Generic Multi-Mode e*Way Extension Kit User's Guide

Release 5.0.5 for Schema Run-time Environment (SRE)

Contents

List of Figures

List of Tables

Preface

This Preface identifies the intended reader, describes nomenclature and writing conventions, outlines the organization of information, and provides a list of related documents.

P.1 Intended Reader

The reader of this guide is presumed to be an experienced Java programmer who wants to create e*Ways, e*Way Connections, and Event Type Definition (ETD) builders and deploy them for e*Gate end users. As a reader of this guide, you should have a thorough understanding of the following:

- The external application for which the extension is to be written.
- The structure and format of the data to be modeled in your ETDs, which will be transported and transformed by your e*Ways and e*Way Connections.
- Terminology, concepts, and operation of e*Gate Integrator 4.5.1 and later:
  - Java-based Events, ETDs, and the ETD Editor.
  - Java Collaborations, Collaboration Rules, and the Collaboration Rules Editor.
  - Java-based e*Ways, e*Way Connections, and the Configuration Editor.
  - Java programming language and environment; Java Native Interface (JNI).

In addition, as a reader of this guide, you should have familiarity with Microsoft Visual Basic.

P.2 Organization

This User's Guide is organized into the following chapters:

Chapter 1 **"Introduction"**: Introduces the guide and lists prerequisites for installing and using the kit.

Chapter 2 **"Installation"**: Describes the procedures for installing the kit on Windows and UNIX operating systems.

Chapter 3 **"Architectural Overview"**: Reviews e*Gate concepts and terminology, explains the purpose and function of e*Way Connections, and provides an in-depth summary of the technical material covered in the rest of the guide.

Chapter 4 **"Functional Flow"**: Explains the run-time behavior of the Multi-Mode e*Way from startup to shutdown.

Chapter 5 **"e*Way Development Methodology"**: Provides development and design considerations for creating an e*Way.

Chapter 6 **"e*Way Development Workflow"**: Explains the typical steps you need to take to develop, deploy, and validate a custom e*Way Connection.

Chapter 7 **"Event Type Definitions"**: Describes the e*Gate Event Type Definitions (ETDs), focusing on their role in the Generic Multi-Mode e*Way Extension Kit User's Guide.

Chapter 8 **"Developing an e*Way Using ETD Builder Components"**: Provides a detailed discussion of the concepts and common practice for creating an ETD Builder—the tool used by end users to create ETDs.

Chapter 9 **"Developing an e*Way Using the Builder API"**: Guides you through the creation and deployment of a simple ETD for a fictional accounting application. For the purpose of the sample, the APIs called by the ETD are simulated using a Remote Method Invocation (RMI) server.

Chapter 10 **"Developing the Automatic e*Way Connection"**: Guides you through the creation and deployment of a sample project, discussing the sample code and providing steps for compiling it, creating the **.xsc** files, **.def** files, and **.ctl** files, deploying the new e*Way Connection component, and using it in a sample Collaboration.

Chapter 11 **"Developing an e*Way Connection With Connection Management"**: Provides a similar sample to that in **Chapter 10**, guiding you through the creation and deployment of a sample project that uses connection management.

Chapter 12 **"Developing a Transactional e*Way Connection"**: Provides a similar sample to those in **Chapter 10** and **Chapter 11**, guiding you through the creation and deployment of a sample project that uses XA.

Chapter 13 **"Best Practices"**: Recommends best practices for developing and deploying a custom e*Way.

Chapter 14 **"e*Way Classes and Methods"**: Lists the Java classes that contain the methods used to extend the functionality of an e*Way. Also provides instructions for accessing the Generic Multi-Mode e*Way Extension Kit Javadocs.

Appendix A **"Extending the .def File"**: Provides an in-depth description of the syntax and keywords in the **.def** (default configuration-file template) file, to enable you to enhance its capabilities.

Appendix B **"The XSC Format"**: Specifies the content, structure, and rules governing XSC elements and attributes.

Appendix C **"The RMI Server"**: Describes the use of an RMI server to simulate a generic external system.

Javadocs: The API documentation is provided as Javadoc files on the e*Gate Integrator Installation CD-ROM.

Note: *Refer to the* ***Readme.txt*** *file for any last-minute information regarding the Generic Multi-Mode e*Way Extension Kit.*

P.3 Writing Conventions

The writing conventions listed in this section are observed throughout this document.

Hypertext links

When you are using this guide online, cross-references are also hypertext links and appear in **blue text** as shown below. Click the **blue text** to jump to the section.

Command line

Text to be typed at the command line is displayed in a special font as shown below.

```
java -jar ValidationBuilder.jar
```

Variables within a command line are set in the same font and bold italic as shown below.

```
stcregutil -rh host-name -rs schema-name -un user-name
-up password -ef output-directory
```

Code and samples

Computer code and samples (including printouts) on a separate line or lines are set in Courier as shown below.

```
Configuration for BOB_Promotion
```

However, when these elements (or portions of them) or variables representing several possible elements appear within ordinary text, they are set in *italics* as shown below.

path and *file-name* are the path and file name specified as arguments to **-fr** in the **stcregutil** command line.

Notes and cautions

Points of particular interest or significance to the reader are introduced with ***Note***, ***Caution***, or ***Important***, and the text is displayed in *italics*, for example:

Note: *The Actions menu is only available when a Properties window is displayed.*

User input

The names of items in the user interface such as icons or buttons that you click or select appear in **bold** as shown below.

Click **Apply** to save, or **OK** to save and close.

File names and paths

When names of files are given in the text, they appear in **bold** as shown below.

Use a text editor to open the **ValidationBuilder.properties** file.

When file paths and drive designations are used, with or without the file name, they appear in **bold** as shown below.

In the **Open** field, type **D:\setup\setup.exe** where **D:** is your CD-ROM drive.

Parameter, function, and command names

When names of parameters, functions, and commands are given in the body of the text, they appear in **bold** as shown below:

The default parameter **localhost** is normally only used for testing.

You can use the **stccb** utility to start the Control Broker.

Chapter 1

Introduction

The Generic Multi-Mode e*Way Extension Kit enables you to develop custom Multi-Mode e*Ways and e*Way Connections using Java with e*Gate 4.5.1 or later. This guide provides an overview of the architecture, functional flow, and development methodology of Multi-Mode e*Ways. In addition, this guide explains how to use the kit to create and deploy the e*Ways and e*Way Connections to communicate with each of your existing IS systems, networks, and/or applications.

Note: *This guide describes the e*Way development process for each supported version of e*Gate. Any steps that are unique to a particular version of e*Gate are noted.*

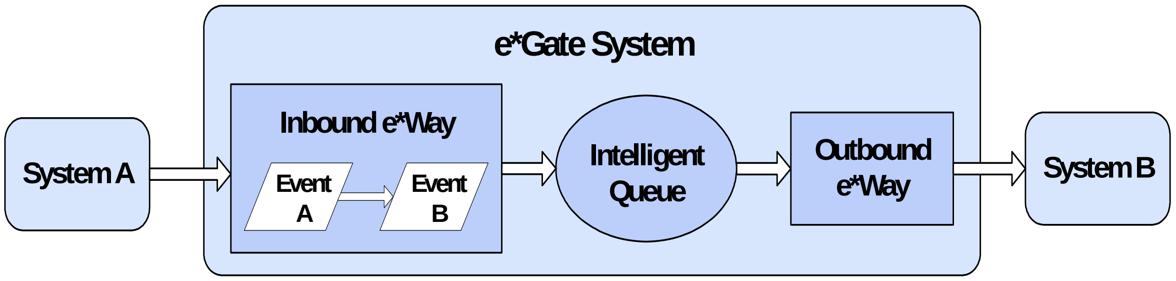
1.1 Overview

e*Way Intelligent Adapters (*e*Ways*) are the gateways to the e*Gate Integrator system. They bring data from outside to inside e*Gate, pass data from inside to outside e*Gate, or both. e*Ways establish connectivity with applications in either of the following ways:

- **Inbound**: receives unprocessed data from external components, transforms it into Events, and forwards it to other e*Gate components.
- **Outbound**: sends processed data to external components.

In e*Gate, e*Ways are mainly responsible for data transport. Figure 1 shows a simplified system setup of data flow.

Figure 1 Basic e*Gate Data Flow Relationships



In the sample system shown in Figure 1, e*Gate system data flow occurs in the following basic steps:

- Events (which are predefined data packets) flow into e*Gate from an external system (System A) through an inbound e*Way.
- The inbound e*Way transforms Events from Event Type A to Event Type B, and then places them in an Intelligent Queue (IQ) for temporary storage.
- The outbound e*Way takes the Events from the IQ and sends them out of e*Gate to another external system (System B), without changing them.

*e*Way Connections* are the encoding of access information for one particular gateway to an external system. An e*Way with an e*Way Connection functions in the same manner as shown in Figure 1, except that it can transfer multiple components of data in both directions at the same time, to and from multiple IQs. Also, any Java-enabled Collaboration can use several e*Way Connection components simultaneously, to transfer data to and from multiple external systems.

A *Multi-Mode e*Way* is a flexible multi-threaded type of e*Way executable (**stceway.exe**) that uses e*Way Connections as its interfaces between the e*Gate Java Collaboration Service (JCS) and external systems.

The following sections of this introductory chapter discuss supported operating systems and prerequisites for using the kit.

1.2 Supported Operating Systems

The Generic Multi-Mode e*Way Extension Kit is available on the same operating systems as the e*Gate Integrator. For more information, see the **readme.txt** file provided on the installation CD.

1.3 System Requirements

To develop e*Ways using the Generic Multi-Mode e*Way Extension Kit, you need the following:

- An e*Gate Participating Host.
- A TCP/IP network connection.
- A computer running Windows, to allow you to use the e*Gate Schema Designer and ETD Editor
- Additional disk space for e*Way executable, configuration, library, and script files. The disk space is required on both the Participating and the Registry Host. Additional disk space is required to process and queue the data that this e*Way processes; the amount necessary varies based on the type and size of the data being processed, and any external applications performing the processing.

Installed on the Participating Host

- Java JDK version 16.1_19

The e*Way must be configured and administered using the Schema Designer.

Note: *Additional disk space may be required to process and queue the data that this e*Way processes. The amount necessary can vary based on the type and size of the data being processed and any external applications doing the processing.*

1.3.1 Host System Requirements

The external system requirements are different for a GUI (Graphical User Interface) host machine—specifically a machine running the ETD Editor and the Java Collaboration Editor GUIs---versus a Participating Host which is used solely to run the e*Gate schema.

GUI Host Requirements

To enable the GUI editors to communicate with the external system, the following items must be installed on any host machines running the GUI editors:

- If you are using driver types 1, 2, or 3, the client library for your specific database installed on Windows to utilize the ETD builder.
- ODBC driver.

1.4 Prerequisites for Installing and Using the Kit

The Generic Multi-Mode e*Way Extension Kit requires a pre-existing installation of e*Gate.

Chapter 2

Installation

This chapter describes the procedures for installing the Generic Multi-Mode e*Way Extension Kit.

- **"Installing the Generic Multi-Mode e*Way Extension Kit on Windows" on page 23**
- **"Installing the Generic Multi-Mode e*Way Extension Kit on UNIX" on page 24**
- **"Files/Directories Created by the Installation" on page 25**

2.1 Installing the Generic Multi-Mode e*Way Extension Kit on Windows

You install this kit as you would any other application Add-on in accordance with the following instructions.

2.1.1 Pre-installation

- Exit all Windows programs before running the setup program, including any antivirus applications.
- You must have Administrator privileges to install the kit.

2.1.2 Installation Procedure

To install the Generic Multi-Mode e*Way Extension Kit on a Windows system

1. Log in as an Administrator to the workstation on which you are installing the kit.
2. Insert the installation CD-ROM into the CD-ROM drive.
3. If the CD-ROM drive's Autorun feature is enabled, the setup application launches automatically; skip ahead to step 4. Otherwise, use the Windows Explorer or the Control Panel's **Add/Remove Applications** feature to launch the file **setup.exe** on the CD-ROM drive.
4. The InstallShield setup application launches. Follow the installation instructions until you arrive at the **Please choose the product to install** dialog box.
5. Select e*Gate Integrator, then click **Next**.

6 Follow the on-screen instructions until you come to the second **Please choose the product to install** dialog box.

7 Clear the check boxes for all selections except Add-ons, and then click **Next**.

8 Follow the on-screen instructions until you come to the **Select Components** dialog box.

9 Select (but do not check) **Agents**, and then click **Change**. The **SelectSub-components** dialog box appears.

10 Select **kit**. Click **Continue** to return to the **Select Components** dialog box, then click **Next**.

11 Follow the rest of the on-screen instructions to install the kit. Be sure to install the kit files in the client installation directory suggested by the installation utility.

Note: *Unless you are directed to do so by Oracle support personnel, do not change the suggested installation directory setting.*

12 After installation is complete, exit the installation utility and launch the Schema Designer.

For more information about configuring the kit, see the **e*Gate Integrator User's Guide**.

2.2 Installing the Generic Multi-Mode e*Way Extension Kit on UNIX

2.2.1 Pre-installation

Root privileges are not required to install the kit. Log in with the user name of the user who will own the kit files. Be sure that this user has sufficient privileges to create files in the e*Gate directory tree.

2.2.2 Installation Procedure

To install the Generic Multi-Mode e*Way Extension Kit on a UNIX system

1 Log in on the workstation containing the CD-ROM drive, and insert the CD-ROM into the drive.

2 If necessary, mount the CD-ROM drive.

3 At the shell prompt, type

cd /cdrom

4 Start the installation script by typing

setup.sh

5 A menu of options will appear. Select the **Install e*Way** option. Then, follow the additional on-screen directions.

6 Follow the rest of the on-screen instructions to install the kit. Be sure to install the kit files in the client installation directory suggested by the installation utility.

Note: *Unless you are directed to do so by Oracle support personnel, do not change the suggested installation directory setting.*

7 After installation is complete, exit the installation utility and launch the Schema Designer.

For more information about configuring the kit, see the **e*Gate Integrator User's Guide**.

2.3 Files/Directories Created by the Installation

The kit installation process installs the files within the e*Gate directory tree. Files are installed within the **egate\client** tree on the Participating Host and committed to the **default** schema on the Registry Host.

The files listed in the following table are provided on the installation CD-ROM.

Note: *Files named with **_451** are for use with e*Gate version 4.5.1.*

Table 1 Product Files Shipped on the CD-ROM

File	Purpose / Contents
ewayETDxsc.xsd	XML Schema Definition file for XSC specification. It validates the **.xsc** files. ***Note:*** *This file only specifies those attributes that are relevant to e*Way ETDs.*
gmeek.taz	Compressed file containing source code, ETD files, and compile and install scripts for the sample ETDs. For detailed contents, see **Table 3 "Product Files Contained in File gmeek.taz" on page 26**.
GmeekDemoEway Sample.zip ▪ MySchema.zip ▪ TcpEcho.zip ▪ XAFile.zip	Sample e*Gate schemas for validating the various samples: ▪ Schema for the simple XSC 0.4–compliant e*Way sample. ▪ Schema for the Connection Management e*Way sample. ▪ Schema for the Transactional e*Way sample.
GmeekDemoEway Sample_451.zip	Sample e*Gate version 4.5.1 schema for validating the schema for the simple XSC 0.4–compliant e*Way sample.
INDATA.zip	Compressed file containing sample input data for validating the samples.
VBGmeekWizard.zip	Compressed file containing the Visual Basic forms and project files, libraries, and other files for creating a wizard (front end) for the "heavyweight" ETD Builder sample.

Decompressing the **common.taz** file loads the files shown in Table 2.

Table 2 Product File Shipped on the CD-ROM in setup\addons\gmeek

File	Purpose / Contents
common.taz	Compressed file containing two **.jar** files (**stcewcommonbuilder.jar** and **stcgmeek.jar**) and two **.ctl** files (**gmeek.ctl** and **connectionpoint.ctl**) used by the kit in developing ETD builders.
Readme.txt	Technical notes and installation instructions.

Decompressing the **gmeek.taz** file loads the files shown in Table 3 to the **gmeek** folder:

Table 3 Product Files Contained in File gmeek.taz

File	Purpose
In **gmeek/SampleETD/** ▪ EwayConnectionETDImpl.java ▪ SampleETD.java ▪ SampleETDConnector.java ▪ SampleETDDefs.java ▪ SampleETDExternalClass.java ▪ SampleETDExternalException.java ▪ SampleETDTester.java ▪ compile.bat ▪ compile.sh	▪ A simple (mostly no-op) implementation of the ETD interface. ▪ ETD java source. ▪ Connector class source. ▪ Property names from e*Way connection config file. ▪ Sample to illustrate interface to an external. ▪ Sample to support the external. ▪ Source of stand alone ETD tester. ▪ Windows batch script for compiling the source. ▪ UNIX shell script for compiling the source.
In **gmeek/TcpClientETD/** ▪ server/TCPEchoServer.java ▪ server/compile.bat ▪ server/compile.sh ▪ server/runServer.bat ▪ server/runServer.sh ▪ EwayConnETDConnectorExtImpl.java ▪ EwayConnectionETDImpl.java ▪ TcpClient.java ▪ TcpClientETD.java ▪ TcpClientETDConnector.java ▪ TcpClientETDDefs.java ▪ TcpClientException.java ▪ compile.bat ▪ compile.sh	▪ Source for a sample socket server. ▪ Windows batch script for compiling the source. ▪ UNIX shell script for compiling the source. ▪ Windows script to start the socket server. ▪ UNIX shell script to start the socket server. ▪ A simple (mostly no-op) implementation of the connector extension interface. ▪ A simple (mostly no-op) implementation of the ETD interface. ▪ A sample socket client. ▪ ETD java source. ▪ Connector class source. ▪ Property names from e*Way connection config file. ▪ Sample to support the external. ▪ Windows batch script for compiling the source. ▪ UNIX shell script for compiling the source.

Table 3 Product Files Contained in File gmeek.taz (Continued)

File	Purpose
In **gmeek/XAFileETD/** ▪ EwayConnectionETDImpl.java ▪ Resource.java ▪ XAFile.java ▪ XAFileETD.java ▪ XAFileETDConnector.java ▪ XAFileETDDefs.java ▪ XidValue.java ▪ compile.bat ▪ compile.sh	▪ A simple (mostly no-op) implementation of the ETD interface. ▪ XA resource implementation. ▪ Source sample. ▪ ETD java source. ▪ Connector class source. ▪ Property names from e*Way connection config file. ▪ This is an implementation of the Xid class for XA. ▪ Windows batch script for compiling the source. ▪ UNIX shell script for compiling the source.
In **gmeek/builder/javawizard/** ▪ GmeekWizard.java ▪ JConverterIntf.java ▪ RegModel.java ▪ compile.bat ▪ compile.sh	▪ Sample Java Wizard GUI. ▪ Interface class for the Java Wizard. ▪ Utility class for eGate registry. ▪ Windows batch script for compiling the source. ▪ UNIX shell script for compiling the source.
In **gmeek/builder/apiDemo/** ▪ GmeekDemoBuilder.java ▪ compile.bat ▪ compile.sh ▪ runapidemo.bat ▪ runapidemo.sh ▪ runapidemo1.bat ▪ runapidemo1.sh	▪ Sample builder program that uses the GMEEK builder API. ▪ Windows batch script for compiling the source. ▪ UNIX shell script for compiling the source. ▪ Windows script to run the builder to generate e*Way runtime. ▪ UNIX script to run the builder to generate e*Way runtime. ▪ Windows script to run the builder to generate ETD. ▪ UNIX script to run the builder to generate ETD.
In **gmeek/builder/rmiDemoSvr/** ▪ RmiDemoClient.java ▪ RmiDemoSvr.java ▪ RmiDemoSvrImpl.java ▪ RmiDemoSvrIntf.java ▪ compile.bat ▪ compile.sh ▪ runRmitest.bat ▪ runRmitest.sh ▪ runrmiclient.bat ▪ runrmiclient.sh	▪ Sample RMI client source. ▪ Sample RMI server source. ▪ Sample RMI server implementation. ▪ Sample RMI server remote interface. ▪ Windows batch script for compiling the source. ▪ UNIX shell script for compiling the source. ▪ Windows script to run the RMI server. ▪ UNIX script to run the RMI server. ▪ Windows script to run RMI client. ▪ UNIX script to run RMI client.
▪ In **gmeek/classes/** ▪ stcexception.jar	▪ Contains the exceptions that may be thrown in the GmeekDemoEwaySample_451 sample.
▪ In **gmeek/installETD**/ ▪ installETD.bat ▪ installETD.sh	▪ Windows script to install a ETD in command line. ▪ UNIX script to install a ETD in command line.

