Code

The `SampleIni.java` code begins with these imports:

```
import com.vitria.install.ConnectorLib;
import com.vitria.bclient.ConnectorDef;
import com.vitria.bclient.ConnectorRep;
import com.vitria.fc.io.ExportException;
import java.io.IOException;
```

Next is the name of the class, followed by the `main()` method; the `main()` method calls the `newSampleConnector` method, which provides the code for the bulk of the work of initializing (loading and registering) the connector components.

```
public class SampleIni {
public static void main(String[] args) throws ExportException, IOException {
    ConnectorDef cDef=newSampleConnector("SampleConnector“SampleConnector.ini”“);
    ConnectorLib.exportItem(cDef, “SampleConnector.ini”);
    System.out.println(“Generated ConnectorDef for Sample”);
}
public static ConnectorDef newSampleConnector(String type) {
    ConnectorRep rep = new ConnectorRep();
    rep.init();
    rep.setClassName(ConnectorLib.className);
    rep.setMethodName(ConnectorLib.methodName);
    rep.setConnectorType(type);
```

As shown above, the `newSampleConnector` method starts by setting the class name, method, and type for the connector. The next few sub-sections focus on the other tasks that the `newSampleConnector` method accomplishes.

Adding the Flow Type to the Import and Export Registry

These next two statements add export tags for the two new connectors to the Flow Registry:

```
// add the flow types to the import registry
    rep.addFlowType(SampleLib.SOURCE_EXPORT_TAG,
                    “SampleLib”,
                    “createSampleSourceDef”);
    rep.addFlowType(SampleLib.TARGET_EXPORT_TAG,
                    “SampleLib”,
```

```
                    "createSampleTargetDef");
```

Adding Connection Model Templates

These next two statements make the two connection model templates available on the Connection Models menu of the Console:

```
rep.addConnectionTemplateItem("Sample to Channel",
                              "SampleLib",
                              "createSampleSourceToChannel");
rep.addConnectionTemplateItem("Channel To Sample",
                                 "SampleLib",
                                 "createChannelToSampleTarget");
```

(Note that you'll still need to wire together the connection model itself. That's done in the `SampleLib.java` file, as described in "Step-through of the SampleLib.java Code" on page 105.)

Adding a Transformer Class

This statement adds a new transformer class to the registry:

```
rep.addTranslatorLib("MyDataTranslator");
```

All the methods in MyDataTranslator will be available from the connection model.

Registering IDL Files

The next statement registers the IDL in the repository, creating editable metadata.

```
// register the IDL in the repository
rep.addTypeToRegister("../../interdef/SampleEvents.idl");
return rep;
}
```

The interface definition language file that this section of code refers to must be located in the `interdef` directory in the BusinessWare installation.

This concludes the code for the `newSampleConnector` method in the `SampleIni.java` file. The remainder of the code in this chapter is from `SampleLib.java` of the *Simple Connector Sample*; the code in SampleLib will be used to embed the necessary information in the .ini file to create the connection model ("Channel to Sample") template with the associated event logger and transformer class.

Step-through of the SampleLib.java Code

The `SampleLib.java` file starts with the list of imports for the API classes and declares the classname:

```
import com.vitria.bclient.BClientLib;
import com.vitria.bclient.FlowManagerDef;
import com.vitria.fc.io.ImportStream;
import com.vitria.jct.EventLoggerDef;
import com.vitria.bfolder.ContextFolder;
import com.vitria.jct.JctLib;
import com.vitria.jct.SubscriberDef;
import com.vitria.jct.PublisherDef;
import com.vitria.connectors.common.ConnUtil;
import com.vitria.connectors.common.ConnectorLib;
import com.vitria.connectors.datalators.simpletranslator.SimpleTranslatorDef;
import com.vitria.connectors.datalators.simpletranslator.SimpleTranslatorLib;
public class SampleLib {
```

Defining Export Tags for Connectors

The .ini must create definitions for the export tag for the connector, to uniquely identify it in the Flow Registry. This code listing handles the export tags for the source and target connector for the Simple Connector; it also provides reference to the create methods that will provide a new Rep in the Console, when an end-user begins creating a connection model:

```
public static final String SOURCE_EXPORT_TAG =
  “com/myCompany/myFlows/SampleSourceDef:1.0”;
public static final String TARGET_EXPORT_TAG =
  “com/myCompany/myFlows/SampleTargetDef:1.0”;
public static SampleSourceDef createSampleSourceDef(ImportStream s) {
    return createSampleSourceDef();
}
public static SampleSourceDef createSampleSourceDef() {
    SampleSourceRep ssd = new SampleSourceRep();
    ssd.init();
    return ssd;
}
public static SampleTargetDef createSampleTargetDef(ImportStream s) {
    return createSampleTargetDef();
}

public static SampleTargetDef createSampleTargetDef() {
    SampleTargetRep std = new SampleTargetRep();
    std.init();
    return std;
}
```

You can use the ending characters of the export tags (the “1.0” in ...`SampleSourceDef:1.0` above) to differentiate versions of the connector. The duplication of the `createSampleTargetDef` method is for compatibility with previous versions of BusinessWare.

The next few code excerpts are highlights from the createConnectorSubscriberDef method in SampleLib.java. This method creates the connection model for a channel source to the Sample flow.

(SampleLib.java also contains a method () for the “Sample flow to the channel” connection model, not shown in the listings in this section; see *Simple Connector Sample* for complete information).

In simple terms, the method has several tasks, including creating the target flow def, the transformer, the event logger flow, and wiring all these elements together. The first line is the method signature; the second line creates the channel source def, which will be the source of events for the connection model template that the .ini file will ultimately load into the BusinessWare Console:

```
public static FlowManagerDef createChannelToSampleModel(){
SubscriberDef channel = JctLib.createSubscriberDef();
```

Creating and Adding Transformer Classes and Methods

If you want the connection model to include transformation methods, you can have the .ini file add the Simple Data Transformer flow and set the initial values for the class name and method. This section of the `SampleLib.java` code adds a transformer flow to the target list of the flows from which it is to receive events, and adds to the transformer's target list the flows to which it will send events:

```
// === create translation flow ===
SimpleTranslatorDef transformer = SimpleTranslatorLib.createSimpleTranslatorDef();
// initial transformer configuration
transformer.setClassName(“MyDataTransformer”);
transformer.setMethodName(“dateToIDLString”);
```

In the code above, createSimpleTranslatorDef() creates the standard Simple Transformer flow; transformer.setClassName() sets the name of the class that contains the transformer method; and transformer.setMethodName() sets the name of the transformer method to be used. The class name and method name will be the default values, which can be changed by the user of the connector in the Console.

NOTE: Use the fully-qualified package names for the connector code (rather than the non-qualified names shown in these code listings). For example, if you have a `mySpecial` connector with a stringToSpecial transformer() method in a class named MySpecialTransformer, all of which are members of a package named `myspecial`, the last two lines in the listing above would appear as follows:

```
transformer.setClassName(“myspecial.MySpecialTransformer”);
transformer.setMethodName(“stringToSpecial”);
```

For information about creating your own custom transformer methods, see “Customizing Methods” on page 93.)

Creating and Initializing the Def

The following code creates and initializes the sample target connector def:

```
SampleTargetDef std = createSampleTargetDef();
```

The following code wires all three defs together:

```
FlowManagerDef flowManager = BClientLib.createFlowManagerDef();
channel.getTargetList().append(translator);
translator.getTargetList().append(std);
flowManager.getSourceList().append(channel);
```

Notice in the previous line that the flowManager maintains a list of source flows, so that this connection model can have many source flows.