Table 3 Product Files Contained in File gmeek.taz (Continued)

File	Purpose
In **gmeek/installETD/ETD4Account/** ▪ Account.ctl ▪ Account.jar ▪ Account.xsc	e*Gate-specific files for the Accounting example. ▪ Sample ETD .install ctl file. ▪ Sample ETD .jar. ▪ Sample ETD .xsc.
In **gmeek/installETD/SampleETD/** ▪ SampleETD.ctl ▪ SampleETD.def ▪ SampleETD.xsc ▪ connectionpoint.ini ▪ install.ctl ▪ runTester.bat ▪ runTester.sh	 ▪ ETD .ctl file. ▪ Connection point .def. ▪ Sample ETD .xsc. ▪ Content to append to connectionpoint.ini. ▪ Sample ETD install .ctl file. ▪ Windows script to run the test. ▪ UNIX script to run the test.
In **gmeek/installETD/TcpClientETD/** ▪ TcpClientETD.ctl ▪ TcpClientETD.def ▪ TcpClientETD.xsc ▪ connectionpoint.ini ▪ install.ctl ▪ runTester.bat ▪ runTester.sh	 ▪ ETD .ctl file. ▪ Connection point .def. ▪ Sample ETD .xsc. ▪ Content to append to connectionpoint.ini. ▪ Sample ETD install .ctl file. ▪ Windows script to run the test. ▪ UNIX script to run the test.
In **gmeek/installETD/XAFileETD/** ▪ XAFileETD.ctl ▪ XAFileETD.def ▪ XAFileETD.xsc ▪ connectionpoint.ini ▪ install.ctl ▪ runTester.bat ▪ runTester.sh	 ▪ ETD .ctl file. ▪ Connection point .def. ▪ Sample ETD .xsc. ▪ Content to append to connectionpoint.ini. ▪ Sample ETD install .ctl file. ▪ Windows script to run the test. ▪ UNIX script to run the test.
In **gmeek/installETD/utils/** ▪ errs.sh ▪ inifile.sh ▪ utils.sh	 ▪ UNIX script. ▪ UNIX script. ▪ UNIX script.
In **gmeek/installEWAY/** ▪ installEWAY.bat ▪ installEWAY.sh	 ▪ Windows script to install e*Way. ▪ UNIX script to install e*Way.
In **gmeek/installEWAY/utils/** ▪ errs.sh ▪ inifile.sh ▪ utils.sh	 ▪ UNIX script. ▪ UNIX script. ▪ UNIX script.

Table 3 Product Files Contained in File gmeek.taz (Continued)

File	Purpose
In **gmeek/installEWAY/GmeekDemoEway/**	
▪ DemoRmiConnector.java	▪ Sample connector class source.
▪ ETDWizards.ini	▪ Content to append to ETDWizards.ini.
▪ EwayConnectorExtImpl.java	▪ Sample connector base class source.
▪ GmeekDemoEway.def	▪ Connection point .def.
▪ GmeekDemoEwayETDbuilder.jar	▪ Back-end builder .jar.
▪ GmeekDemoEwayrt.jar	▪ e*Way runtime .jar.
▪ gmeekdemoewaywizard.ctl	▪ Wizard .ctl file.
▪ GmeekWizard.bmp	▪ Heavy Visual Basic wizard icon bitmap.
▪ GmeekWizard.dll	▪ Heavy Visual Basic wizard ActiveX dll.
▪ GmeekWizard.jar	▪ Java wizard jar (works with lightweight Visual Basic).
▪ GmeekWizardLight.bmp	▪ Lightweight Visual Basic wizard icon bitmap.
▪ GmeekWizardLight.dll	▪ Lightweight Visual Basic wizard ActiveX dll (works with Java wizard).
▪ RmiDemoSvr.jar	▪ Sample external system .jar.
▪ addon.log	▪ Content to append to addon.log.
▪ compile.bat	▪ Windows batch script for compiling the source.
▪ compile.sh	▪ UNIX shell script for compiling the source.
▪ connectionpoint.ini	▪ Content to append to connectionpoint.ini.
▪ stcewgmeekdemoeway.ctl	▪ e*Way .ctl file used to load the e*Way.
In **gmeek/installEWAY/GmeekDemoEway_451/**	
▪ DemoRmi_451Connector.java	▪ Sample connector class source.
▪ EwayConnectorImpl.java	▪ Sample connector base class source.
▪ GmeekDemoEwayETDbuilder_451.jar	▪ Back-end builder .jar.
▪ GmeekDemoEway_451.def	▪ Connection point .def.
▪ GmeekDemoEway_451rt.jar	▪ e*Way runtime .jar.
▪ RmiDemoSvr.jar	▪ Sample external system .jar.
▪ addon.log	▪ Content to append to addon.log.
▪ compile.bat	▪ Windows batch script for compiling the source.
▪ compile.sh	▪ UNIX shell script for compiling the source.
▪ connectionpoint.ini	▪ Content to append to connectionpoint.ini.
▪ stcewgmeekdemoeway_451.ctl	▪ e*Way .ctl file used to load the e*Way.
▪ tempaddon.log	▪ Content to append to temporary addon.log.
In **gmeek/testdata/**	
▪ test.prop	▪ Test for SampleETD.

Chapter 3

Architectural Overview

This chapter provides an overview of how e*Ways operate and describes the components that comprise an e*Way. It then describes Multi-Mode e*Ways, describing the architecture of their component parts and their relationship with Collaborations and Event Type Definitions. It also describes the architecture and relationship between Multi-Mode e*Ways and e*Way Connections.

3.1 Overview of e*Way Operation

e*Ways provide points of contact between the e*Gate system and external applications. e*Ways handle the communication details necessary to send and receive information, including:

- Responding to or generating positive and negative acknowledgments
- Rules that govern resend and reconnect criteria
- Time-out logic
- Data envelope parsing and reformatting
- Buffer size
- Retrieval/transmission schedules

In addition to handling communications, e*Ways are also able to apply business logic within Collaboration Rules to perform any of e*Gate's range of data identification, manipulation, and transformation operations.

e*Ways are tailored to meet the communication requirements of a specific application or protocol.

3.1.1 Component Parts

Functionally, each individual e*Way contains the following component parts:

- **Executable Component**: An .exe file, this component is the engine of the e*Way. It performs the operations necessary to send, receive, and process data.
- **Configuration Files**: These .cfg files store the parameters that govern the e*Way's functions. For example, the configuration for a TCP/IP e*Way specifies the port numbers to send and receive data. The configuration for a file-based e*Way specifies the name of the directory to poll for input data.

- **Library Files**: These files (such as .dll files under Windows) support the operations that the executable component and functions require.
- **Function Definitions:** Depending on the e*Way, these functions are written in C, Java, or Monk.

Library files are loaded automatically by the script or executable file that calls them. All the other listed components are associated with the e*Way using either the Schema Designer or the e*Way Configuration Editor.

An e*Way derives its character from the e*Way executable file (**stcew*.exe**) it uses and the particular configuration file created and tailored for it. All parts of the e*Way, including the executable file, its configuration file, its Monk scripts and library files (if any), are either properties of the e*Way component or called by elements of those properties. The logic that the e*Way executes to process information is carried out by Collaborations assigned to each e*Way.

The procedures required to create and configure the e*Way component within the Schema Designer are discussed in the *e*Gate Integrator User's Guide*.

3.2 Multi-Mode e*Ways

A Multi-Mode e*Way is a multi-threaded component that extends the routing and transforming of data within e*Gate, and exchanges information with multiple external systems. A Multi-Mode e*Way uses e*Way Connections to send and receive topics directly to and from multiple external systems, and Oracle SeeBeyond JMS IQ Managers, or both.

The e*Way Connections are gateways to external systems, allowing a single e*Way to adopt several configuration profiles simultaneously to communicate with external systems.

3.2.1 Multi-Mode e*Way Characteristics

Multi-Mode e*Ways have the following characteristics:

- **Adapting**: Multi-Mode e*Ways are multifaceted, as they must interact with and adapt to multiple external systems. They normally communicate with e*Gate as well, but it is possible to configure a Multi-Mode e*Way so that it merely bridges between two or more external systems without bringing data into e*Gate.
- **Transporting**: Acting as “smart” gateways, Multi-Mode e*Ways direct the flow of multiple components of data in and out of e*Gate.
- **Collaborating**: Inbound and outbound e*Gate Collaborations reside in Multi-Mode e*Ways and form the core of their operation. They determine:
  - The routing (publishing/subscribing) of the Events they handle.
  - Any transformation of data as it passes through the Multi-Mode e*Way.

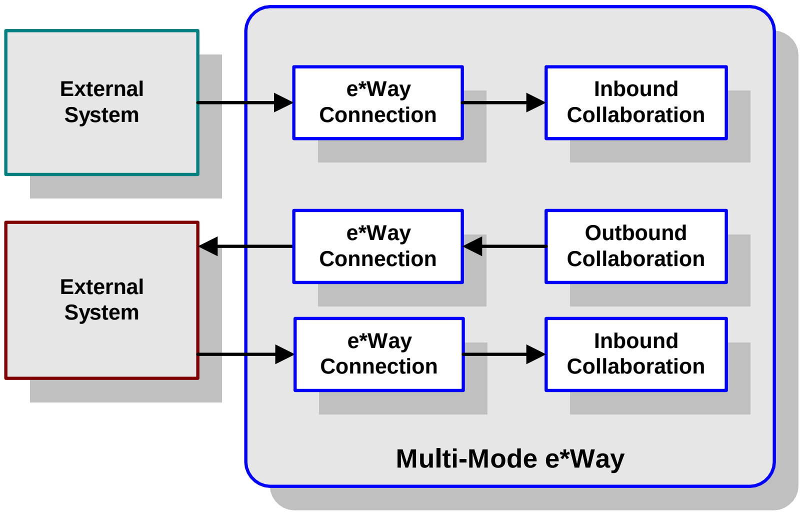
Multi-Mode e*Ways interact with Collaborations as follows:

- Every Multi-Mode e*Way requires at least one Collaboration, but it can have more than one.
- The Collaboration(s) hosted by a Multi-Mode e*Way must not use **<EXTERNAL>** as either a subscription source or a publication destination.
- Every Multi-Mode e*Way Collaboration that publishes internal e*Gate Events requires at least one IQ.

Multi-Mode e*Way properties are set in the e*Gate Schema Designer.

Figure 2 shows the flow of a Multi-Mode e*Way.

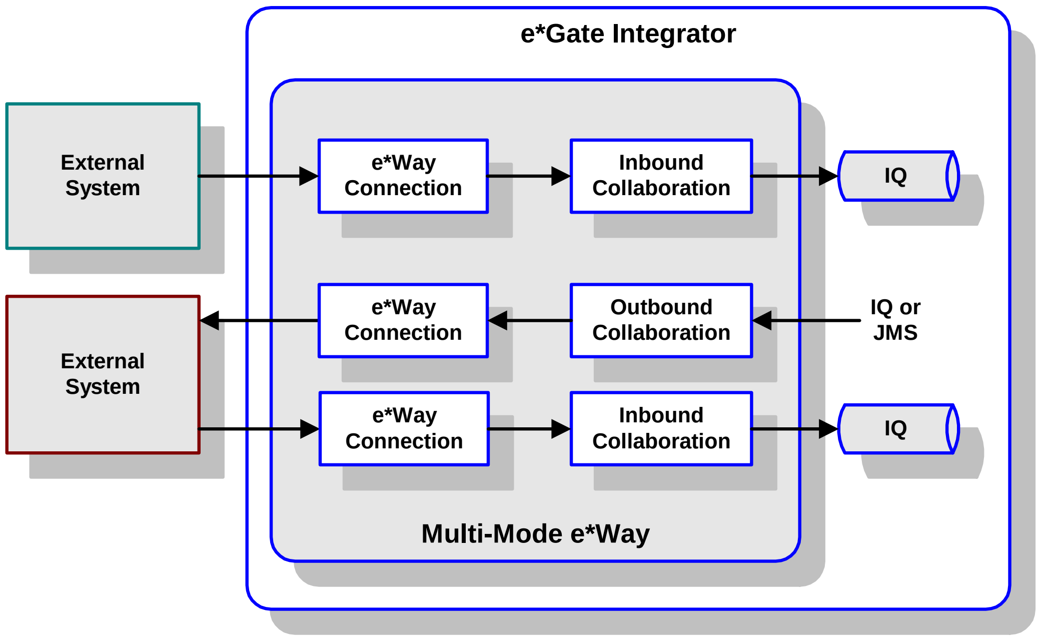
Figure 2 Multi-Mode e*Way



Each e*Way performs one or more *Collaborations*. Bidirectional data flow requires at least two Collaborations, one *Inbound* and one *Outbound*, as shown in Figure 2. Each Collaboration processes a stream of messages, or *Events*, containing data or other information.

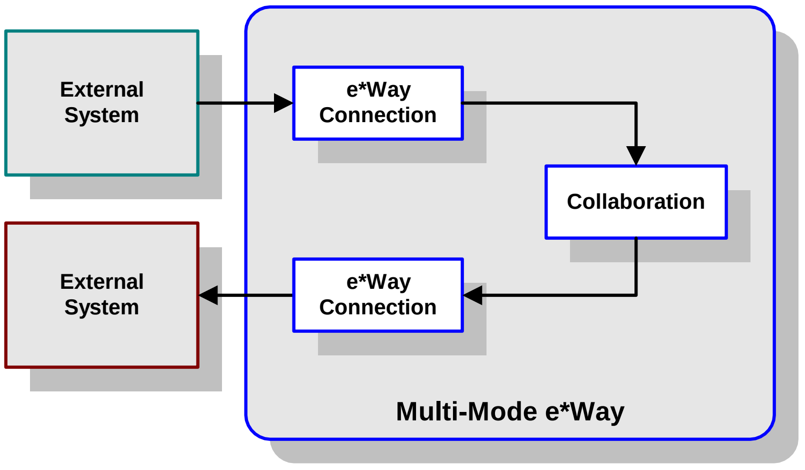
Each Collaboration that publishes its processed Events internally (within e*Gate Integrator) requires one or more *Intelligent Queues* (IQs) to receive the Events. See **Figure 3 on page 33**. Any Collaboration that publishes its processed Events only to an external system *does not* require an IQ to receive Events.

Figure 3 e*Way within e*Gate Integrator



Although usually implemented within e*Gate Integrator as shown in Figure 3, this e*Way can also be implemented as a stand-alone bridge between two or more external systems. See Figure 4.

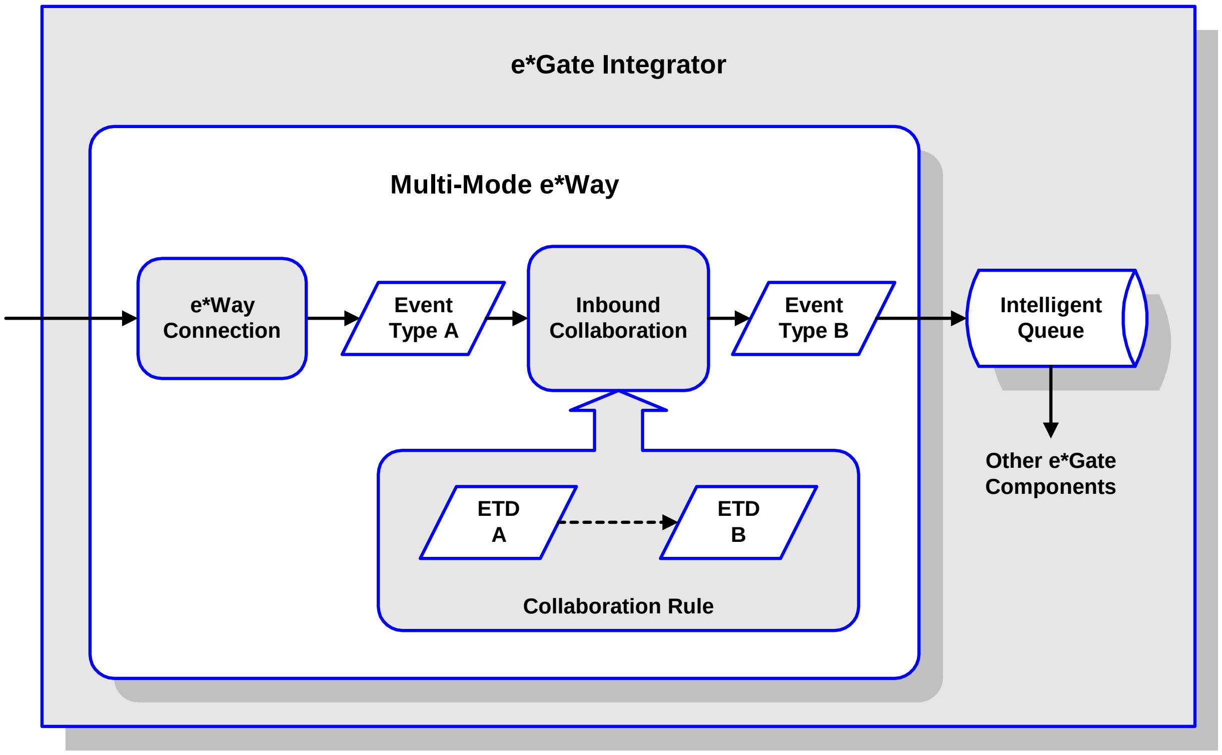
Figure 4 Stand-alone e*Way



3.3 Collaborations and Event Type Definitions

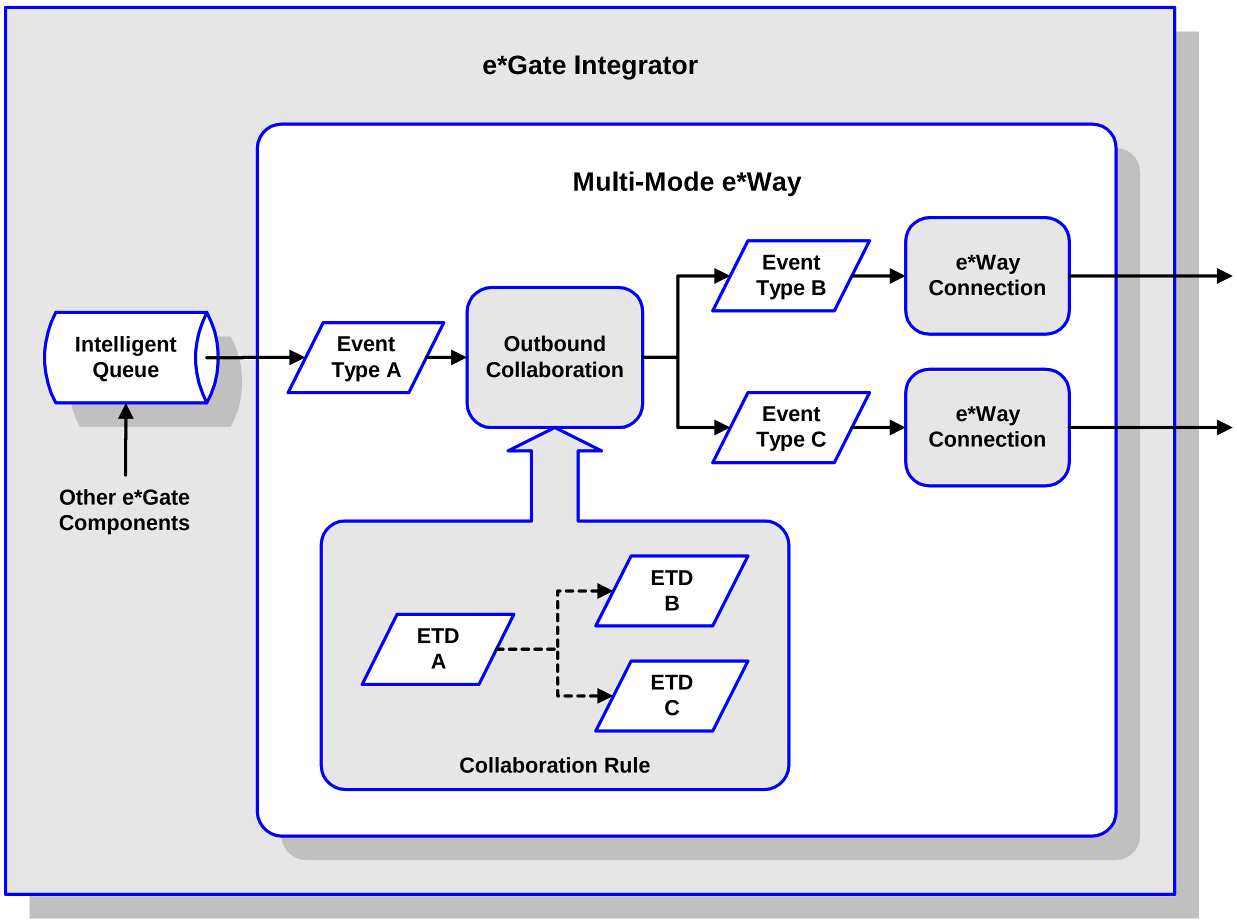
Collaborations execute the business logic that enables the e*Way to perform its intended task. Each Collaboration executes a specified *Collaboration Rule*, which contains the actual instructions to execute the business logic and specifies the applicable *Event Type Definitions* (ETDs). Events represent *instances* of their corresponding Event Types. Figure 5 shows a typical inbound Collaboration.

Figure 5 Inbound Collaboration



A corresponding look inside a typical outbound Collaboration is shown in Figure 6. In this diagram, two e*Way Connections are shown, feeding two external systems. More than two e*Way Connections are accommodated in each e*Way and, as stated previously, multiple Collaborations as well.

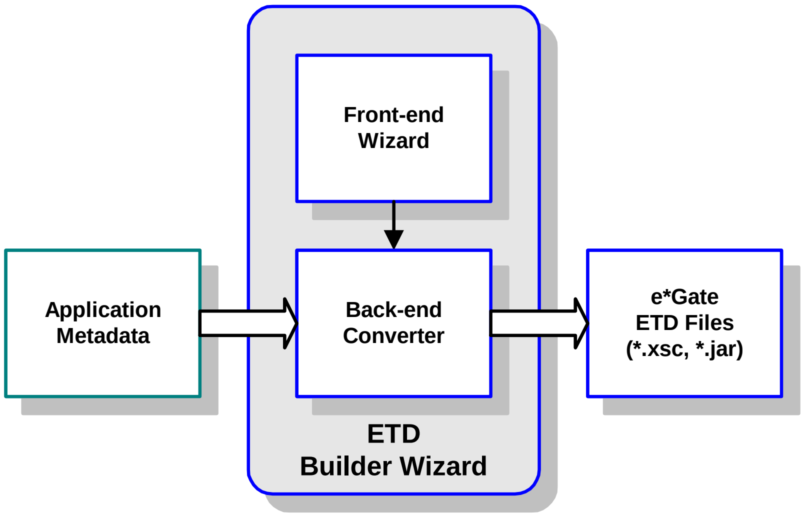
Figure 6 Outbound Collaboration



ETDs are representations of the data structure required by specific external systems. Transforming data from one format to another in accordance with the ETD is a major part of the processing performed by the e*Way. Building an ETD requires knowledge of the internal data structure of the specific application. This information is obtained by extracting metadata from the external application. (The importance of metadata is discussed in **"e*Way Design Considerations" on page 47**.)

In core e*Gate Integrator, the collection of metadata is automated by using an *ETD Builder*. A typical ETD Builder is shown in **Figure 7 on page 36**. The builder contains a wizard, accessed from within the ETD Editor GUI, which presents various options for your selection. A back-end converter then performs the prescribed metadata extraction. The resulting ETD is accessed in the ETD Editor.

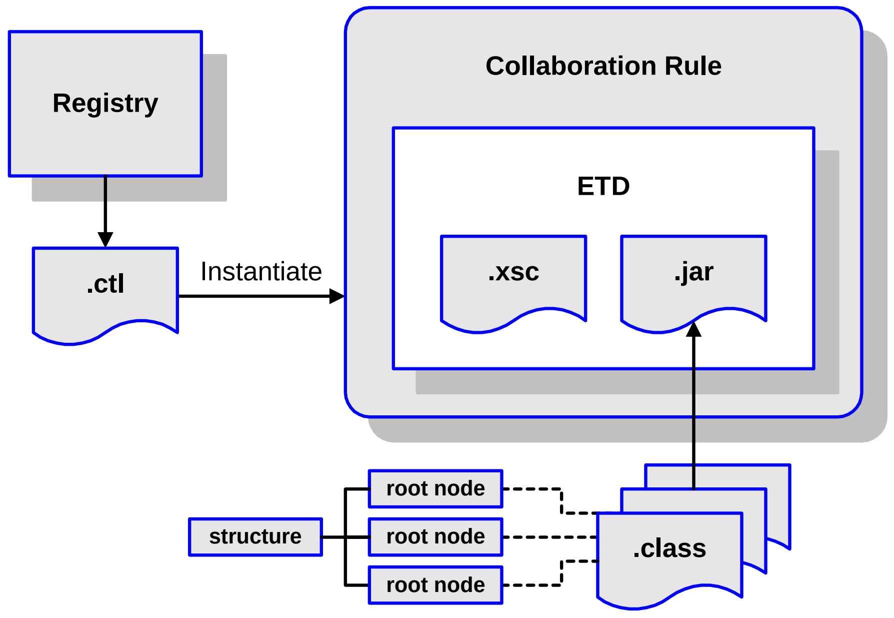
Figure 7 Typical ETD Builder



Version 4.5.2 or later of the Generic Multi-Mode e*Way Extension Kit User's Guideprovides an ETD Builder API. You can use this API either to create a command-line interface for building an ETD or to create an ETD Builder wizard that end users can access from within the ETD Editor. Details of the back-end (converter) and front end (builder wizard) are discussed in **Chapter 8 "Developing an e*Way Using ETD Builder Components" on page 89** and **Chapter 9 "Developing an e*Way Using the Builder API" on page 119**.

Once compiled, an ETD has two components, an **.xsc** file and a **.jar** file, both having the same file name. The **.jar** file contains **.class** files whose names correspond to the root node names in the ETD. Ultimately, the ETD is used within a Collaboration Rule to define the structure of the corresponding Event. At run time, the Collaboration Rule is instantiated according to information contained in a **.ctl** file contained in the e*Gate Registry (see Figure 8).

Figure 8 Event Type Definitions

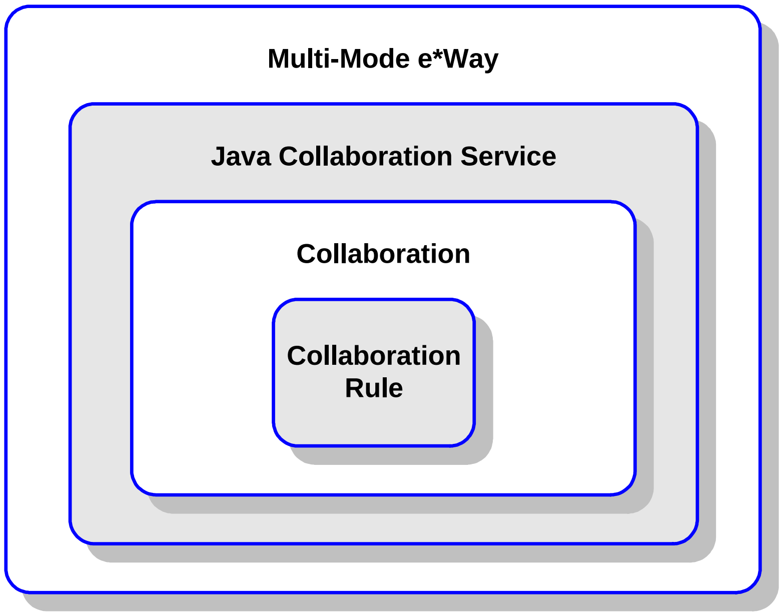


3.3.1 Java Collaboration Service

The Java Collaboration Service (JCS) provides an environment that allows you to use a Java class to implement the business logic that transforms Events as they move through e*Gate. When data passes through e*Gate by way of a Java Collaboration, a Java Virtual Machine (JVM) is instantiated and uses the associated Java Collaboration Rules class to accomplish the data transformation.

The relationships between the various Java e*Way components is depicted as a nested structure, as shown in Figure 9.

Figure 9 Java Component Relationships



A Multi-Mode e*Way serves as a container for one or more Collaborations whose Collaboration Rules use JCS. The Java Collaboration Service makes it possible to develop Collaboration Rules that execute e*Gate business logic using Java code. Using the Java Collaboration Editor, you create Java classes that implement the **executeBusinessRules()**, **userInitialize()**, and **userTerminate()** methods.

To use the Java Collaboration Service, you create a Collaboration Rule and select Java as the service. Using Event Type instances of previously defined ETDs, you then use the Java Collaboration Rules Editor to add the rules and logic between the Event Type instances. Compiling the Collaboration Rule creates a Java Collaboration Rules class and all required supporting files. This Java class implements the data transformation logic.

For more information on the Java Collaboration Service, see the *e*Gate Integrator Collaboration Services Reference Guide*.

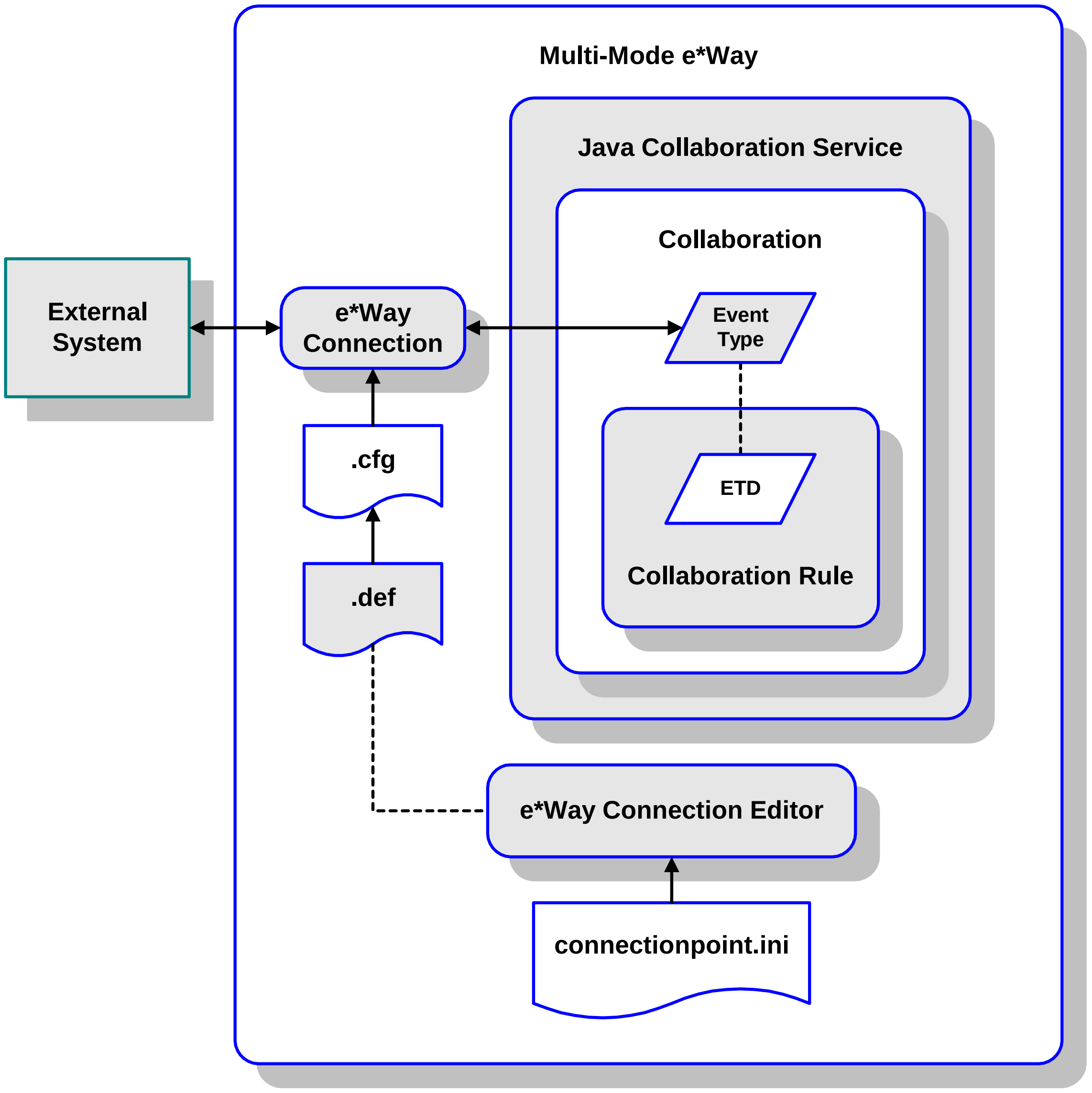
3.4 e*Way Connections

The e*Way Connections provide portals to external systems, allowing a single Multi-Mode e*Way to adopt several configuration profiles simultaneously. Individual e*Way Connections are configured using the e*Way Connection Editor to establish a particular kind of interaction with the external system.

3.4.1 Configuring e*Way Connections

An e*Way Connection to an external application is set up as shown in Figure 10. A **.def** file supplied with the e*Way is a template, configured for the specific application using the e*Way Connection Editor, and instantiated as a **.cfg** file for each e*Way Connection.

Figure 10 e*Way Connection Configuration



The e*Way Connection Editor allows you to modify all parameters of a Multi-Mode e*Way that control how the e*Way communicates with an external application. Because each e*Way functions in a specific way to provide an interface to a specific external application or communications protocol, each e*Way Connection has a unique set of configuration parameters.

The **connectionpoint.ini** file, stored in the **configs** directory, lists all e*Way Connections that are offered to the user by the GUI. For each e*Way Connection, it also specifies the directory for the associated **.def** file.

Chapter 4

Functional Flow

This chapter explains the run-time behavior of the Multi-Mode e*Way from startup to shutdown.

4.1 Overview of Run-Time Operation

Integration run-time execution involves various interfaces between the Multi-Mode e*Way (which houses user Collaborations) and the main e*Gate components (such as the e*Gate Registry, the IQ Managers with their associated IQs, and the JMS subsystem and associated JMS interfaces).

The basic functional flow of the Multi-Mode e*Way at run time is represented by a three-step process:

1. **Initialize**—When a Multi-Mode e*Way is started (normally, through the Control Broker) it goes through a two-phase initialization process that allows it to perform its configured Collaborations.

   - In the Environment phase, it obtains component information and downloads files from the e*Gate Registry, such as **.ctl**, **.jar**, **.exe**, and **.dll** files.
   - In the Setup phase, it starts the VM and then creates and starts separate threads to initialize each Collaboration.

   For complete details on this two-phase step, see **"Step 1: Initialize" on page 42**.

2. **Execute**—After the Multi-Mode e*Way and its Collaborations are initialized, as incoming Events trigger instantiation of e*Way Connection classes, the e*Way enters the Collaboration execution phase. During this phase, Collaborations process Events according to Collaboration Rules and ETDs.

   For complete details on this step, see **"Step 2: Execute" on page 43**.

3. **Shutdown/Reload**—If a Collaboration or e*Way Connection configuration is modified during execution, the e*Way must be reloaded. Both shutdown and reload trigger execution of the e*Way's termination task sequence. When the Multi-Mode e*Way is shut down, the appropriate cleanup must be performed.

   For complete details on this step, see **"Step 3: Shutdown/Reload" on page 45**.

4.2 Step 1: Initialize

The initialization step has two phases: the *Environment* phase and the *Setup* phase.

4.2.1 1A: Environment Phase

In the Environment phase, the e*Way obtains component information and downloads component files from the e*Gate Registry. This includes configuration information set through the GUI, such as the JVM settings, the Control Broker port, the Collaboration maps specifying publications and subscriptions, the Event Types used, and the corresponding **.jar** files, as well as any **.exe** files and **.dll** files that are needed, including those required by the IQ and Collaboration Services. It also downloads a **.ctl** file for each Collaboration.

About .ctl files

Each Collaboration Rule has an associated **.ctl** file, <*collabRuleName*>**.ctl**, called its *control file* (or, sometimes, its *initialization file*). The control file for a Collaboration Rule:

- Serves as a way to specify components that must be downloaded to the client side, where the e*Way will be executed.
- Specifies how the classpath for the Collaboration Rule is generated.

These **.ctl** files reside in the **collaboration_rules\** subdirectory. On the server side, this subdirectory is in the e*Gate Registry repository, of the <*schemaName*>**\runtime\** directory (and also in the <*schemaName*>**\sandbox\**<*userName*>\ directory if it exists). At run time, in the Environment phase of the initialization step, the items listed in the Collaboration Rule **.ctl** files are downloaded to the **collaboration_rules\** subdirectory on the client side, where they will be loaded in the e*Way's Startup phase.

Each e*Way ETD has its own associated **.ctl** file, named <*etdType*>**.ctl**, stored in the **etd\** subdirectory.

Note: *<etdType> must match the value of the* ***type*** *attribute for the <etd> entity in the corresponding* ***.xsc*** *file. See* **Chapter 7**.

When a **.ctl** file for a Collaboration Rule is created, the contents of the **.ctl** files for all of the ETDs it uses are appended.

4.2.2 1B: Startup Phase

The startup phase consists of:

1. Starting the JVM.
2. Creating a thread for each Collaboration.
3. Starting each thread. When started, each Collaboration thread then:
   A. Retrieves additional information from the e*Gate Registry.
   B. Loads the configuration.

C Initializes the Java Collaboration Service (JCS). After it is initialized, the JCS attaches to the JVM, initializes the e*Way Connections, and initializes the user Collaboration.

4.3 Step 2: Execute

1 Upon receiving the first Event (or, for inbound ETD instances, the first scheduled **get** interval), the Java Collaboration Service:

A Initializes the user Collaboration.

B Creates ETDs.

C Initializes ETDs.

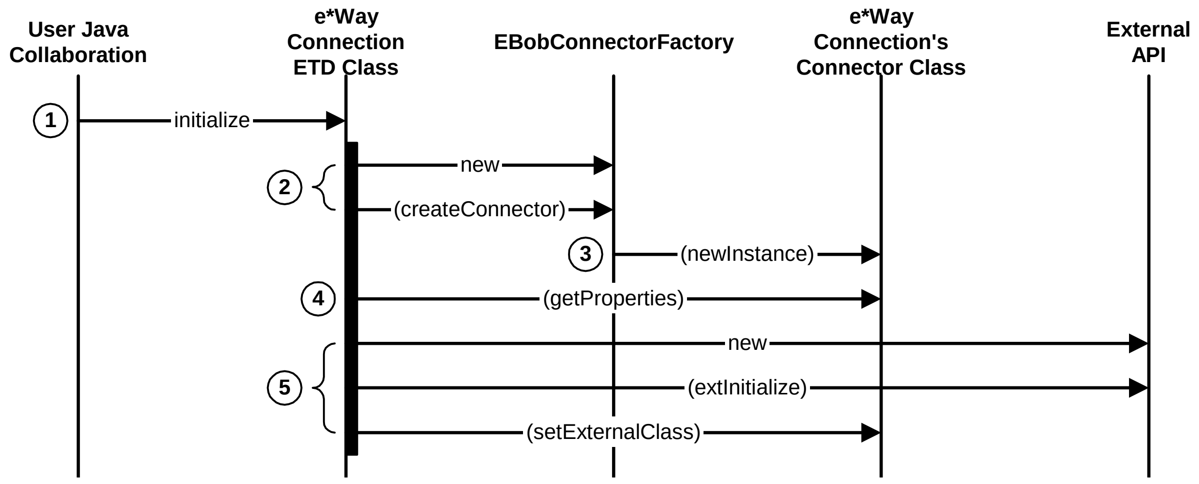
2 Starts the business process.

3 After processing each Event, calls the **reset()** method on each ETD (see **ETD Reset Flow** on page 44).

4.3.1 ETD Initialization Flow

Figure 11 gives an overview of the functional flow of the ETD initialization process.

Figure 11 Sequence of ETD Initialization Flow



Note: *The roles of the classes and methods mentioned in this section are explained in much greater detail in* **"e*Way Development Workflow" on page 51**.

1 The Collaboration instantiates and calls **initialize()** on the e*Way Connection's ETD class.

2 The ETD object's **initialize()** method instantiates the **EBobConnectorFactory**, and then calls **createConnector()**.

3 A new instance of the **Connector** class is created, and the configuration values from the e*Way Connection's **.cfg** file are loaded into a Properties object. The connector object is then returned to the ETD object.

4 The ETD object obtains configuration values by calling the **getProperties()** method on its connector object.

5 A typical implementation instantiates and initializes the external APIs.

4.3.2 Automatic Connection and Connection Management

As indicated in section 4.3 (**Step 2: Execute** on page 43), the instantiation of e*Way Connection classes occurs when the first inbound Event is received by the associated Collaboration.

The connection to the external system associated with the e*Way Connection is established either through Automatic Connection or through Connection Management. Table 4 shows two e*Way Connection control techniques.

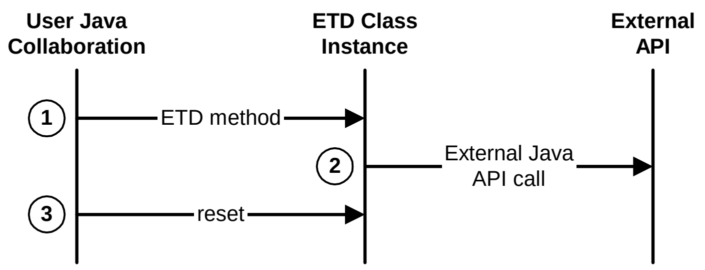
Table 4 e*Way Connection Control Techniques

Automatic Connection Mode	Connection Management Mode
▪ available in e*Gate 4.5.1 or later	▪ available in e*Gate version 4.5.2 or later
▪ establishes connection automatically	▪ registers with a Connection Manager

For more information on e*Way Connection Control Techniques, see the *e*Gate Integrator User's Guide*.

4.3.3 ETD Reset Flow

Figure 12 Sequence of ETD Reset Flow



1 A **rule** statement in the Collaboration is executed, causing the e*Way Connection's ETD method to be called.

2 A call is made to the external system's APIs.

3 The **reset()** method is called after all **rule** statements in the Collaboration have been executed.

4.4 Step 3: Shutdown/Reload

Upon shutdown or reload, the Java Collaboration Service:

1 Calls **terminate()** on each ETD.

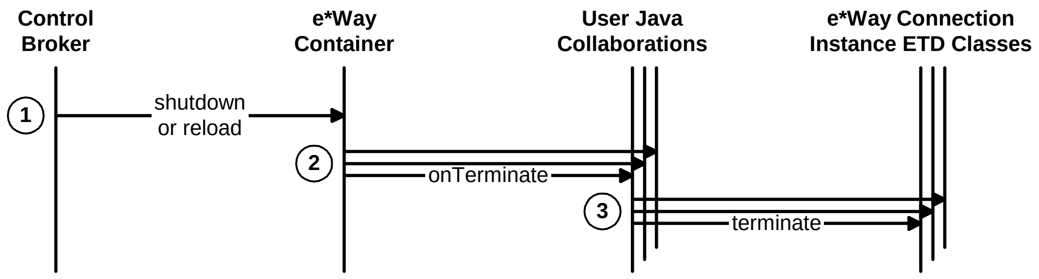
2 Calls user terminate on each User Collaboration.