Creating the StopOn Flow

The code below creates and initializes the StopOn flow for the error model, and connects it to the Logger. To create the StopOn flow, add these lines to the Java file generating the connection model’s `.ini` file:

Adding the StopOn Flow to the Logger Target List

The code below creates and initializes a StopOn flow for the error model and connects it to the Logger:

```
SimpleTranslatorDef logTranslator =
  SimpleTranslatorLib.createStopOnTranslatorDef();
    logTranslator.setName("StopOn_Sample");
```

Next, the `createChannelToSampleModel` concludes by returning a Flow Manager object, the creation of which was the primary goal of this method:

```
return flowManager;
}
```

Finally, this next method calls the `createChannelToSampleModel` method above to create a flow manager:

```
public static void createChannelToSampleTarget(ContextFolder folder)
    throws com.vitria.fc.folder.UnresolvedException,
           com.vitria.fc.folder.NoPermissionException,
```

```
            com.vitria.fc.io.ExportException,
            com.vitria.fc.folder.OperationException
{
    FlowManagerDef flowManager = createChannelToSampleModel();

    // === bind the new flowmanager into the name space ===
    ConnUtil.newConnectionModel(folder, flowManager, “New Channel To Sample”);
}
```

Ultimately, when a business user selects the “New Channel To Sample” connection model from the menu on the Console, the createChannelToSample model is called, the flow manager creates the defs, reps, and instantiates the flow and wires it all together according to the specifications in the .ini file, and the template displays in the Connection Model on the Console.

INSTALLING A CONNECTOR

BusinessWare components, including connectors, can be installed on a single machine or across multiple machines, in client/server (distributed) fashion. Specifically, you can install and use the Console on one machine and have BusinessWare installed and running on a server located elsewhere, over a network.

In either case, the .class files that comprise the connector must be installed on the client machine where the Console is running. The next two sections discuss the files and how to install them in either a client/server or stand-alone server configuration.

CONNECTOR FILES

As discussed throughout this guide, particularly in Chapter 4, “Creating Java Components for a Connector Flow,” a connector flow is comprised of several Java .class files that enable business analysts or other end-users to manipulate and configure it (as a flow) in the context of a connection model in the Console. These include specifically .class files for the def, the rep, the BeanInfo, and the flow.

The connector flow also includes code for the transaction resource, if the transaction service will manage transactions for this flow. This summary table shows the naming standard, in which *name* is the connector name and type is the *type* of flow (source, target, or source-target) as well as the specific classnames that comprise the *EBank Connector Sample* target flow:

	Naming Standard	EBank Java Target Flow
Def	*nameType*Def	EBankJavaTargetDef.class
Rep	*nameType*Rep	EBankJavaTargetRep.class
BeanInfo	*nameType*RepBeanInfo	EBankJavaTargetRepBeanInfo.class
Flow	*nameType*Flow	EBankJavaTarget.class
[BaseFlow]	*nameType*FlowBase	na [EBankJavaTargetBase.class]
Transaction Resource	*nameType*Resource	EBankJavaTargetResource.class

Note that when the Connector Generator Tool is used to generate files, the flow is implemented as two files, a base and a flow file.

All these .class files for the connector must be in the `CLASSPATH` on the machine where BusinessWare server is installed; if you've implemented a client/server environment, you must also install these in the classpath on the client machine, where the Console is installed and running.

In addition, the files for any custom transformer classes implemented in your connection model must also be in the `CLASSPATH`.

- On Windows NT, C++ connectors also have C++ `.dll` files that must be in the PATH.
- On UNIX systems, C++ connectors also have C++ shared library (`.so`) files which must be in the shared library path.

You can install the files for a connector flow by following the steps in the next section. In addition to the .class files, you'll need an initialization (.ini) script, as discussed in "Creating an .ini File From a Java Program," to load and register the metadata and other objects in the BusinessWare repository.

Installing a Connector

All .class files for the connector must be located in the appropriate sub-directories on both the client machine and server (for a client/server (distributed) BusinessWare installation), or just on the server machine (for a stand-alone BusinessWare installation). The sub-directory will be specfic to win32 or UNIX, depending upon the platform on which BusinessWare is installed.

Developers should create an InstallShield (for Windows platform) or InstallAnywhere (for Unix) application to handle these installation tasks for their connector users.