3 Releases all internal resources.

4 Detaches from the JVM.

4.4.1 Shutdown/Reload Flow

Figure 13 Sequence of Shutdown/Reload Flow



1 Shutdown or reload is triggered by the Control Broker.

2 The e*Way container complies by calling **onTerminate()** on all its Collaborations.

3 The **terminate()** method is called on each e*Way Connection ETD used in the Collaboration.

Note: *Special considerations apply to the **terminate()** method in the case of Subcollaboration Rules; see "Implications for e*Way Development" on page 55.*

Chapter 5

e*Way Development Methodology

This chapter provides considerations for the development and design of an e*Way.

5.1 e*Way Development Considerations

The following areas are critical to the development of an e*Way:

- Careful planning of both the e*Gate components and the project itself.
- Gathering and validating requirements and creating a library of use cases.
- Designing the code, including:
  - Connectivity and data factors—for example, needing to connect to servers using a variety of open or propriety protocols and data formats.
  - Opportunities to re-use existing code or components.
  - Facilities offered by the external system or systems for collecting metadata.
  - Communication modes and integration interfaces.
  - Open or third-party–specific Application Programming Interfaces (APIs).
- Implementing your design and validating the implementation.
- Building and testing your components.
- Packaging and distributing your deliverables.

5.2 e*Way Design Planning

At a minimum, your planning process should include the following:

- Identify what will be required in the integration solution. For example:
  - e*Gate components, including existing e*Ways.
  - Platforms to be supported, such as operating systems and versions.
  - Underlying third-party systems.
  - Development and deployment tools.

5.3 e*Way Design Considerations

The purpose of an e*Way will dictate the type of e*Way you design. You can leverage existing e*Ways or ETD libraries to create your e*Way. Many of the off-the-shelf e*Ways—especially those in the protocol-wrapper, database, and Web interface categories—are highly generic and self-contained, and are used as the building blocks of the new e*Way.

Table 5 shows examples of the types of e*Ways that are currently available.

Table 5 e*Way Categories

Type of e*Way	Description and Examples
Protocol-wrapper e*Ways	e*Ways that implement standard Internet protocols. Examples include TCP/IP, FTP, SMTP (e-mail), HTTP(S), Dial-up (Kermit, z-modem), and LDAP.
Database e*Ways	Examples include Oracle, Sybase, UDB/DB2, ODBC, and JDBC.
Web interface e*Ways	Examples include CGI, MSIIS, ISAPI, and SOAP.
Enterprise application e*Ways	Some enterprise application types are: ▪ *Customer Resource Management (CRM)*. Examples include Siebel, Clarify, and Vantive. ▪ *Enterprise Resource Planning (ERP)*. Examples include SAP and PeopleSoft. ▪ *Billing systems*. Examples include Kenan and Portal ▪ *Groupware*. Examples include Lotus Notes.

You can also take advantage of the many pre-packaged e*Gate ETD libraries, such as CIDX, cXML, HIPAA, UN/EDIFACT, RosettaNet, X12, and xCBL.

Additional e*Way design considerations include:

- Establishing connectivity protocols
- Metadata collection
- Communication and integration
- Third-Party System APIs

These are described in the following sections.

5.3.1 Establishing Connectivity Protocols and Defining Event Types

Most third-party systems operate on a client/server or multi-tiered architecture. When developing e*Ways, you may be required to connect to the appropriate servers using one or combinations of the following protocols:

- Application-specific proprietary protocols, such as MQSeries or COM/DCOM.
- Industry-specific open standard protocols, such as SOAP, JDBC, or JMS.
- Common Internet protocols, such as TCP/IP, FTP, HTTP, CGI, or SMTP.

These protocols may include data format provisions. These must be considered when you define your custom ETDs.

Many enterprise applications or third-party systems use data structured in proprietary data formats. To be able to work with these data, they must be represented in e*Gate through ETDs, as discussed in **Chapter 7** “Event Type Definitions” on page 79.

For complex ETDs, you should consider the use of internal and/or external ETD *templates*:

- If your ETD has different areas with identical subtree definitions, you can decrease effort and increase maintainability by using *internal* templates instead of re-creating the subtree for each new place it is used. Internal templates are local to the ETD.
- You can use *external* templates to allow your ETD to reference a frozen copy of another ETD of the same type. An external template is an ETD you can reuse to duplicate a particular structure in other ETDs. The modularity of this approach lends itself to team development, especially of multiple interrelated ETDs.

5.3.2 Facilities for Collecting Metadata

Metadata is generic data that describes, characterizes, or qualifies the Events that travel through the system. One of the most important considerations in creating e*Ways is to determine whether the third-party system, or a knowledgeable vendor, provides tools for gathering metadata. Such tools greatly facilitate the design of ETDs.

Examples

Two specific examples (illustrating the wide variety of external systems that may be encountered) are SAP and Jacada:

- SAP, an ERP architecture, provides facilities that are queried for a set of business objects available in the system. It provides an API that allows you to find out what operations are available for these objects and what the input and output data look like for these operations.
- Jacada, a “screen-scraper” for interfacing with legacy mainframe systems, provides APIs that are utilized to query for a set of services and their associated methods that are easily mapped to ETDs.

In cases like SAP and Jacada, the use of tools specifically provided for interfacing with the external system diminishes ambiguity and reduces the need for special-purpose research and requirements analysis. This allows you to create well-defined integration scenarios.

XML

The e*Gate *XML Toolkit* contains e*Gate-related information on builders, converters, and so on. It includes the e*Gate Registry API for XML Schema metadata. This is useful for any external system that uses XML based on pre-packaged Document Type Definitions (DTDs), XML Schema Documents (XSDs), and Extensible Stylesheet Language Transformations (XSLTs).

In systems that make use of XML for data exchange, metadata can usually be obtained through a vendor-specific or standards-based library of DTDs (or the equivalent, such as xCBL for Commerce One), or by tools used to generate these DTDs. It can often be worthwhile to develop or enhance these tools yourself.

5.3.3 Communication Modes and Integration Interfaces

In addition to the protocols used, you must consider the semantics of the communication interface during data exchange. Whether for outbound delivery or inbound delivery, you must determine strategies to address the quality of service and responsiveness of the data exchange.

Strategies for outbound delivery

The following questions must be addressed:

- Is synchronous data transfer required? (In other words, will the client be blocked during the call?) If so, is it best accomplished through simple Request/Reply, or through remote procedure calls such as RPC or Java RMI?
- Is asynchronous data transfer required? If so, can it be accomplished through a “fire-and-forget” strategy, or is a response required for some (or all) data that is sent?
- Is polling required? (For example, the Batch and FTP e*Ways provide data transfer based on a time schedule.)
- Is JMS applicable? (For a further discussion of JMS, see **“Oracle SeeBeyond JMS”**.)

Strategies for inbound delivery

The recommended mechanism for delivering Events (messages) inbound to e*Gate is the Oracle SeeBeyond implementation of Java Message Service (JMS).
When considering how or whether to implement JMS, the following questions must be addressed:

- Is connection pooling support required?
- Where should JMS client API calls be placed for all the components involved?
- What JMS programming models should be employed?

Oracle SeeBeyond JMS

Oracle SeeBeyond JMS supports both persistent and non-persistent delivery modes:

- *Persistent* mode guarantees delivery; if a failure occurs for any reason (such as a disconnect or a transient load problem), the system continues to attempt redelivery.
- *Non-persistent* mode has lower overhead, but does not guarantee delivery.

The Oracle SeeBeyond implementation of JMS also supports both of the messaging models defined in the JMS specification:

- In the *publish-and-subscribe* (pub/sub) model, one producer can broadcast a message to many consumers on a virtual channel called a *topic*. Publish-and-subscribe is an excellent mechanism for many-to-many conversations, with consumers registering their interest in certain messages by subscribing to a topic. Because this model is push-based, consumers do not have to request or poll the topic for new messages.

- In the *point-to-point* (p2p) model, JMS clients can send and receive messages through virtual channels called *queues*. A message from a queue is consumed when it is first received; thus, even when a queue has many receivers, each message is received only once. This model is desirable for one-on-one conversations and for messages that need to be processed separately, serially, or both.

In either model, messages are transmitted synchronously or asynchronously. Both models support *request-reply* messaging, where a client expects to receive a response as a result of a sent message.

e*Gate version 4.5.2 JMS includes the *message selector* feature, which supports the following additional capabilities:

- In the *publish-and-subscribe* (pub/sub) model, when a message needs to be distributed to many clients, message header data visible to the JMS provider can allow the provider to handle much of the filtering and routing, without impacting each client application.
- In the *point-to-point* (p2p) model, if a message is sent to a single receiver and the criteria of filtering and categorization are included in the message, the receiving client can discard any message that is not required.

5.3.4 APIs for Third-Party Systems

Third-party systems usually provide Application Programming Interfaces (APIs) that expose native functions. These APIs can generally be called using any or all of the following: C/C++, Java, COM interfaces, and CORBA interfaces.

Wrapping these APIs and exposing the relevant methods needed for integration with other systems is a common approach in the development of e*Ways. Java-based e*Ways are implemented as thin wrappers around these third-party APIs through the appropriate interfaces.

“Developing an e*Way Using the Builder API” on page 119, guides you through the process of creating an ETD whose API calls are simulated by corresponding stubs in an RMI (Remote Method Invocation) server.

Chapter 6

e*Way Development Workflow

This chapter provides high-level information for creating an e*Way, including custom e*Way Connections and ETDs. The following are described in this chapter:

- Usage of the Java classes in e*Way Connection ETDs
- Subcollaboration rules
- e*Way development execution, including:
  - Design implementation
  - Building and testing components
  - Packaging and distribution

6.1 Java Classes Used in e*Way Connection ETDs

Each **.xsc** file must have a corresponding **.jar** file that contains all associated Java classes. The **.xsc** file exposes methods for accessing attributes, as well as exposed methods in third-party wrapped classes, so that end users can view them through the e*Gate GUI.

Before developing your Multi-Mode e*Way, you should thoroughly review the third-party Application Programming Interfaces (APIs) that are available to you. The API's tell you what objects are available and how input and output data look. The functionality of the e*Way you create is determined by how you wrap these APIs and which methods you expose.

6.1.1 Class Relationships

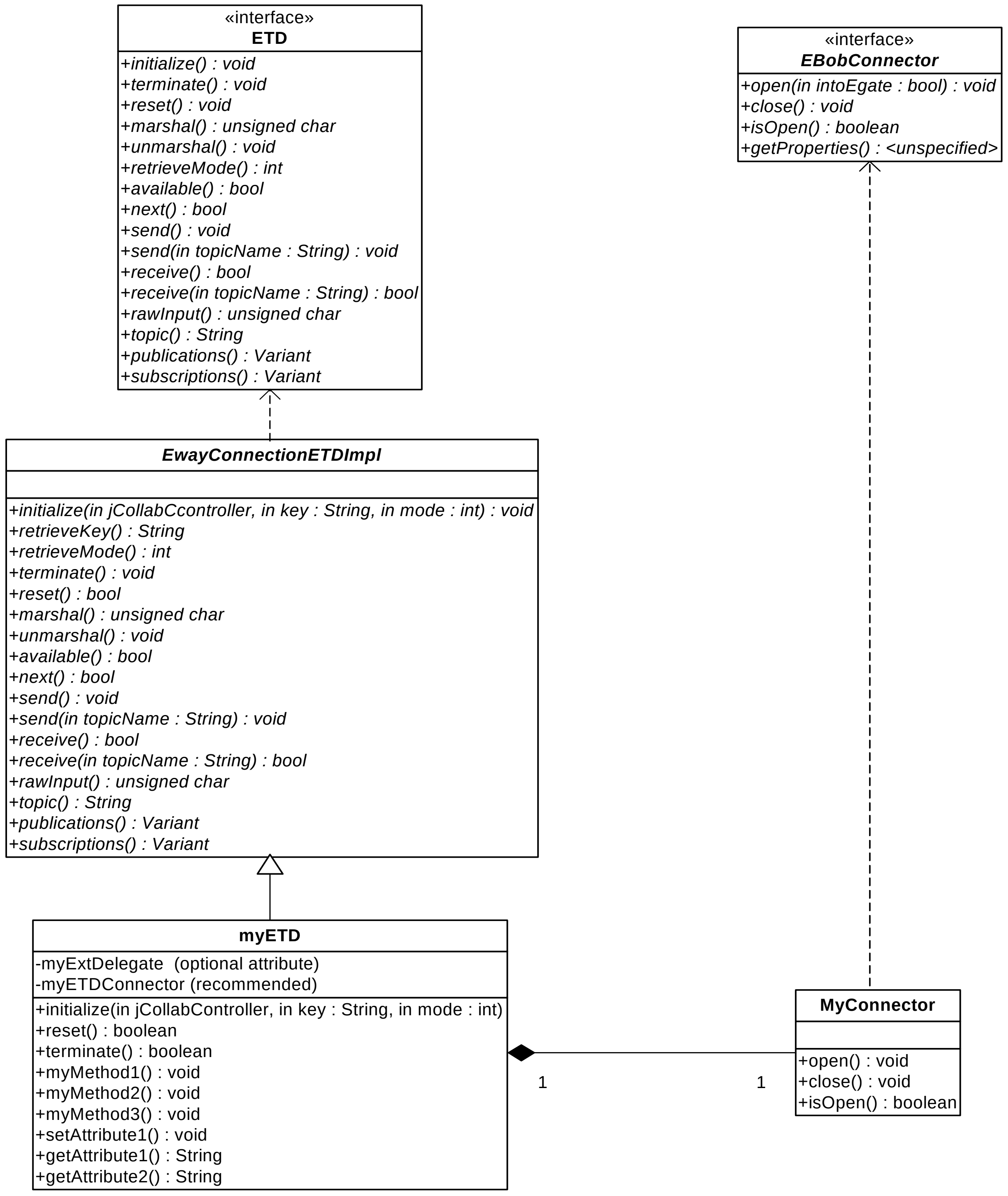
A custom Multi-Mode e*Way must contain classes that implement two key interfaces:

- Its ETD class must extend the supplied abstract class *EwayConnectionETDImpl*, which implements the interface **ETD**.
- Its connection class must implement the interface **EBobConnector**.

For complete information on the ETD and EBobConnector interfaces, refer to the Javadoc in **"e*Way Classes and Methods" on page 254**.

A class diagram for the ETD and connector classes you create is shown in Figure 14.

Figure 14 Class Relationships of the ETD and Connector Classes


«interface»
ETD
+initialize() : void
+terminate() : void
+reset() : void
+marshal() : unsigned char
+unmarshal() : void
+retrieveMode() : int
+available() : bool
+next() : bool
+send() : void
+send(in topicName : String) : void
+receive() : bool
+receive(in topicName : String) : bool
+rawInput() : unsigned char
+topic() : String
+publications() : Variant
+subscriptions() : Variant
«interface»
EBobConnector
+open(in intoEgate : bool) : void
+close() : void
+isOpen() : boolean
+getProperties() : <unspecified>
EwayConnectionETDImpl
+initialize(in jCollabCcontroller, in key : String, in mode : int) : void
+retrieveKey() : String
+retrieveMode() : int
+terminate() : void
+reset() : bool
+marshal() : unsigned char
+unmarshal() : void
+available() : bool
+next() : bool
+send() : void
+send(in topicName : String) : void
+receive() : bool
+receive(in topicName : String) : bool
+rawInput() : unsigned char
+topic() : String
+publications() : Variant
+subscriptions() : Variant
myETD
-myExtDelegate (optional attribute)
-myETDConnector (recommended)
+initialize(in jCollabController, in key : String, in mode : int)
+reset() : boolean
+terminate() : boolean
+myMethod1() : void
+myMethod2() : void
+myMethod3() : void
+setAttribute1() : void
+getAttribute1() : String
+getAttribute2() : String
1
1
MyConnector
+open() : void
+close() : void
+isOpen() : boolean

Specific examples are provided with the kit:

- In the sample that illustrates Automatic Connection, **Figure 47 on page 158** shows the connector class (**SampleETDConnector**) implements the *EBobConnector* interface, and **Figure 46 on page 157** shows how the ETD class (**SampleETD**) extends the **EwayConnectionETDImpl** abstract class.

- In the sample that illustrates Connection Management, **Figure 52 on page 189** shows how the connector class (**EBobConnectorExtImpl**) extends the abstract class **EBobConnectorExt**, and **Figure 51 on page 187** shows how the ETD class (**TcpClientETD**) implements the *ETDExt* interface.
- In the sample that illustrates Transactional functionality, the connector class (**XAFileETDConnector**) is the same as for the Automatic Connection sample. **Figure 56 on page 226** shows how the ETD class (**XAFileETD**) implements the *ETDExt* interface as well as the *JXAResourceAdapter* interface.

6.1.2 Application Programming Interfaces (APIs)

For an Automatic e*Way Connection, the custom APIs consist of the following:

- **EBobConnectorFactory**—A factory class that instantiates the connector class you develop (by means of its **createConnector()** method) and loads your configuration.
- **JCollabController**—An instance of this class is associated with each Collaboration; the **this.jCollabController** object holds references to the various objects needed by the Collaboration.
- **CollabConnException**, **CollabDataException**, and **CollabResendException**—Classes for handling exceptions that are thrown inside either of the classes you develop (ETD or connector); these exceptions are caught in the end user’s Collaboration.
- **EBobConnectionException**—Class for handling exceptions that are thrown inside the connector class you develop.
- **ETD**—The interface that must be implemented by the ETD class you develop.
- **EBobConnector**—The interface that must be implemented by the connector class you develop.

For a Connection-Managed e*Way Connection, the custom APIs consist of the following:

- **EBobConnectorExt**—For Connection Management e*Ways, interface that must be implemented by the connector class you develop (instead of **EBobConnector**).
- **EBobConnectorExtFactory**—For Connection Management e*Ways, this the factory class corresponding to **EBobConnectorFactory**.
- **JConnectionManager**—Specifies the interface implemented by the Connection Manager. This interface provides the methods called by ETDs that need to support Connection Management.
- **JConnectionNotifier**— implemented by user Collaborations to specify actions/ rules (in the **onConnectionUp()** or **onConnectionDown()** methods, enabled by Connection State Trapping) that are called based on the state of a connection.

Note: *Implementing* ***JConnectionNotifier*** *only applies to ETDs that utilize the* ***Automatic*** *connection establishment mode.*

For a Transactional e*Way Connection, the custom APIs consist of the following:

- **JTransactionAdapter**—Specifies the interface implemented by the ETDs that support one-phase transactions (non-XA). It provides the methods called by the Transaction Manager to commit or roll back non-XA transactions.
- **JXAResourceAdapter**—Specifies the interface implemented by the XA Resource Managers. It provides the methods called by the Transaction Coordinator component of the Transaction Manager on XA ETDs.

For details, see the Javadoc files supplied on the e*Gate Integrator Installation CD-ROM.

Note: *APIs for Standard (**.ssc** file-based) ETDs are described in the **e*Gate User's Guide**. **.ssc** file based ETDs are not Java-enabled. For details on **.ssc** based entities, see* **"Delimiter-Related Entities (SSC only)" on page 295**.

6.2 Subcollaboration Rules

A Collaboration Rule are used as a parent or a child of another Collaboration Rule. This allows you to insulate connectivity from transformation by separating out transformation-specific Collaboration Rules and maintaining them as individual units.

Every Collaboration Rule is either:

- Used as a *Root Collaboration Rule*—in other words, a Collaboration Rule invoked by e*Gate itself; or
- Used as a *Subcollaboration Rule*, invoked by a parent Collaboration Rule.

A Collaboration Rule, when used as a Root Collaboration Rule, is like a main program; when used as a Subcollaboration Rule, it is like a subroutine. For example:

- A Subcollaboration Rule allows you to reuse a valuable piece of work in another context without having to reinvent it or reconstruct it from scratch.
- Typically, a Subcollaboration Rule takes care of details or special-purpose parsings and transformations, allowing the parent Collaboration Rule to be simpler and more general.

Every Collaboration Rule runs in a *mapping environment* defined by its container:

- A Root Collaboration Rule's mapping environment is defined through the e*Gate GUI—namely, the **Collaboration Properties** dialog box.
- A Subcollaboration Rule's mapping environment is defined programmatically, through its parent's call to **setInstanceMap()**.

For more information on Subcollaboration Rules, especially the **JSubCollabMapInfo** object and the **JCollabController.createSubCollabMapInfo()** and **setInstanceMap()** methods, refer to the material in the *e*Gate Integrator User's Guide*.

6.2.1 Caveats

The following caveats apply to Collaboration Rules invoked as Subcollaboration Rules:

- Two-phase transaction processing—that is, Prepare/Commit/Rollback—can only be handled at the Root Collaboration Rule level, never by a Subcollaboration Rule.
- A Collaboration Rule that uses ELS are invoked as a Subcollaboration Rule, but its **executeBusinessRules()** code runs immediately, bypassing its ELS-specific code.
- Collaboration Rules that use an API ETD—other than database ETDs—are ineligible for being used as Subcollaboration Rules.
- When an outbound Event Type instance is set to **Manual Publish**, its data is handled by its ETD's **send()** method (or not at all), and cannot be intercepted by its container. In other words, for a manually published Event Type instance, its data goes wherever **send()** sends it—typically to an IQ or to a JMS e*Way Connection.
- When an ETD is being used in a Subcollaboration Rule, you must override the **terminate()** method of your ETD class. For details, see **"Implications for e*Way Development"**.

6.2.2 Implications for e*Way Development

Subcollaboration Rules have the following implications for e*Way development:

- *What to do:* Override the **terminate()** method of your ETD class.

  Why: The **terminate()** method needs to check whether it is being called from within a Subcollaboration Rule. For details, see the **isSubCollaboration()** method in the Javadocs in **"e*Way Classes and Methods" on page 254**.

  When an ETD is being used in a Subcollaboration Rule, the Connector object returned by the **createConnectorExt()** method is the same instance used by the Root Collaboration Rule. The **isInSubcollab** boolean flag is set here, to be used later by the **terminate()** method to determine whether or not to close the connection. If the connection *is* in a Subcollaboration Rule, it should *not* close external connections at this time. Instead, it must release resources used in the Subcollaboration Rule without closing the external connection. Later, the connection will be closed in the Root Collaboration Rule.

- *What to do:* Make sure your e*Way Connection ETD implements the additional methods **setConnector()** and **getConnector()**.

  Why: To support Subcollaboration Rules, your e*Way ETDs must be able to share connectors. The **set|getConnector()** methods are called during initialization of your ETD based on whether the current Collaboration Rule is a Subcollaboration or a Root Collaboration Rule.

6.3 Implementing Your Design

Implementation of an e*Way Connection ETD class involves the following steps:

1. Create the **.java** files.
2. Create a **compile.bat** (or, on UNIX, **compile.sh**) file that reflect your development environment, and compile the **.java** files to create **.jar** files.
3. Create **.ctl** files that reflect your CLASSPATH environment.
4. Create **.def** files for your e*Way Connections' default configuration parameters.
5. Create the **.xsc** files, using either an XML editor or a text editor such as Notepad, so that your ETDs' methods and properties are graphically exposed to the end user through the ETD Editor and the Collaboration Editor.

6.3.1 Creating .java Files

Sample files have been provided and require only minor edits to enable the sample schemas to be implemented; **MySchema.zip** contains the files for a simple Automatic Connection e*Way, and **TcpEcho.zip** contains the files for a more complex Connection Management e*Way.

This section provides the basic information necessary to create code, compile it, and commit your files to the e*Gate Registry, including the following:

- Class development and configuration instructions
- Information and instructions for incorporating support for e*Way Connections with Connection management
- Information and instructions for incorporating support for XA-enabled e*Way Connections.

Class Development and Configuration

The following are instructions for creating and configuring the classes that will be stored in the **.java** file.

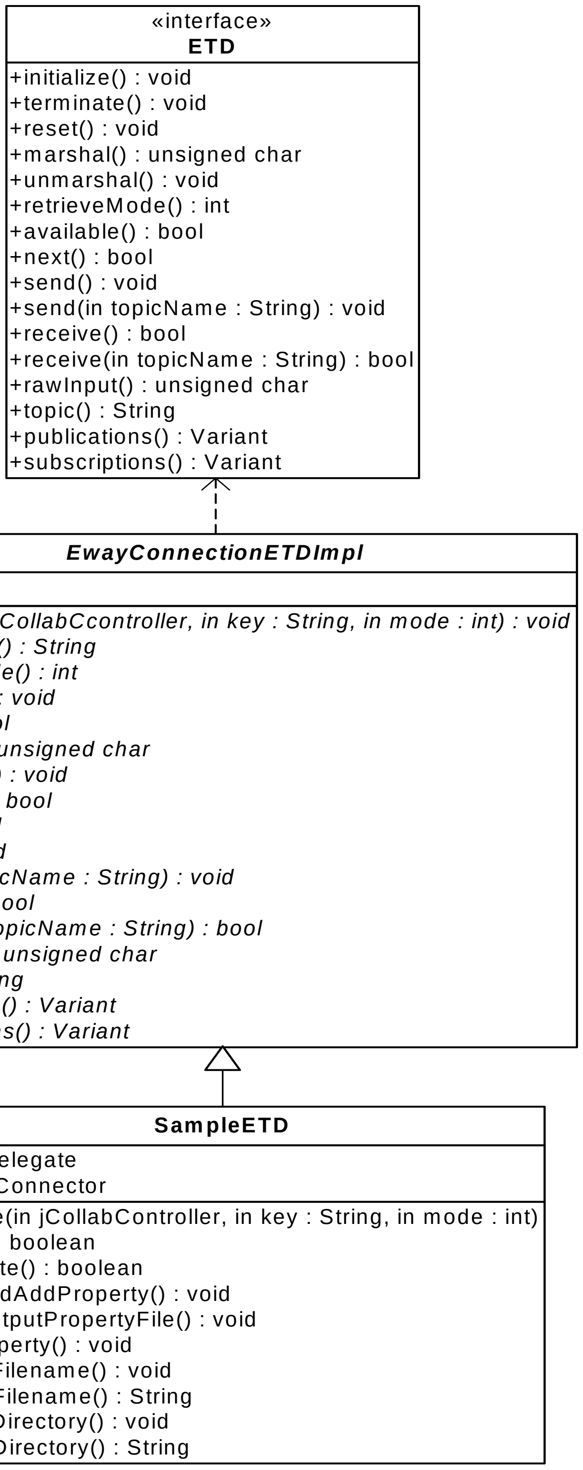
1. *What to do:* Create your ETD class, using the class relationships shown in either:
   - **Figure 15 on page 57**, for a 4.5.1 e*Way that does not use Connection Management; or
   - **Figure 17 on page 62**, for an e*Way that does use Connection Management. (Additional steps are required; details of these steps are provided in **"e*Way Connections with Connection Management" on page 60**.)

Why: Java Collaborations operate based on instances of inbound and outbound ETD classes. ETD classes are classes that implement the Java interface **com.stc.jcsre.*ETD***.

Note: *For details on the ETD interface, see the Javadoc files supplied on the e*Gate Integrator Installation CD-ROM.*

Collaborations expect to operate on objects that implement the *ETD* interface. The *ETD* interface includes the methods involved in obtaining data from or putting them in e*Gate queues (such as IQs or JMS). It also includes the **initialize()**, **reset()**, and **terminate()** methods that are called as part of the standard Collaboration life cycle.

Figure 15 Class Relationships of an ETD Class


«interface»
ETD
+initialize() : void
+terminate() : void
+reset() : void
+marshal() : unsigned char
+unmarshal() : void
+retrieveMode() : int
+available() : bool
+next() : bool
+send() : void
+send(in topicName : String) : void
+receive() : bool
+receive(in topicName : String) : bool
+rawInput() : unsigned char
+topic() : String
+publications() : Variant
+subscriptions() : Variant
EwayConnectionETDImpl
+initialize(in jCollabCcontroller, in key : String, in mode : int) : void
+retrieveKey() : String
+retrieveMode() : int
+terminate() : void
+reset() : bool
+marshal() : unsigned char
+unmarshal() : void
+available() : bool
+next() : bool
+send() : void
+send(in topicName : String) : void
+receive() : bool
+receive(in topicName : String) : bool
+rawInput() : unsigned char
+topic() : String
+publications() : Variant
+subscriptions() : Variant
SampleETD
-myExtDelegate
-myETDConnector
+initialize(in jCollabController, in key : String, in mode : int)
+reset() : boolean
+terminate() : boolean
+copyAndAddProperty() : void
+saveOutputPropertyFile() : void
+addProperty() : void
+setOutFilename() : void
+getOutFilename() : String
+setOutDirectory() : void
+getOutDirectory() : String

An e*Way Connection has an associated class that implements the **ETD** interface. We refer to this class as your ETD class.

The class **EwayConnectionETDImpl** is a sample provided to implement the **ETD** interface. It is an abstract class that your ETD class must extend, and it provides the default implementation used for e*Way Connection (non-messageable) ETDs. The **EwayConnectionETDImpl** class should not normally be modified; any additional desired functionality should be included in your ETD class.

By extending **EwayConnectionETDImpl**, your ETD class inherits common behavior when interacting with e*Gate IQs and JMS.

Note: *Since it is not intended for message parsing, **EwayConnectionETDImpl** contains empty implementations of the **marshal()** and **unmarshal()** methods. For more information on parsing messageable ETDs, see* **"Handling Messageable ETDs" on page 243**.

We refer to the class that extends **EwayConnectionETDImpl** as your *e*Way Connection ETD class*. This is the class that exposes your ETD's functionality; the methods generated by the Collaboration Rules Editor correspond to methods in your ETD.

2 *What to do:* Create a delegate object in your ETD class.

Why: Delegation is the use of a separate class encapsulating specific functionality In this case, the functionality being encapsulated consists of the calls dealing with the external system or entity. A reference to the delegate object (an instance of the **Delegate** class) is kept in the ETD class. Delegation allows for cleaner separation of functions dealing with an external system; for example, you can wrap third-party API calls in a delegate class. This is a useful technique for developing most ETDs where the API calls are isolated and wrapped easily.

3 *What to do:* Override the **initialize()** method of your ETD class.

Why: If you support Connection Management or transaction processing, you need to call some registration methods. The initialization procedure performed here includes obtaining the associated connector for the ETD using **EBobConnectorFactory** or **EBobConnectorExtFactory**. Once the connector is available, the ETD has access to configuration information. Any initialization that needs to be done on the external system through third-party APIs through the **Delegate** class may be done here as well.

4 *What to do:* Instantiate your connector class, using the class relationships shown in either:

- **Figure 16 on page 59**, for a 4.5.1 e*Way that does not use Connection Management; or
- **Figure 17 on page 62**, for an e*Way that uses Connection Management. (Additional steps are required; details of these steps are provided in **"e*Way Connections with Connection Management" on page 60**.)

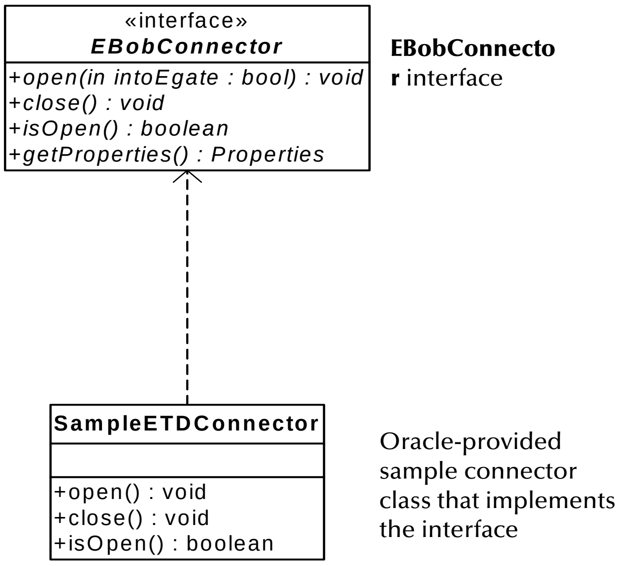
Why: The implementation of your e*Way Connection's configuration and connection management functions must be provided in a class that extends **EBobConnector**; this class is referred to as the *connector class* for your ETD.

The *EBobConnector* interface is used encapsulate the connection methods **open()**, **close()**, and **isOpen()**, as well as the **getProperties()** method, which allows access to configuration values as Java Properties.

In e*Gate version 4.5.2, the *EBobConnector* interface was extended by the *EBobConnectorExt* interface to support Connection Management functionality.

Note: *For details on all interfaces and methods, see the Javadoc files supplied on the e*Gate Integrator Installation CD-ROM.*

Figure 16 SampleETDConnector (Connector Class) Relationship



5 *What to do:* Override your ETD class's **reset()** method so that it returns **true** if you do not want the ETD instance to be re-instantiated for each new incoming Event.

Why: This provides an opportunity to clear any resources or cached information.

6 *What to do:* Override your ETD class's **terminate()** method.

Why: The **terminate()** method is called when the Multi-Mode e*Way shuts down, terminating its configured Collaboration Rules. The Collaboration Controller will need to call **terminate()** on ETDs in Subcollaboration Rules as well before returning control to the parent Collaboration Rule.

7 *What to do:* Create your ETD's connector class; this is the class that implements the **EBobConnector** or **EBobConnectorExt** class.

Example: The abstract class **EwayConnETDConnectorExtImpl** in the TCP Client sample provides a default implementation of the EBobConnectorExt interface. You can use this class as boilerplate code and extend it for your connector class. For a class diagram showing how this works, see **Figure 51 on page 187**.

e*Way Connections with Connection Management

In e*Gate version 4.5.2, Connection Management offers the following connection establishment options:

- **Automatic**—The connection to the external system is established automatically. If the connection is lost, connection is re-established.

  In Automatic connection establishment mode, user Collaborations are notified of external connections being established and disestablished. The end user can take advantage of these notifications in the Collaboration Rules Editor through the option **Enable Connection State Trapping** on the **File** menu. Enabling this option indicates that the user Collaboration class will implement the JConnectionNotifier interface, which contains an **onConnectionUp()** and an **onConnectionDown()** method. The user can specify rules to be executed in these methods. For example, the user might choose to insert a rule that sends an Alert into the **onConnectionDown()** method.
- **On Demand**—The connection to the external system is established before the Collaboration executes business rules. After the business rules are executed, the e*Way disconnects from the external system.
- **Manual**—The end user is responsible for establishing connection with the external system by adding Collaboration Rules that call the **connect()** method.

An e*Way Connection with Connection Management allows you to do all of the following.

- Control when a connection is established:
  - Connection occurs when Collaboration is loaded.
  - Connection occurs when Collaboration is executed.
  - Connection is performed manually as an additional method on the ETD.
  - Connection to external are overridden by the custom values in the Collaboration.
- Control when a connection is closed:
  - Disconnect at end of Collaboration's life.
  - Disconnect at end of Collaboration **executeBusinessRules()** processing.
  - Disconnect at timeout.
  - Disconnect on method call.
- Monitor connectivity.
- Specify the methods to be called based on connection status.
- Use Connection Sharing.

The functionality listed above primarily involves changes in how the ETD class is coded. The ability to supply actions in the user Collaboration when an external connection ends abnormally, or when it is initially established or re-established after disconnection, is provided if the user Collaboration implements the

JConnectionNotifier interface. The **JConnectionNotifier** interface contains the methods **onConnectionUp()** and **onConnectionDown()**.

The Connection Management sample uses a very simple TCP/IP client that connects to a server that echoes back messages sent to it. The class diagrams for this sample show the extensions to the basic interfaces used in the sample described in **Chapter 10**.

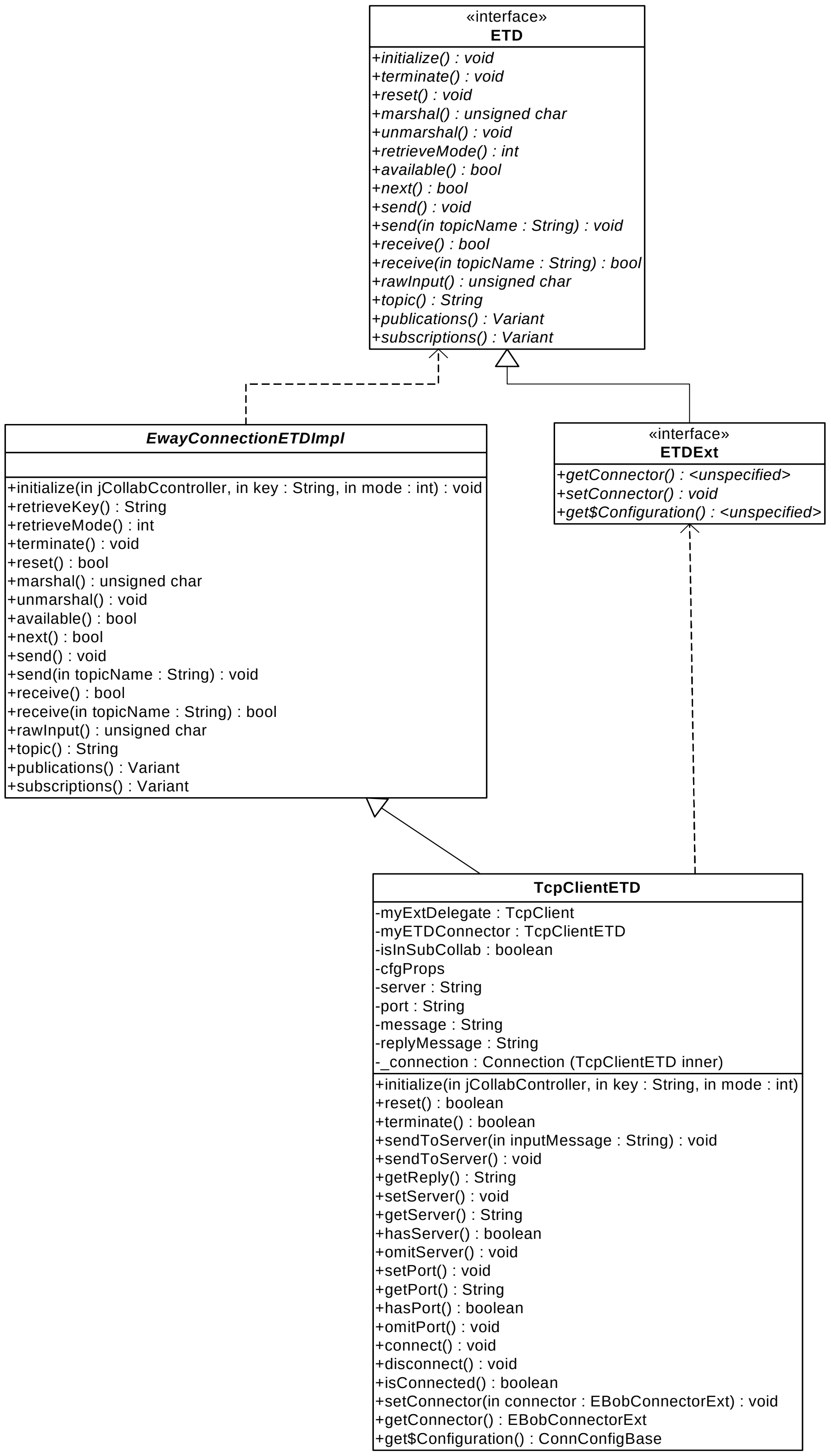
Classes and Interactions for Connection Management

As noted previously, the ETD class and connector class for a Connection Management e*Way are slightly different from the ETD and connector classes for an Automatic Connection e*Way, and have different interactions. These differences are discussed in detail in the following sections.

ETD class

For the Connection Management sample, the class diagram for the ETD class for the e*Way Connection is shown in Figure 17.

Figure 17 ETD Class for a Connection Management e*Way

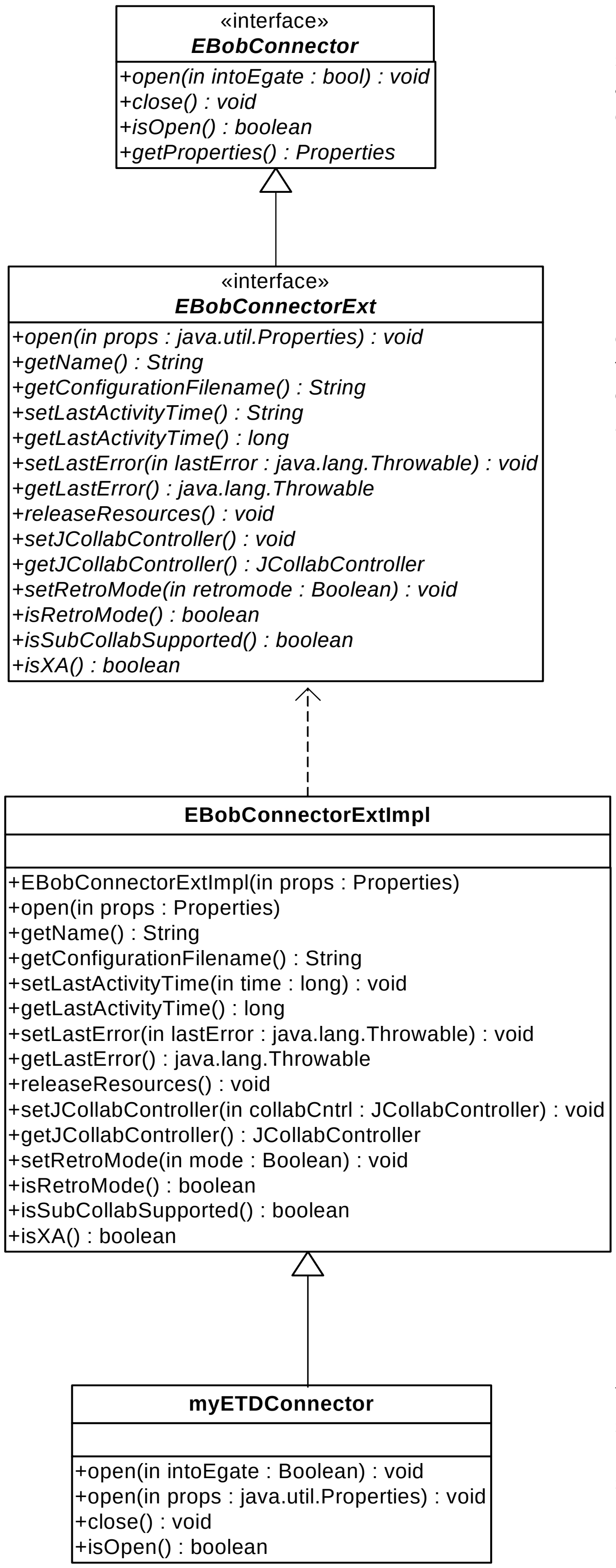


As shown in Figure 17, the *ETDExt* interface extends the *ETD* interface to allow setting and getting the Connector object associated with an ETD. The configuration associated with the ETD can also be obtained as an object. These are needed mainly to allow the user to set connection parameters on the ETD. The corresponding changes in the **.xsc** file are discussed in the next section.