In the steps below, replace *platform* with `win32` or `UNIX` as necessary for your install. To install a connector:

1. Copy all connector .class files—the def, rep, BeanInfo, and the flow—to `%VITRIA%\java\`*`platform`*, where *platform* is either `win32` or `UNIX`. This places them in the `CLASSPATH`.
   - For a C++-based flow, also copy the `.dll` (or on UNIX, the shared library `(.so)` file) for the connector to %VITRIA%`\bin\`*platform;* this places these files into the PATH (or shared library path).
2. Copy the connector's `.ini` file to `%VITRIA%\data\install`.
3. Add the name of the `.ini` file (without the `.ini` extension) to the end of the file `vtInstalled.cfg`. The default location of `vtInstalled.cfg` is `%VITRIA%\data\install.`
4. Run the `loaditems` script (also located by default in `%VITRIA%\data\install`. When you execute the script, the BusinessWare server must be running. The `loaditems` script reads the `vtInstalled.cfg` file and loads the matching .ini file. If the .ini file is found, and it is dated later than the last time loaditems was run, the objects created by the .ini are loaded into BusinessWare.
5. Open the Console and verify that the connector is displayed in the Flow palette; if the connector doesn't display in the Flow Palette:
   - Re-start the Console to force the JVM (Java virtual machine) to reload all byte code that comprise the Console).
   - Double-check the classpath and make sure that all .class files are in the right sub-directory under the %VITRIA% installation.
6. Test the connector by either creating a new connection model or importing an existing model, and then using the connector in the model. This typically requires configuring the connector flow, saving and registering the configuration, starting the connection model and testing the results.

If you're running the Console on a client machine rather than on the BusinessWare server, simply copy the connector's class files to a directory in the `CLASSPATH` on the machine on which the Console is running. By default, the classpath is %VITRIA`\java\`*platform*, where *platform* is either `win32` or `UNIX`.

A REFERENCE

The classes and interfaces you'll use to develop Connectors for BusinessWare belong to just a handful of Java packages and C++ include files. You'll find the C++ files in the \include sub-directory of the BusinessWare installation.

For Java development, you should have BusinessWare installed on your development machine and make sure that CLASSPATH is set according to the installation instructions. Vitria recommends using Sun's Java JDK (Java Development Kit) compiler, release 1.2.2. The command line compiler requires fewer resources than many of the IDEs on the market. In addition, Vitria doesn't provide the source code for the Java API, and some IDEs require that you have source code in order to write new applications.

Java API Summary

See *BusinessWare Programming Reference* for additional information.

Table A-1 com.vitria.connectors.BaseConnector

Method	Description
init()	Initializes state. This method must be called before the flow is started.
public ConnectorBaseDef getDef(){return def_;}	Returns the def associated with this flow
public void startEvents()	Called by the FlowManager when the connection model is started. Flows that perform startup work must override this method and call super.startEvents() after they have finished their startup work and before returning.
public void stopEvents(){	Called by the FlowManager when the connection model is stopped. Flows that perform shutdown work must override this method and call super.stopEvents() before doing shutdown work and before returning.

Table A-2 com.vitria.fc.flow.FlowEnv

Method	Function
int getInstanceCount()	Returns the total number of instances inside this connection model (multi-instance support).
int getInstanceIndex()	Returns the instance index that this flow env is running inside the connection model (multi-instance support).

These are the methods that you must implement in the Rep to communicate with the repository.

Table A-3 com.vitria.fc.io.Exportable

void exportTo(com.vitria.fc.io.ExportStream stream) throws com.vitria.fc.io.ExportException	Export the object to the given stream.
void importFrom(com.vitria.fc.io.ImportStream stream) throws com.vitria.fc.io.ImportException	Import the object from the given stream.
getExportFormat()	Return the tag that defines the format of the object in the stream

Table A-4 com.vitria.fc.trans.OnePhaseResource

Method	Function
prepareToCommit()	Returns the ID of the active transaction (one-phase transaction resources only).
commitResource()	Commits the transaction.
abortResource()	Kills the transaction
getPrepareStatus()	Returns a flag showing whether the active transaction has been committed yet (one-phase transaction resources only)

Table A-5 com.vitria.fc.trans.Resource

Method	Function
precommitResource()	Prepares the system for a commit (two-phase transaction resources only)

Method	Function
commitResource()	Commits the transaction.
abortResource()	Kills the transaction

Request/Response API

Includes the ResponseListener interface (public extends Subscriber) and the ResponseListenerLib class. A ResponseListener is an object that can be sent as a parameter of a request event, and then used to wait for a response event. ResponseListener objects are not persistent. Events sent to a ResponseListener that no longer exists will be lost. See Request/Reply Sample for information about how to use.