Connector class

For a Connection Management e*Way, the class diagram for the connector class for the e*Way Connection is shown in Figure 18.

Figure 18 Connector Class for a Connection Management e*Way


«interface»
EBobConnector
+open(in intoEgate : bool) : void
+close() : void
+isOpen() : boolean
+getProperties() : Properties
«interface»
EBobConnectorExt
+open(in props : java.util.Properties) : void
+getName() : String
+getConfigurationFilename() : String
+setLastActivityTime() : String
+getLastActivityTime() : long
+setLastError(in lastError : java.lang.Throwable) : void
+getLastError() : java.lang.Throwable
+releaseResources() : void
+setJCollabController() : void
+getJCollabController() : JCollabController
+setRetroMode(in retromode : Boolean) : void
+isRetroMode() : boolean
+isSubCollabSupported() : boolean
+isXA() : boolean
EBobConnectorExtImpl
+EBobConnectorExtImpl(in props : Properties)
+open(in props : Properties)
+getName() : String
+getConfigurationFilename() : String
+setLastActivityTime(in time : long) : void
+getLastActivityTime() : long
+setLastError(in lastError : java.lang.Throwable) : void
+getLastError() : java.lang.Throwable
+releaseResources() : void
+setJCollabController(in collabCntrl : JCollabController) : void
+getJCollabController() : JCollabController
+setRetroMode(in mode : Boolean) : void
+isRetroMode() : boolean
+isSubCollabSupported() : boolean
+isXA() : boolean
myETDConnector
+open(in intoEgate : Boolean) : void
+open(in props : java.util.Properties) : void
+close() : void
+isOpen() : boolean

EBobConnector is the standard interface for all connector classes, either directly or indirectly.

EBobConnectorExt is the extension of EBobConnector that allows your connector class to use Connection Management.

EBobConnectorExtImpl is a Oracle-supplied implementation of EBobConnectorExt.

Your connector class should extend the implementation supplied by Oracle.

As shown in Figure 18, the **EBobConnector** class subclass **EBobConnectorExt** adds the methods that interact with the Collaboration Controller and the Connection Manager. The default implementation **EBobConnectorExtImpl** has been supplied; you can just extend this class to implement the connector class for your e*Way Connection.

Developing an e*Way with Connection Management Support

The following summarizes the steps that differ when developing an e*Way with Connection Management support:

- Java code changes
  - e*Way Connection ETDs must implement ExtETD.
  - In the **initialize()** method:
    - Add a call to obtain your connection manager using **JCollabController.getConnectionManager()**.
    - Use **EBobConnectorExtFactory** to instantiate your EBobConnectorExt implementation class, referred to as your Connector (you can extend the default implementation **EBobConnectorExtImpl**). The factory sets the mode call retro mode (this is false if you are using an e*Gate installation that supports Connection Management).
    - Register your connector with the Connection Manager—for example: **jConnMgr.registerConnector(**myETDConnector**);**
      Here, **jConnMgr** is a reference to the Connection Manager.
  - Make sure your e*Way Connection ETD implements the additional methods **setConnector()** and **getConnector()**. To support Subcollaboration Rules, your e*Way ETDs must be able to share connectors. The **set/getConnector()** methods are called during initialization of your ETD based on whether you are in a Subcollaboration Rule or not.
  - If you are supporting Manual mode, you must add the appropriate attributes and methods that will allow the user to set the connection parameters and perform the following method calls from the ETD:
    - **connect()**
    - **disconnect()**
    - **isConnected()**
  - When a method using your external connection is called, you must call **setLastActivityTime()** to mark when the connection was last used. Ideally, this should be put in one place per operation using the connection. For instance, **setLastActivityTime()** may be called inside the commit method for database operations.
- **.xsc** file changes
  - If you are supporting Manual mode, add nodes for configuration information. This corresponds to the items in your e*Way Connection's **.cfg** file. The values specified in the **.cfg** file are used as the default values. The end user of the e*Way Connection just needs to be able to set the same values to support the

Manual connection mode. The associated Java code for this will be based on the **ConnConfigBase** class.

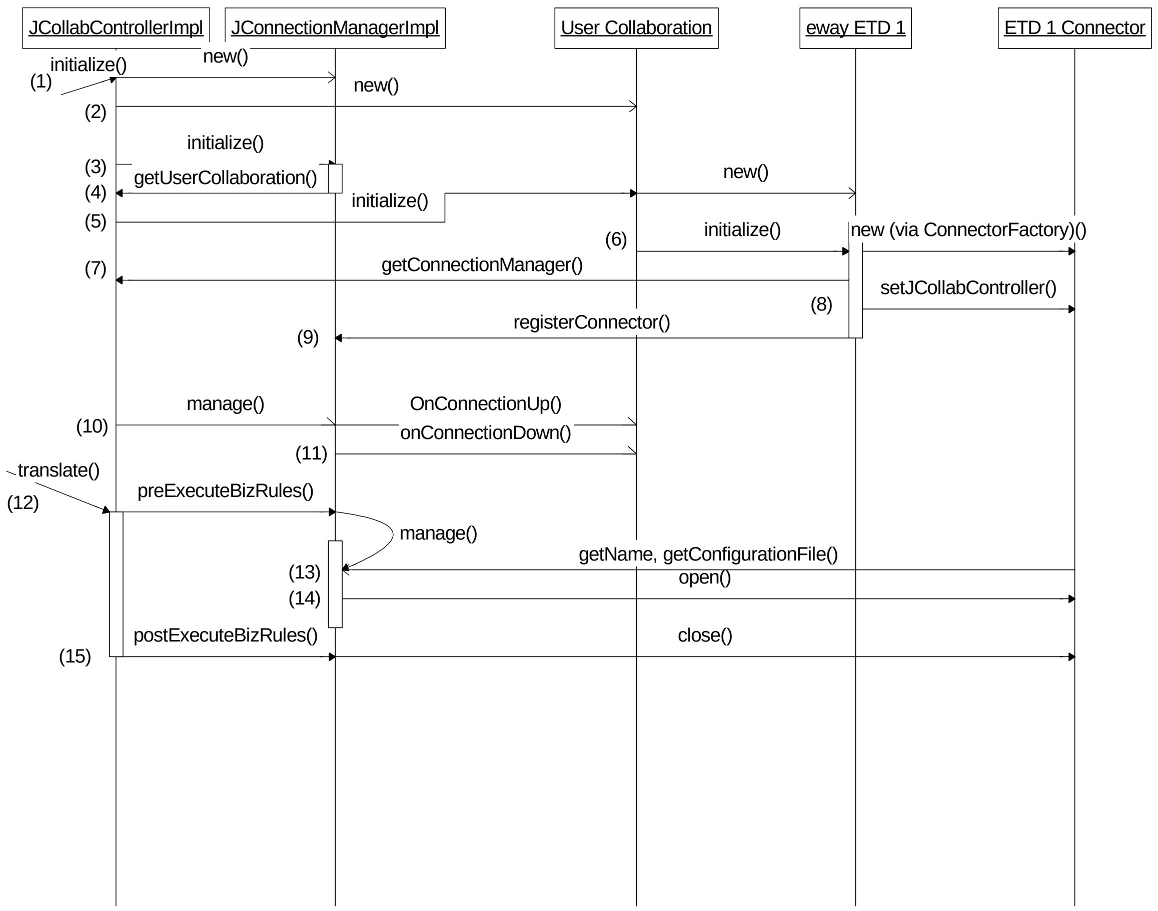
- If you are supporting Manual mode, add nodes for connecting, disconnecting, and checking the external connection:
  - **connect()**
  - **disconnect()**
  - **isConnected()**

The corresponding Java implementation for these methods must use the connection configuration set in the nodes listed above.

- **.def** file changes
  - In the Connector section, add a new item, **Connection Establishment Mode**. The valid values must be one of the following constants:
    - Automatic
    - OnDemand
    - Manual
  - **Connection Inactivity Timeout**—This value is used to specify timeout (in milliseconds) for the Automatic connection establishment mode. If this is not set, or if it is set to 0, the connection is not brought down due to inactivity. The connection is always kept alive; if it goes down, the re-establishment of the connection is automatically attempted. If a non-zero value is specified, the Connection Manager tries to monitor for inactivity so the connection is brought down if the value specified is reached.
  - **Connection Verification Interval**—This value is used to specify the minimum period of time (milliseconds) between checks for connection status to the external server. If the connection to the server is detected to be down during verification, the user Collaboration's **onConnectionDown()** method is called. If the connection comes from a previous connection error, the user Collaboration's **onConnectionUp()** method is called. If no value is specified, 60000 ms is used.

The sequence diagram in Figure 19 shows the interactions among the ETD classes, the Collaboration Controller, and the Connection Manager.

Figure 19 Class Interactions for a Connection Management e*Way



1 In the **JCollabControllerImpl** object, the **initialize()** method is called. An instance of the **JConnectionManager** is created.

2 The user Collaboration is instantiated by the Collaboration Controller.

3 The **initialize()** method of the Connection Manager is called.

4 The Connection Manager obtains a reference to the user Collaboration using it from the Collaboration Controller.

5 The **initialize()** method of the user Collaboration is called by the Collaboration Controller, and the user Collaboration instantiates its e*Way ETDs.

6 The user Collaboration calls the **initialize()** method of its e*Way ETDs. The ETD creates its Connector object in the **initialize()** method, obtaining its configuration from the **.cfg** file as Java properties. The Connector configuration includes the connection establishment mode (Automatic, OnDemand, or Manual).

7 The e*Way ETD gets the Connection Manager from the Collaboration controller so it can register the Connector object it instantiated.

8 The e*Way ETD sets a reference to the Collaboration Controller in its Connector.

9 The e*Way ETD registers its Connector object with the Connection Manager.

At this point, initialization of connectors and registration of interface implementors is completed. The Connection Manager should have performed the appropriate actions to inform the Collaboration Controller of specific junctures when timer events will be generated triggering calls to its **manage()** method.

10 The **manage()** method is the main method that implements the Connection Manager functions. One of its functions is to monitor connections.

11 The Connection Manager polls the registered connectors by calling their **isOpen()** methods. It calls the **onConnectionUp()** and **onConnectionDown()** methods on the user Collaboration based on the change in connection status detected.

12 In the **translate()** method of the Collaboration Controller, the **preExecuteBizRule()** method of the Connection Manager is called before performing any business rules.

13 The **manage()** method will include logic called to loop through the registered connectors and determine if its associated connection needs to be activated.

14 The Connection Manager calls the **open()** method for the connectors that need to activate its connection.

15 The **postExecuteBizRule()** method in the Connection Manager is called after the user's Collaboration Rules are executed by the Collaboration Controller. This method will call the **close()** method of registered connectors for e*Way Connections that are configured to OnDemand mode.

e*Way Connections with Transaction Processing and XA

e*Gate supports both one-phase and two-phase commit transactions through its implementation in user Collaborations of the Java Transaction API (JTA) specification:

http://java.sun.com/products/jta/

JTA specifies an architecture for building transactional application servers and defines a set of interfaces for various components of this architecture. The components are:

- the Application Program (AP)
- the Application Server (AS)
- Resource Managers (RMs)

In e*Gate, the AP is represented by the Collaboration, which runs inside the Multi-Mode e*Way; the Multi-Mode e*Way acts as the AS. The RMs are the ETDs that allow access to external resources.

Note: *Two-phase transaction processing — that is, Prepare/Commit/Rollback — can only be handled at the Root Collaboration Rule level, never by a Subcollaboration Rule.*

Distributed transactions and JTA

A *distributed transaction* is a transaction that goes across multiple independent RMs. For example, the transaction might include an Oracle database at the corporate office, and an SQL Server database at the partner's warehouse. The involved RMs attempt to complete and commit their part of the transaction. If any part of the transaction fails, all RMs roll back their respective updates. This is accomplished using the *two-phase commit* protocol. In this protocol, the activity of one or more RMs is controlled by a Transaction Manager (TM).

JTA provides a Java mapping of the industry-standard XA protocol (defined by the X/Open Consortium) used between RMs and a TM for coordinating transactions across distributed systems. Two-phase commit is part of the XA specification.

Two-phase commit protocol

The activity of one or more RMs is controlled by the TM (also referred to as the *transaction coordinator*). There are five steps in the two-phase commit protocol.

1. An application invokes the **commit()** method in the TM.
2. The TM contacts the various RMs relevant to the transaction and directs them to *prepare* to commit the transaction. This is the beginning of the first phase.
3. Each RM must be able to guarantee the ability to either *commit* the transaction or else to perform a *rollback* of the transaction. (For example, most RMs write to a journal file in durable storage that contains the intended changes.) Any RM that is unable to prepare the transaction sends a negative response to the TM.
4. All responses from the involved RMs are collected.
5. The TM sends a command to the involved RMs. (This is the second phase.) The command takes one of two possible forms:
   - If any response of the RMs is negative, the TM sends a **rollback()** command.
   - If all of the RM responses are affirmative, the TM sends a **commit()** command. The transaction cannot fail after this point.

The Multi-Mode e*Way container has a TM component that interacts with e*Way Connections and ETDs. The TM contains Oracle SeeBeyond JMS publications that act as RMs. e*Gate provides JTA-compliant interfaces which must be implemented by e*Way Connection ETDs to be able to participate as RMs in transactional Collaborations.

For details of implementing one-phase (non–XA-compliant) and two-phase (XA-compliant) transaction processing in e*Way Connection ETDs, see **“Transactional Interfaces for e*Way Connection ETDs” on page 224**.

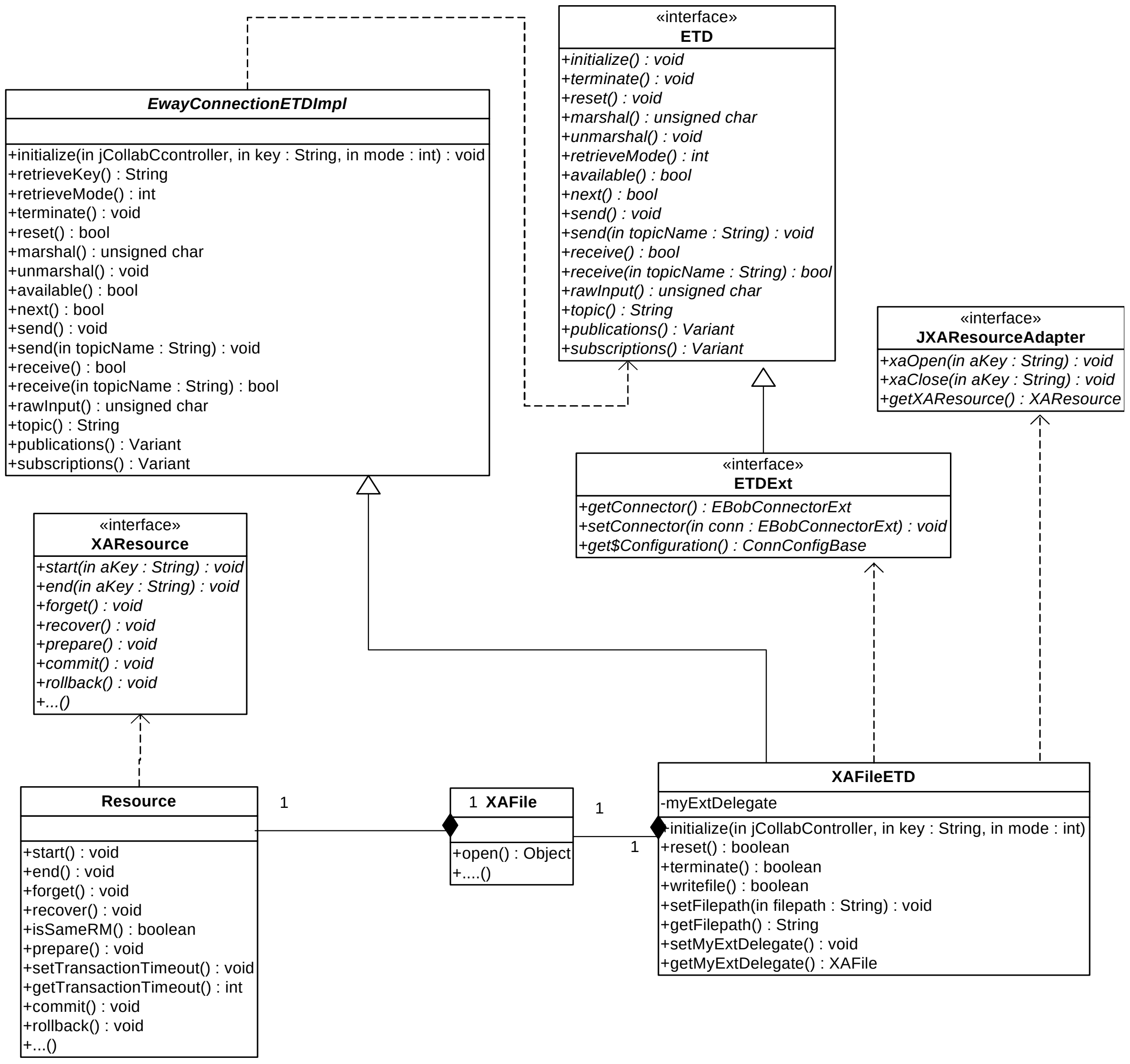
Classes and Interactions for Transaction Processing

For an XA-enabled ETD Connection, the connector class is the same as for the Automatic Connection sample, but the ETD class is slightly different and has different interactions. These differences are discussed in detail in the following sections.

ETD class

For the Transactional sample, the class diagram for the ETD class for the e*Way Connection is shown in Figure 20.

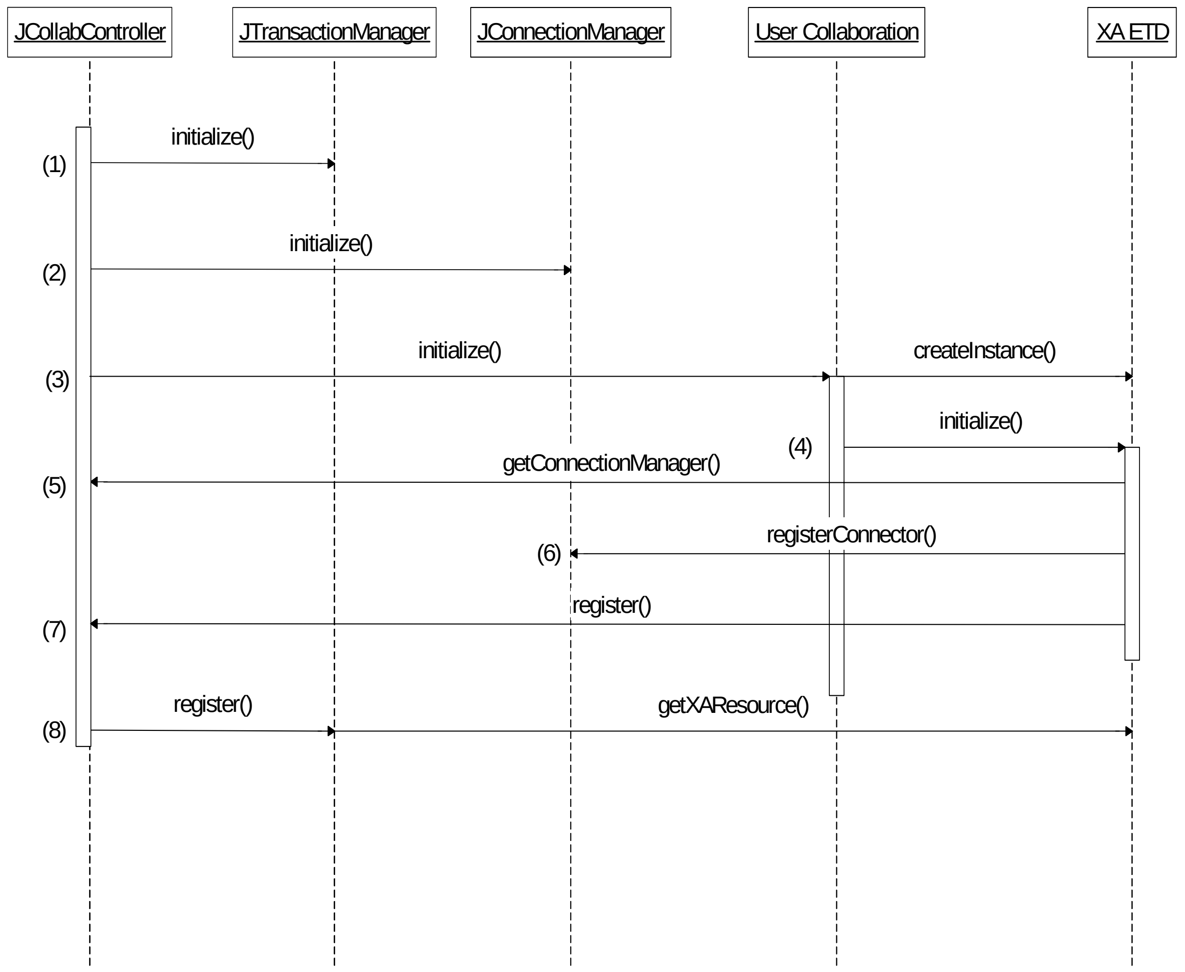
Figure 20 ETD Class for an XA-enabled e*Way Connection


«interface»
ETD
+initialize() : void
+terminate() : void
+reset() : void
+marshal() : unsigned char
+unmarshal() : void
+retrieveMode() : int
+available() : bool
+next() : bool
+send() : void
+send(in topicName : String) : void
+receive() : bool
+receive(in topicName : String) : bool
+rawInput() : unsigned char
+topic() : String
+publications() : Variant
+subscriptions() : Variant
EwayConnectionETDImpl
+initialize(in jCollabCcontroller, in key : String, in mode : int) : void
+retrieveKey() : String
+retrieveMode() : int
+terminate() : void
+reset() : bool
+marshal() : unsigned char
+unmarshal() : void
+available() : bool
+next() : bool
+send() : void
+send(in topicName : String) : void
+receive() : bool
+receive(in topicName : String) : bool
+rawInput() : unsigned char
+topic() : String
+publications() : Variant
+subscriptions() : Variant
«interface»
JXAResourceAdapter
+xaOpen(in aKey : String) : void
+xaClose(in aKey : String) : void
+getXAResource() : XAResource
«interface»
ETDExt
+getConnector() : EBobConnectorExt
+setConnector(in conn : EBobConnectorExt) : void
+get$Configuration() : ConnConfigBase
«interface»
XAResource
+start(in aKey : String) : void
+end(in aKey : String) : void
+forget() : void
+recover() : void
+prepare() : void
+commit() : void
+rollback() : void
+...()
Resource
+start() : void
+end() : void
+forget() : void
+recover() : void
+isSameRM() : boolean
+prepare() : void
+setTransactionTimeout() : void
+getTransactionTimeout() : int
+commit() : void
+rollback() : void
+...()
1
1 XAFile
+open() : Object
+....()
1
1
XAFileETD
-myExtDelegate
+initialize(in jCollabController, in key : String, in mode : int)
+reset() : boolean
+terminate() : boolean
+writefile() : boolean
+setFilepath(in filepath : String) : void
+getFilepath() : String
+setMyExtDelegate() : void
+getMyExtDelegate() : XAFile

Sequence of Interactions

The following sequence diagrams show the interactions among the Collaboration Controller, the Transaction Manager, the Connection Manager, the Collaboration, and the XA-enabled ETD during the initialization phase (Figure 21) and the translation phase (Figure 22).

Figure 21 Sequence of Class Interactions in XA: Initialization Phase



In the following steps, the *Collab Controller* is **JCollabController**, the *Transaction Manager* is **JTransactionManager**, and the *Connection Manager* is **JConnectionManager**.

1 The Collab Controller calls **initialize()** on the Transaction Manager. This process includes initialization of its associated Transaction Coordinator component.

2 The Collab Controller calls **initialize()** on the Connection Manager. The Connection Manager starts with an empty list of connectors and obtains a reference to the user Collaboration. If the user Collaboration implements **JConnectionNotifier**, the Connection Manager will call **onConnectionUp()** and **onConnectionDown()** on the user Collaboration based on the state of registered connectors.

3 The Collab Controller calls **initialize()** on the user Collaboration.

4 The user Collaboration instantiates its ETDs when the first Event arrives (or as triggered by the “get” interval of the inbound ETD—for example, Figure 21 shows the XA ETD, which may be one of the Collaboration's inbound or outbound ETDs).

5 The user Collaboration calls the **initialize()** method for all its ETDs.

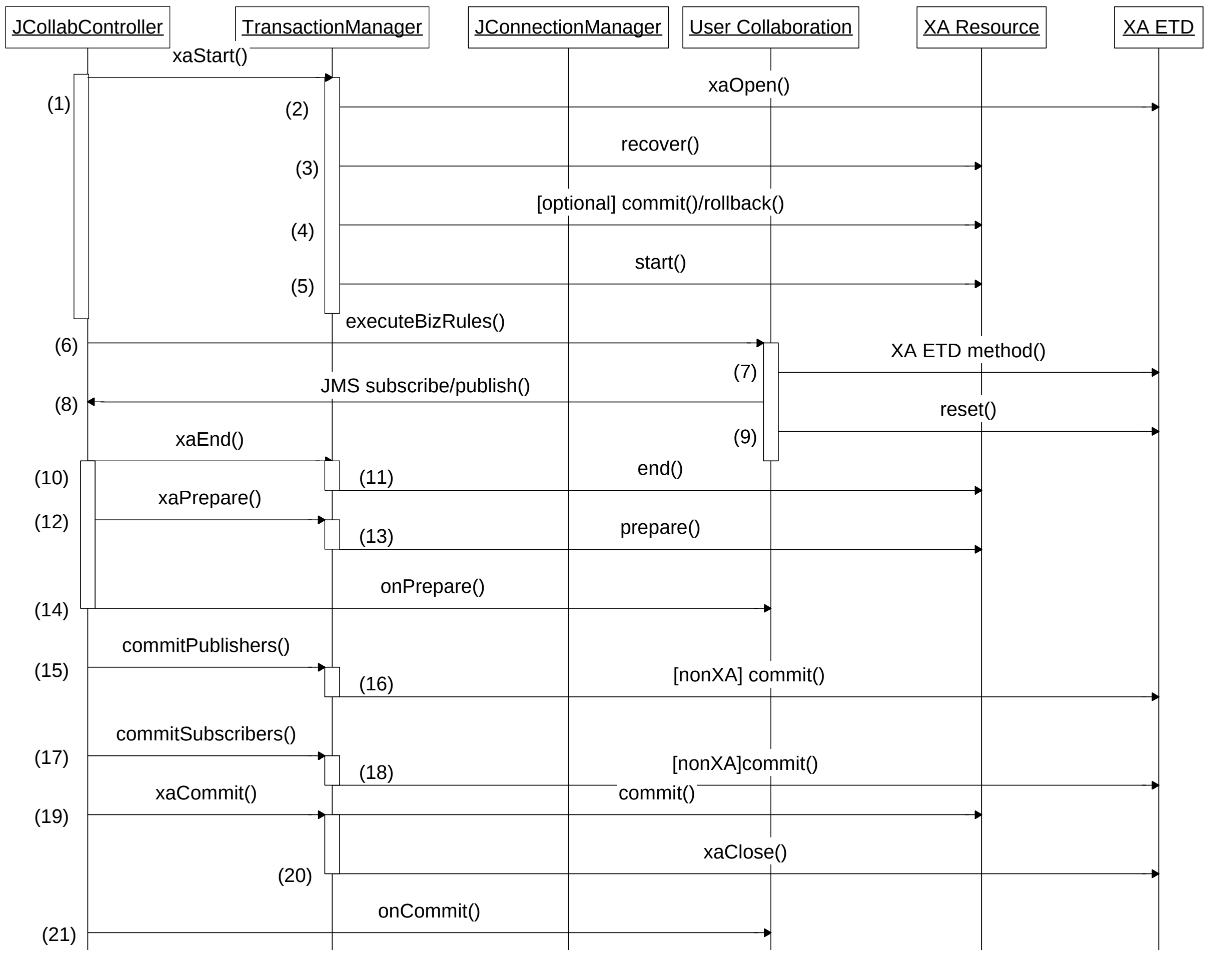
6 Each ETD—assuming it supports Connection Management—registers itself with the Connection Manager. The Connection Manager adds the connector to its appropriate connector list based on the connection establishment mode.

7 The XA ETD registers an object which implements the **JXAResourceAdapter** interface with the Collab Controller. This is generally XA ETD object itself and will be treated by the Transaction Manager as the Resource Manager.

Note: *For non-XA transactions, the ETD uses the **registerTransactionAdapter()** method to register.*

8 The Collab Controller registers the Resource Manager that was registered in step 7 with the Transaction Manager. The associated Transaction Coordinator uses the **getXAResource()** method to obtain a reference to the XA resource associated with the Resource Manager, and then the Transaction Manager enlists the XA resource. The sequence of these interactions is shown in Figure 22.

Figure 22 Sequence of Class Interactions in XA: Translation Phase



1 On each iteration of the Execute phase of a Collaboration, the Collab Controller calls **xaStart()**, demarcating the start of the XA transaction.

2 The Transaction Manager calls **xaOpen()** on the ETD (Resource Manager).

3 If necessary, based on the state of the transaction, the Transaction Manager calls **recover()** on XA resource.

4 As necessary (as part of recovery), the Transaction Manager calls **commit()** or **rollback()**.

5 The Transaction Manager calls **start()** on the XA resources that are part of the transaction.

6 The Collab Controller starts calling the business rules in the user Collaboration.

7 The business rules in the user Collaboration include calls to the methods exposed by the XA ETD.

8 If there are any JMS message subscriptions and publications involved in the Collaboration, the messages are obtained or published from JMS through the Collab Controller.

9 After all rules in the Collaboration have been performed, the ETD's **reset()** method is called.

10 The Collab Controller can now demarcate the end of the XA transaction, calling **xaEnd()** on the Transaction Manager.

11 The Transaction Manager calls the **end()** method for all XA resources.

12 The Collab Controller calls **xaPrepare()** on the Transaction Manager to start the prepare stage of the two-phase commit protocol.

13 The Transaction Manager instructs XA resources to perform **prepare()** to check if all participants are ready to commit the transaction.

14 The Collab Controller calls **onPrepare()** on the user Collaboration.

15 In steps 15 through 18, for ETDs that implement the **JTransactionAdapter** interface supporting one-phase commit, the Collab Controller ensures that **commit()** is called. This is also done for JMS publications, JMS subscriptions, and non-XA ETDs.

19 The Collab Controller calls **xaCommit()** to commit the XA transaction. This causes **commit()** to be called on all XA resources.

20 The Transaction Manager calls **xaClose()** on the XA ETD to demarcate the end of the XA transaction.

21 The Collab Controller calls **onCommit()** on the user Collaboration.

6.3.2 Compiling .java Files

A**compile.bat** file (and, for UNIX, **compile.sh**), are provided with the sample. This file sets the CLASSPATH and creates a **.jar** file for the **.class** files upon compilation of the **.java** files. For details, see **"Customizing the Compile Script" on page 174**.

The following edits must be made to the **compile.bat** (or **compile.sh**) file:

1 Specify the correct location if your e*Gate installation resides anywhere other than the root **\eGate** directory on your local drive.

2 Specify the correct path location for your JDK installation.

3 When creating e*Way Connections from scratch, modify the directory locations in the **.bat** (or **.sh**) file as needed.

How Classpaths Are Determined

When a Multi-Mode e*Way starts the JVM for running Java Collaborations, it specifies the classpath based on the configuration specified in the **.cfg** file for the Multi-Mode e*Way. Each Collaboration also has an associated **.ctl** file—<*collabruleName*>**.ctl**—which lists any class files and **.jar** files the Collaboration needs at run time.

e*Gate version 4.5.1 and e*Gate version 4.5.2

In e*Gate version 4.5.1, the classpath is set up by the e*Way and specified when the JVM is started. In e*Gate version 4.5.2, a *dynamic class loader* is used. The function of a dynamic class loader is to load the classes the Collaboration needs, effectively setting the classpath. This occurs when a Collaboration is initialized (after the JVM has already started).

6.3.3 Creating .ctl Files

The **.ctl** file contains the information required by the GUI to be able to successfully load the ETD; for example, the samples use a file named **install.ctl** (see **"Editing/Viewing the .ctl Files" on page 175**, or similar material on page 215 or page 235). If the ETD uses any **.jar** files, they must be specified in the **.ctl** file. The following considerations apply:

- The file name of the **.ctl** file must match the file name of the **.xsc** for the ETD.
- The **.ctl** file must specify all **.jar** files containing classes associated with your ETD.
- The **.ctl** file must specify all third-party **.jar** files associated with your ETD.

Note: *.ctl file names must be lowercase. The Visual Basic Collaboration Editor will not download any files that are not lowercase.*

6.3.4 Creating .def Files

The e*Way Connection is configured using the e*Way Connection Editor, a GUI that enables you to change configuration parameters quickly and easily. A default configuration-file template (**.def** file) configures the e*Way Connection Editor to gather those parameters by specifying the name and type of each parameter, as well as other information (such as the range of permissible options for a given parameter).

The **.def** file has three major divisions:

- The *header* describes basic information about the file itself, such as version number, modification history, and comments.
- The *sub-header* contains several read-only variables that are for internal use. Those default values must not be modified.

- The *body* contains configuration parameters, grouped into sections:
  - Two sections (Connector and External Configuration) must be included in all e*Way Connection **.def** files.
  - Additional sections are added as needed to support user-created functions.

For detailed information, see **Appendix A "Extending the .def File" on page 256**.

6.3.5 Creating the .xsc File

The developer must create the required **.xsc** file including the necessary functionality and structure. The **.xsc** file is an XML file that allows the GUI to load your ETD and generate correct code in the Collaboration Rules Editor. For more information, see **"Event Type Definitions" on page 79**.

6.4 Building and Testing Your Components

To validate the e*Gate components you have created, you must perform the following steps:

1. Open or create a schema into which to commit the files, and then run **installETD** script (**installETD.bat** on Windows, or **installETD.sh** on UNIX) to make the files available to e*Gate.
2. Validate the results by refreshing the schema, run the components, and then monitor the components using the in-schema Java Debugger. For more information on the Java Debugger, see **"Troubleshooting and Debugging" on page 244**.

6.4.1 Running the installETD Script

The kit provides the **installETD** script as an easy way to install the customized files into an existing e*Gate schema. For example, to double-check a sample the first time you go through it, you can create an empty schema, run **installETD**, and check to see that the results match your expectations. Alternatively, you can run **installETD** to update a complex pre-existing schema with minor changes.

If your Participating Host and Registry Host are running on different computers, you must have *stcinstd* running against the schema and Registry Host in use while running the **installETD** script.

1. Type the following text at the command line:

```
stcinstd -rh <host> -rs <schema> -un <username> -up <password> -ss
```

2. Press **ENTER**.

Note: *In the* ***stcinstd*** *command line as shown, the flag* ***-ss*** *is optional and means to run the host as a service.*

For information on the Windows version of the **installETD** script, see **Windows: installETD.bat** on page 76. For information on the UNIX version of the **installETD** script, see **UNIX: installETD.sh** on page 76.

Windows: installETD.bat

Usage

```
installETD.bat -e etdName -s schemaName [optional params]
```

Parameters

-g <*directoryName*>— Specifies the name of the e*Gate root directory (the same value specified in **compile.bat** as EGAT_ROOTDIR). If not specified, the default is: **\eGate**

-e <*etdName*>—Required; specifies the name of the ETD to install. The **installETD** script fails if no corresponding install subdirectory exists.

-s <*schemaName*>—Required; specifies the name of the e*Gate schema.

-h <*hostName*>—Specifies the name of the e*Gate Registry Host. If not specified, the default value is: **localhost** (in a Windows environment especially, it is common for an e*Way developed using this kit to be developed on the same machine where e*Gate is installed).

-r <*portNumber*> — Specifies the port number where the e*Gate Registry runs. If not specified, the default value is: **23001**.

-u <*userName*>—In conjunction with the **-p** flag, must specify a username/password combination that is valid for this Registry Host.

-p <*password*>—In conjunction with the **-u** flag, must specify a username/password combination valid for the Registry Host.

-?— Displays help information.

Example

To run the **installETD** script, open a Command Prompt, change the correct directory, and enter the following command:

```
.\installETD -e yourETD -s yourSchema -h localhost -g c:\eGate
```

Notes

By default, an installation log file named <*etdName*>**_install_log.txt** is created in the root directory of \. If a different location is preferable, you can modify the value of environmental variable __LOGDIR, either within the **installETD.bat** or externally.

UNIX: installETD.sh

Usage

```
./installETD.sh -e etdName -s schemaName [optional params]
```

Parameters

-g <*directoryName*>— Specifies the name of the e*Gate root directory (the same value specified in **compile.sh** as **EGATE_ROOTDIR)**. If not specified, the default is: **/eGate**.

-e <*etdName*>—Required; specifies the name of the ETD to install. The **installETD** script fails if no corresponding install subdirectory exists.

-s <*schemaName*>—Required; specifies the name of the e*Gate schema.

-h <*hostName*>—Specifies the name of the e*Gate Registry Host. If not specified, the default value is: **localhost**.

-r <*portNumber*> — Specifies the port number where the e*Gate Registry runs. If not specified, the default value is: **23001**.

-u <*userName*>—In conjunction with the **-p** flag, must specify a username/password combination that is valid for this Registry Host.

-p <*password*>—In conjunction with the **-u** flag, must specify a username/password combination valid for the Registry Host.

-v—Specifies verbose mode (full prompts and explanations).

Example

```
./installETD.sh -g /home/user1/eGate -e yourETD -s yourSchema -
h localhost -u Administrator -p STC
```

By default, an installation log file named <*etdName*>**_install_<###>.txt** is created, where <###> is the process-ID (pid) of the current process. If a different location is preferable, you can modify the value of environmental variable DIR_SOURCE, either within the **installETD.sh** or externally.

Note: *For a complete explanation on using the e*Gate command line, including the* ***stcinstd*** *command, see the e*Gate Integrator System Administration and Operations Guide.*

6.4.2 Validating the Results

1. In e*Gate Schema Designer, open your schema (or, if the schema is already open, on the **View** menu, click **Refresh** to update the display with the latest changes), and then double-check that your new files have been successfully committed.
2. Start the Control Broker and then, in Schema Manager, start all the components in the schema. Double-check any Alert Notifications the system has issued, and browse the trace and logging files as needed. To use the e*Gate Java Debugger, right-click the Multi-Mode e*Way and, on the shortcut menu, click **Debugger**.

For more information on Alerts, trace and log files, and the debugger, see the appropriate topics in **"Troubleshooting and Debugging" on page 244**.

6.5 Packaging and Distribution

When developing your files, it is important to use a source code control system to manage these components as you make changes between builds, to ensure that files that logically belong together are updated as a unit. For example, when you check in files for SampleETD, you should create a separate directory where the **.ctl**, **.def**, and **.xsc** files for SampleETD are together.

For your source code directory structure, you can use whatever setup you normally use; the directory structure used in the samples provided in **samples\sdk\gmeek\** is one possibility, but not a requirement.

About the Packaging of the Samples in the Kit

The **gmeek.taz** file is extracted into any directory, using either the **Winzip** program or the UNIX **uncompress** command. There is no restriction on where you extract the contents of this file and on the directory structure you use for your source code. The directory structure of the **installETD** directory, on the other hand, is set up so that the **installETD** scripts are used in a generic fashion (see **“Scripts”**). It is simplest to keep files that make up a particular ETD together in the same directory, namely the **.jar** file output from the compiled source code, the **.xsc**, **.def**, and **.ctl** files. For distribution, these files may be bundled into a compressed file with an accompanying version file.

6.5.1 Scripts

Once the components are built, they are deployed in an e*Gate installation by means of scripts, either as **.bat** files on Windows, or as **.sh** files on UNIX. Sample installation scripts have been provided with the kit.

The **installETD** script performs the following steps:

- Updates the **connectionpoint.ini** file, located in the **configs** directory of the e*Gate Registry repository.

  The **connectionpoint.ini** file is a Windows-style **.ini** file that has a separate section for each ETD. Each ETD section has a **configsDir** variable that specifies the subdirectory in the **configs** directory where the ETD's **.def** file is stored. The utility **stcregutil.exe** must be used to download files from the Registry. A **.ctl** file is used to specify what files to download or commit to the Registry.

- Commits the e*Way ETD’s **.jar** file, **.xsc** file, and **.def** file.

  There is a **.ctl** file associated with the e*Way Connection. For example, if the name of the ETD is **SampleETD**, a **.ctl** file named **SampleETD.ctl** will be committed into the **etd** directory of the e*Gate Registry repository.

  The **.jar** and **.xsc** files for the ETD must also be committed into the **etd** directory.

  The **.def** file for the e*Way must be committed into the **configs** subdirectory for the e*Way; for example, the file **SampleETD.def** would be committed into the directory **configs\SampleETD\**.

Chapter 7

Event Type Definitions

This chapter provides the following:

- A description of the role of the Event Type Definition (ETD).
- An overview of the Application Programming Interfaces (APIs).
- An overview of the XSC format. Complete details on the XSC formats are provided in **Appendix B "The XSC Format" on page 287**.
- General notes, tips, and caveats for creating custom ETDs.

7.1 Events and ETDs

e*Gate uses the following terminology.

- An *Event* (equivalently, a *message*) is a unit package of data processed by the e*Gate system. The data has a defined structure, such as a known number of fields with known characteristics and delimiters. Events are classified by Event Type.
- An *Event Type* (called a *topic* in some contexts) is a class of Events with common characteristics. An Event Type is also a logical name entry in e*Gate that points to a single Event Type Definition (ETD).
- An *Event Type Definition*, or *ETD,* is a structural representation of an Event—in other words, the blueprint for describing an Event—that Collaboration Rules can use when parsing, transforming, or routing data. An ETD has a treelike structure, and is composed of entities called *nodes*. A *root node* has no parent; a *leaf node* (also called a *field*) has no children. For more information on nodes, see **"ETD Nodes"**.
- An *ETD Builder* is a special-purpose tool, usually a wizard, for guiding an end user through the process of creating an empty ETD of a specific type. For information on creating ETD builders, see **Chapter 8** and **Chapter 9**.