Table A-6 com.vitria.connectors.request.ResponseListener

public EventBody awaitResponse(long timeout);	Parameter timeout is the maximum number of milliseconds to wait for a response. Returns the response event (if it was received), or null if the timeout period expired.
public void cancel();	Stops listening for response events because the request event is cancelled.

Table A-7 com.vitria.connectors.request.ResponseListenerLib

Method	Description
public static ResponseListener createResponseListener()	Create a ResponseListener; the default logger will be used for diagnostic messages.
public static ResponseListener createResponseListener(DiagLogger log)	Create a ResponseListener. The log will be used for diagnostic messages.
public static ResponseListener createResponseListener(DiagLogger log, int logLevel)	Create a ResponseListener. The log and log level will be used for diagnostic messages.

Method	Description
public static ResponseListener createResponseListener(DiagLogger log, int logLevel)	Create a ResponseListener. The log and log level will be used for diagnostic messages.
public static void respond(Subscriber subscriber, EventBody response, DiagLogger log)	Send the response event to the ResponseListener. The subscriber is the ResponseListener that was sent in the request event.

After creating a ResponseListener, always call either awaitResponse() or cancel() on the ResponseListener.

Parameters:

- `log` The logger to use for logging messages. If null, the default logger will be used.
- `logLevel` The log level to use for logging messages.

C++ API Summary

Table 0-1 FlowBridge (fbutil.hxx)

void flowbridge_commit(vtFlow::Flow* flow);	Deprecated; C++ flows should not call commit. (This method calls commit on the connection model. If commit after is set, the JavaFlow will call commit automatically.)
void flowbridge_abort(vtFlow::Flow* flow);	Abort the transaction on the entire connection model. Note, this only set a flag that abort should occur. When flow returns * to the Java processing environment, an abort will be issued to the transaction service.
void flowbridge_logmessage(vtCORBA::Long iLevel, vtFlow::Flow* flow,char* component, char* message);	Log a string message to the logger. **Use this during development process only**. This method is kept for compatibility--all new code should use diagnostic variant to log internationalized message. iLevel specifies a log level that is checked before logging actually occurs.
void flowbridge_logevent(vtCORBA::Long major, vtCORBA::Long minor, char* message, vtFlow::Flow* flow);	Log a string message with major and minor code to the logger. * Note this method is kept for compatibility, use the internalized version instead.
void flowbridge_logmessage(vtCORBA::Long iLevel, vtFlow::Flow* flow, VTDiagnostic* diagMsg);	Log an internationalized diagnostic message to the logger.
void flowbridge_logerror(vtCORBA::Long iLevel, vtFlow::Flow* flow, VTDiagError* diagError);	Log an internationalized diagnostic message to the logger. iLevel specifies a log level that is checked before logging actually occurs.
void flowbridge_logeventbody(vtCORBA::Long major, vtCORBA::Long minor, char* message, vtFlow::EventBody* e, vtFlow::Flow* flow);	Log an message with an eventbody to the event logger. major major code minor minor code message string message describing event e the original event body that error occured on
void flowbridge_stopAllFlows(vtFlow::Flow* flow, char* message);	Stop all the flows in the connection model. Note this is not synchronous call, a thread is spawned which will issue the stop call. flow the C++ flow param message a string message describing why the flow is stopping.

Simple Translator Interface

Use methods from the `SimpleTranslatorInterface` to get information about the environment through the `FlowEnv`, and to log error and trace messages. The complete SimpleTranslatorInterface is as follows:

```
package com.vitria.connectors.datalators.simpletranslator;

import com.vitria.fc.diag.DiagLogger;
import com.vitria.fc.flow.FlowEnv;
```

```
import com.vitria.fc.flow.EventBody;

public interface SimpleTranslatorInterface {
  public DiagLogger getLogger();
  public FlowEnv getFlowEnv();
  public void log(String message, int level);
  public void log(String message);
  public void logException(Exception e, int level);
  public void logException(Exception e);
  public int getLogLevel();
  public void sendToTarget(EventBody evt);
  public void markIsResponse(boolean b);
  public EventBody[] execute(EventBody e);
  public EventBody[] executeAt(String flowName, EventBody
   e);
}
```

Here's an example of using the SimpleTranslatorInterface to to obtain information from the FlowEnv and to log messages:

```
public static EventBody RDBToString(EventBody evtIn, String[] separator,
  SimpleTranslatorInterface sti)
{
    // just send epoch events on (although this use of indexOf is not complete)
    if ( evtIn.getEventSpec().indexOf("epoch") > 0)
        return evtIn;
    Logger logger = sti.getLogger();
    EventDef evtDef = evtIn.getEventDef();
    List paramDefs = evtDef.getParameterList();
    Object[] inParams = evtIn.getParameters();
    String outstring = new String();
    for ( int i = 0; i < inParams.length; i++) {
        int kind = ((ParameterDef)paramDefs.elementAt(i)).getType().kind();
        outstring = outstring + inParams[i] + separator[0];
        if ( logger.getTraceLevel() >= diagLogger.VERBOSE)
        logger.trace(diagLogger.VERBOSE,"Param " + i + " = " + inParams[i] + "
  type = " + kind);
    }
    EventDef e =
  (EventDef)(BasicEventsHelper.getMetaObject().findDef("stringEvent"));
    Object objs[] = new Object[1];
    objs[0] = outstring;
    EventBody ebOut = JctLib.createEventBody(e, objs);
    return ebOut;
}
```

GLOSSARY

Channel: A structure in BusinessWare that holds events.

Channel source: Aflow that receives events from a channel.

Channel target: A flow that sends events to a channel.

Connection model: A collection of flows and support programs that send data between end points (which may be channels or external systems).

Connector: A flow that connects an external system with BusinessWare, either converting external system data to BusinessWare events or vice versa. (Note that not only are connector flows usually referred to simply as "connectors," but also that what is referred to as a "connector" sometimes comprises a suite of connector flows, transformers, channel and source target flows, and templates. For instance, the "File Connector" includes not just the File Source and File Target connector flows, but also several pre-wired templates.)

Container server: The underlying software infrastructure required to instantiate an object.

Event: A generic data-exchange format that contains a block of bytes that have meaning within the context of a specific BusinessWare implementation. A Vitria-specific keyword added to CORBA IDL as implemented in BusinessWare.

External system - the data repository (such as a database management system, file system, or message queue) that your connector extracts data from or inserts data into.

Flow: A component that can receive events, do processing, and send events to other flows. Flows can be linked or "wired" together to construct a connection model that defines a runtime process that interacts with an external system.

Interface: In the context of CORBA, an interface is an object that describes the set of services (operations) that a CORBA server offers to a client. A CORBA interface is defined using the OMG (Object Management Group) IDL (interface definition language). Similarly, in Java an interface is a list of the methods that can be used by a client application.

Module: A structure that defines a namespace for a set of interfaces.

Source connector: A connector that creates events from data from an external system.

Source-target flow: A flow that both sends and receives events; typically referred to as a transformer because a source-target flow usually translates (tranforms) an event of one type into an event of another type (on the output side of the flow). The RDBMS Connector is the only source-target flow that is not considered a transformer.

Target connector: A connector that receives events and interacts with an external system.

Transaction: A logical unit of work that has ACID (atomicity, consistency, integrity, durability) properties after being processed. In simple terms, a group of events that must not be divided: that the events need to either be all delivered or none of them be delivered.

Transaction resource: Acomponent of each flow in a connection model that handles the code related to transactions in the external system, including committing or aborting the transaction.

Transaction service: A component of BusinessWare that ensures that all the events that participate in a particular transaction pass through the connection model safely if the transaction is committed, pass through the connection model without being committed, or do not pass through the connection model if the transaction is aborted.

Transformer: A source-target flow whose output event is different than the input event it receives. (In the API, the term “translator” is used; this guide uses the term translator only when explicitly referring to the API.)

INDEX

R

S

T

V