Messageable and Non-messageable ETDs

- A *messageable* ETD, also called a *marshalable* ETD, is one that contains **no** references to content external to itself—such as to the current time, or to data residing in an external system. A messageable ETD instance is *marshaled* (compressed into a data stream), stored in an IQ or Oracle SeeBeyond JMS, and then extracted and *unmarshaled* (parsed) as needed.

  If an ETD node represents an object that is available only by calling an external system, such as a database table, stored procedure, or prepared statement, then the

ETD is *non-messageable*, and can thus contain internal state information or external references that are resolved dynamically. The kit helps you create non-messageable ETDs only.

ETDs and e*Gate

The e*Gate Schema Designer contains two important GUIs, the ETD Editor and the Collaboration Rules Editor. When you develop a well-formed **.xsc** file and install it to the correct location within e*Gate, end users can:

- Load the **.xsc** file into the ETD Editor and use the Editor to view the ETD's structure and properties. The ETD Editor has GUI features like tree expand/compress, property dialogs, and internal and external templates, Delimiters dialog box, and so forth. The ETD Editor also can also be used to compile, test, and promote the ETD.
- Load the ETD (along with other ETDs) into the Collaboration Rules Editor and define business rules to describe the relationships between inbound and outbound ETD instances. In the Collaboration Rules Editor you can use simple point-and-click and drag-and-drop techniques to generate, modify, and compile Java Collaborations.
- Use the Configuration Editors to create and modify configuration files (**.cfg** files) for e*Ways and e*Way Connections they create, based on the default configuration-file templates (**.def** files) and the **connectionpoint.ini** file you supply. Details of this are discussed in depth elsewhere; see **"Creating .def Files" on page 74** and **"Extending the .def File" on page 256**.

7.1.1 ETD Nodes

e*Way ETDs contain two types of nodes, *attribute nodes* and *method nodes*.

- **Attribute nodes** correspond to private member variables in the ETD class. Such nodes must have the Java Bean–style interface, with public getter and setter methods for each exposed attribute—in other words, **get*YourNodeName*()** and **set*MyNodeName*()**. For read-only attributes, the setter method would be omitted. In general, static configuration parameters are obtained through the **.cfg** file (whose default values are specified in the **.def** file), but parameters that might change depending on the incoming Event are obtained through attributes exposed in the ETD.
- **Method nodes** may operate on parameters based on the exposed attributes, and they may also be based on the parameters passed. As in Java, the XSC specification allows for defining overloaded methods—in other words, a single method name can have many different signatures. There are no restrictions on the methods exposed in the ETD. As the developer of the e*Way Connection, you can provide methods that call third-party APIs directly or indirectly through wrapper classes, depending upon the design patterns used in the classes supporting the exposed functionality.

7.2 Overview of the XSC Format

Note: *The XSC 0.4 format is supported in e*Gate version 4.5.1 and later; the XSC 0.6 format is supported in e*Gate version 4.5.2 and later. Throughout this chapter, whenever an XSC 0.6 construction is unsupported in XSC 0.4, it is explicitly labeled as being 0.6–specific.*

Terminology and typography

In this discussion, the word "tag" denotes the <> characters and the string they enclose, whereas the word "entity" denotes what is described by the tag—in other words, the content, from the begin-element tag through the end-element tag. Following the usual shorthand, however, the typographical convention "the <xyz> entity" is used to indicate the **xyz** entity, including its attributes and content.

7.2.1 General Rules for Entities

Every **.xsc** file must be a valid XML string composed of a valid combination of <etd>, <javaProps>, <node>, <method>, and <param> entities and their attributes, as well as possibly other entities not mentioned here.

Note: *Rarer and more obscure entities are covered in* **Appendix B**.

- **<etd>**: There is exactly one <etd> entity in every **.xsc** file: Since an <etd ...> tag begins the file and an </etd> tag ends the file, the entire file is itself a single <etd> entity. An <etd> entity must contain one or more <node> entities, and can contain at most one <javaProps> entity.
- **<javaProps>**: There must be zero or one <javaProps> entity in every <etd> entity. Standard practice is to place the <javaProps> entity directly after the <etd> entity. In XSC 0.6, every <etd> entity must contain a <javaProps> entity.
- **<node>**: There must be one or more <node> entities in every <etd> entity, and zero or more <node> entities in every <node> entity. A <node> entity can contain zero or more other <node> entities and zero or more <method> entities. A <node> entity is of type "CLASS" if and only if it contains one or more <node> entities. The parent of a <node> entity must be either a <node> entity of type "CLASS" or else the <etd> entity.

  <class>: An alias for <node>. If you use it, restrict it to those <node> entities that are directly beneath the <etd> entity.
- **<method>**: There must be zero or more <method> entities in every <node> entity. A <method> entity can contain zero or more <param> entities; it cannot contain any other entities. Ignoring the <interface> entity, the parent of a <method> entity must be a <node> entity.
- **<param>**: There must be zero or more <param> entities in every <method> entity. A <param> entity cannot contain any other entities. The parent of a <param> entity must be a <method> entity.

7.2.2 General Rules for Attributes

Each entity has its own set of required and optional attributes. The complete set of required and recommended attributes for each entity is described in detail in **Appendix B**. The following is true for all entities:

- Every entity must have a **name** attribute and a **uid** attribute, and no **uid** attribute is repeated within the **.xsc** file — in other words, each entity must be uniquely identifiable by the value of its **uid** attribute.
- Every entity can have an optional **comment** attribute.

Using Entities and Attributes

<etd>

In XSC 0.6, an <etd> entity must contain one or more <node> (and/or <class>) entities and exactly one <javaProps> entity; it cannot contain any other types of entity.

The <etd ...> tag has three required attributes: **type**, **xscVersion**, and **uid**.

- **type**: The value of the **type** attribute is used to define the ETD type. It is this value that the user sees in the selection list when defining the connection type, and this value must match the file name of the **.ctl** file.
- **xscVersion**: For XSC 0.6, you should always set **xscVersion**=”0.6” to maximize performance by bypassing backward compatibility checks against previous versions.

Other attributes are discussed in **“The <etd> Entity” on page 289**.

<javaProps>

In XSC 0.6, there is exactly one <javaProps> entity in every <etd> entity. A <javaProps> entity can contain zero or more <jar> entities and zero or more <interface> entities; it cannot contain any other types of entity. The parent of a javaProps entity must be the <etd> entity.

The <javaProps ...> tag has three required attributes: **package**, **class**, and **uid**.

- **package** and **class**: Together, these two attributes define the fully qualified Java class name of the Java class that implements the **.xsc** file. For example, when class=”Z”, package=”w.x.y”, the filename will be “w/x/y/Z.java”.

Other attributes are discussed in **“The <javaProps> Entity” on page 292**.

<jar>

There are zero or more <jar> entities in every <javaProps> entity. A <jar> entity cannot contain any other types of entity. The parent of a <jar> entity must be the <javaProps> entity.

The <jar ...> tag has two required attributes: **file** and **uid**.

- **file**: The value for the **file** attribute must be delimited by *forward* slashes (/), regardless of operating system.

<interface>

The <interface> entity is reserved for future use.

<node>

name=*text*

Default value: (undefined)

type

Each node must be of one of the following types:

- A *template* node has **type**="CLASS" and can have a **public**=*boolean* attribute. It is characterized by being a top-level node. The parent of a template node is the <etd> entity. The sequence of <node> entities immediately inside the <etd> entity is known as its *local template list*. Each local template must have a name that is unique in the list—that is, no two top-level nodes may have the same name. If the <etd> entity has a **name** attribute, its value must match the name attribute of a template node.
- A *composite* node also has **type**="CLASS", but lacks a **public** attribute. A node that is a parent element (as opposed to a leaf) is composite if it is not a template.

Note: *For both composite nodes and template nodes, the **javaType** attribute value defaults to the fully qualified Java class name formed by the <javaProps> package value, followed by all ancestor node names and the name of the node itself, separated by "."; node names in this case are the <node> **javaName** values if present, and name values otherwise.*

- A *simple* node has **type**="FIELD". Simple nodes describe data fields that are not further subdivided or described elsewhere in the ETD. The **javaType** attribute value defaults to "java.lang.String"; in this case, the **encoding** attribute can specify the character encoding name used to convert between raw input/output byte data and internal string values.
- An *enumeration* node has **type**="ENUMERATION", and is also a leaf node. Nodes of this type require a list of zero or more members that represent enumeration elements. Enumeration elements are <member> entities with the allowable attributes **name** and **value**. If any of the members of an enumeration has the **value** attribute, then all its member must have it. If this is the case, all values should be distinct strings.
- A *reference* node has **type**="REFERENCE", and is a surrogate for an ETD part that is defined either within the same **.xsc** file (in which case the node is called an *internal reference* to a *local template*), or to a global template defined in another **.xsc** file (in which case the node is called an *external template*). The distinction is signaled by the **reference** attribute. When **reference**=*filepath* is defined, *filepath* is the relative path and filename of the external **.xsc** file being referenced.

The <node> entity can take on a very wide variety of attributes, depending on its type. For a complete discussion, see **"The <node> and <class> Entities" on page 297** and **"Table of XSC Entities and Their Attributes" on page 300**.

<method>

A <method> entity describes an explicit public method associated with a particular generated class (if the parent of the <method> is a <node> entity) or implemented interface (if the parent is an <interface> entity) in the ETD.

Implicit methods, on the other hand, are generated automatically for each node depending on its attributes. Thus, you need not create Bean-style getter/setter methods for each node unless you want to override them. Examples include:

- **get**<*nodeName*>**()** is always generated.
- **set**<*nodeName*>**()** is generated for writable nodes—in other words, nodes for which **readOnly**="false".
- **has**<*nodeName*>**()** is generated for nodes that might not receive data— in other words, nodes for which **minOccurs**="0".
- **count**<*nodeName*>**()** is generated for repeating nodes— in other words, nodes for which **maxOccurs** is either greater than "1" or equal to "unbounded".

The <method ...> tag has four required attributes: **name**, **signature**, **returnType**, and **uid**. (The **signature** attribute is syntactically optional, but highly recommended.)

- **returnType**: The value of the **returnType** attribute is required to specify the data type of the method. Examples:
  - **returnType**="void"
  - **returnType**="java.lang.String"

  Array types are denoted by a trailing pair of brackets: **returnType**=*datatype*[].
- **signature**: The presence of a valid **signature** attribute allows the e*Gate system to load the ETD more quickly, since it need not parse every method entity on the fly. For complete details on the **signature** attribute of the <method> entity, see **"Method Signature Syntax" on page 304**.

For additional details on the entity itself, see **"The <method> Entity" on page 298**.

<param>

There are zero or more <param> entities in every <method> entity. A <param> entity cannot contain any other entities. The parent of a <param> entity must be a <method> entity.

The <param ...> tag has three required attributes: **name**, **paramType**, and **uid**.

<throws>

At the present time, the <throws> entity is reserved for future use.

7.2.3 Sketching an Outline: Entities and Their Hierarchy

The outline of an **.xsc** file will always resemble the following:

```
(1)   <etd ...>
(2)     <javaProps .../>
(3)     <node ...>
(4)       <method .../>
```

```
    <method ...>
      <param .../>
    </method>
  </node>
  <node ...>
    <node ...>
      <node .../>
      [...]
    </node>
  </node>
  <node ...>
    [...]
  </node>
  [...]
</etd>
```

If you are familiar with the interactive ETD Editor, you can see the similarities between the lexical format shown above and the graphical tree displayed in the GUI:

- There is one container of the entire structure, designated by `<etd ...>` and `</etd>` tags.
- There are multiple nodes belonging to a parent, and nodes can nest to an indefinite degree, but each node is the child of one unique parent. Thus, there is a strict hierarchy, and each node has a uniquely determined level in the hierarchy.
- A node can (but need not) contain one or more methods, and a method can (but need not) contain parameters.

If you are creating an ETD from scratch, it is strongly recommended that you first sketch out its overall outline in a manner resembling the outline shown above.

7.2.4 Fundamental Entity Relationships and Attributes

The following six lines represent a near minimum for a valid **.xsc** file:

```
<etd name="myRoot" type="myType" xscVersion="0.4" uid="0">
  <javaProps package="myPackage" class="myClass" uid="1"/>
  <node name="myRoot" type="CLASS" uid="6">
    <node name="Child" type="FIELD" uid="28" />
  </node>
</etd>
```

This file illustrates several important points:

- The contents of the **.xsc** file must be a valid XML string.
- The <etd ...> tag occurs once, at the beginning, and the </etd> tag ends the file.
- There is no more than one <javaProps> entity, and its parent is the <etd> entity.
- Attribute values are almost always strings enclosed by double quotes.
- Each entity requires a **uid** attribute, and all **uid** attribute values must be unique.
- In addition to **uid**, the <etd> entity takes other required attributes:
    - A **name** attribute, which must match the **name** attribute of exactly one <node>.
    - A **type** attribute. In e*Gate, this must match the filename of a **.ctl** file.

  - An **xscVersion** attribute is required in XSC 0.6 (and must have the value “0.6”). It is also highly recommended in XSC version 0.4, because it provides better performance, compatibility, portability, and maintainability.
- The <javaProps> entity has two additional required attributes: **package** and **class**.
- Every <node> entity has two additional required attributes: **name** and **type**.
- Any <node> entity of type=”CLASS” must contain at least one other entity.
- A <node> entity of type=”FIELD” cannot contain any entities.

7.3 Designing Your Entities

After you have created an outline of your **.xsc** file (see **“Sketching an Outline: Entities and Their Hierarchy” on page 84**), it is helpful to flesh out the outline to some extent by sketching in the required and recommended attributes. For example:

```
(1)   <etd name="" type="" packageName="" uid="" comment=””>
(2)     <javaProps package="" class="" uid="" comment=””/>
(3)     <node name="" type="CLASS" [other attributes] uid="" comment=””>
(4)       <node type="" [other attributes] uid="" comment=””/>
(5)       <node
(6)         name=""
(7)         type=""
(8)         otherattr=""
(9)         [...]
(10)        uid=""
(11)        comment=””
(12)      </node>
(13)      <method name="" returnType="" signature=”” uid="" comment=””/>
(14)      <method name="" returnType="" signature=”” uid="" comment=””/>
(15)        <param name="" type="PARAM" paramType="" uid="" comment=””/>
(16)      </method>
(17)      <method .../>
(18)    </node>
(19)  </etd>
```

7.3.1 Sample File XAFileETD.xsc

This section provides a listing and an explanation of the **XAFileETD.xsc** file provided with this kit.

Listing

```
(1)   <?xml version="1.0" encoding="UTF-8"?>
(2)   <etd name="XAFileETD" type="XAFileETD" xscVersion="0.6" uid="0" >
(3)     <javaProps package="xasample" class="myClass" codeAvailable="true"
      uid="1" />
(4)     <node name="XAFileETD" type="CLASS" uid="2">
(5)       <method name="writeFile" signature="writeFile()Z"
      returnType="boolean" comment="This method sends the passed string to
      the specified file (if specified) under XA transactional environment."
      uid="3">
(6)         <param name="XAFileETDObj" paramType="xasample.XAFileETD"
      comment="The XAFileETD object itself or null for default." uid="4" />
(7)         <param name="msg" paramType="java.lang.String" comment="The
      message to post to the file." uid="5" />
```

```
(8)          <param name="filenamepath" paramType="java.lang.String"
     comment="The output file name with path or null for default." uid="6" />
(9)          <param name="testdelay" paramType="java.lang.String" comment="A
     test milliseconds delay string to allow manual error to be introduced or
     null for default of none." uid="7" />
(10)       </method>
(11)     </node>
(12)  </etd>
```

Notes

- Line 1: You can include XML comments in an **.xsc** file, just as you can in any other XML string. As you can see, an XML comment is different from the **comment** attribute, which is specific to the e*Gate ETD construct and is intended to help the end user understand and document the ETD.
- Line 2: When designing custom ETD e*Ways, it is standard to re-use the ETD name as its type. Do not use a standard type name (such as "DB" or "IDOC") for an ETD that is not based on that standard.
- Line 2: For the <etd> entity's **uid** attribute (which is required, just as it is for all other entities), it is customary to assign the special value "0".
- Lines 2 through 9: You can use any **uid** assignment scheme you want, provided that every **uid** value is different from all others in the same file.
- Line 3: In the <javaProps> entity, the **package** and **class** attributes specify the fully qualified Java class name of the Java class that implements the root node of the **.xsc** file. For example, if class="Z" and package="x.y", the file name is "x/y/Z.java". For complete details on rules for specifying package and class names, see **"The <javaProps> Entity" on page 292**.
- Line 3: In the <javaProps> entity, the boolean **codeAvailable** attribute provides a quick way of keeping track of whether the ETD currently matches its compiled state. If **codeAvailable**="false", the ETD is "dirty"; in other words, there are changes to it that are not reflected in the mostly recently compiled version.
- Line 4: If the <etd> entity has a **name** attribute defined, the **.xsc** file must have exactly one <node> entity whose name matches the name of the <etd> entity. This special node is called the *root node*, and it is this node that is displayed in the center pane of the ETD Editor when the ETD is first loaded. If the <etd> entity directly contains other <node> entities in addition to this root node, the Editor displays the other direct-children nodes as internal templates.
- Line 5: The **signature** attribute of the <method> entity has a special syntax that allows you to compactly record the aspects of a signature that render it unique for a particular method, such as the data types of the parameter list and the return value. For complete details, see **"Method Signature Syntax" on page 304**.
- Lines 5 through 9: For <method> and <param> entities, you should take a special effort to supply clear and meaningful comments to help end users understand the purpose and function of each method and its parameters. A good example is shown in the comment for the parameter named "testdelay" in line 9.

You can include special characters in comments, such as \ (backslash) or " (double quotations), by putting them into “normal safe” form—in other words, using the Unicode convention **\u***xxxx* to encode escape characters.

7.4 Notes, Tips, and Caveats

Metadata Representation

The XSC format uses only entities and attribute strings for storing its metadata; it does not use PCDATA or CDATA text, and does not use XML comments or externally defined entities.

Character Sets and Encodings

.xsc files are implicitly UTF-8 encoded, but restricting them to flat ASCII provides maximum compatibility with existing XML tools. All generic identifiers and attribute names are plain ASCII, and the case convention is **thisCaseStyle**—in other words, lowercase for the first component, initial-uppercase for the second and subsequent components, and without underscores.

Characters that have some special meaning in XML or cannot be displayed must be encoded in Unicode. A character outside the range U+0021 through U+007E, or one that is special to XML (such as & < > \ ") is represented as **\u** followed by its Unicode value written as four hexadecimal digits (case irrelevant). This encoding is called the “normal-safe” form, since it will not be affected by attribute normalization. However, it must be explicitly encoded and decoded when going between XSC and the internal representation of such an attribute as a string of Unicode characters. For example, the normal-safe form of this string—**a b** (letter **a**, space, space, letter **b**)—would be **"a\u0020\u0020b"**. XML also allows characters such as & and > to be escaped using the form '&amp;' and '&lt;'.

Table 6 contains the list of characters and their XML forms.

Table 6 Escape Codes for Special Characters

Unicode character	XML form
"&"	"&amp;"
"<"	"&lt;"
">"	"&gt;"
other U+0020 through U+007E	the character itself
anything else	"&#xnnnn" (where nnnn is the hexadecimal Unicode representation of the character)

Chapter 8

Developing an e*Way Using ETD Builder Components

This chapter explains the purpose and function of ETD builders, explains the key aspects of some typical ETD builder code and API calls, and explains the purpose and function of the four components required to develop an e*Way:

1. A prepackaged **run-time environment** for the e*Way: <*eWayName*>**rt.jar**
2. **e*Gate deployment files**, including a **.def** file to define configuration parameters and default values that allow end users to create and save custom configurations, and an **.ini** file to make the e*Way Connection visible to e*Gate.
3. A **back-end converter** for the e*Way's ETD builder: <*eWayName*>**ETDbuilder.jar**
4. A **front-end wizard** for the ETD builder.

Note: *The ETD builder component can only be used with version 4.5.2 of e*Gate or later. If you are using e*Gate version 4.5.1, you must use the command line, not the ETD builder wizard, to build the ETD. For details on building the ETD from the command line, see* **"Creating and Deploying an ETD by Command-Line Interface" on page 98**.

8.1 Overview

e*Gate supplies a wide variety of ETD builder wizards that allow end users to create an ETD that is tailor-made for a particular e*Way. For example, users of the Jacada e*Way can create an ETD using the JacadaWizard, and users of a variety of databases can create ETDs using the DBWizard.

When you supply a custom e*Way Connection for an external system, you can also provide an ETD builder that is tailor-made for that external system. The part of the builder that communicates with the external system, models its metadata, and outputs Java and XSC code is called the *back-end*; the wizard is the *front end*.

Note: *End users running e*Gate version 4.5.2 must install an ESR before they can use builder wizards created using this kit. For details, see the* ***Readme.txt*** *file on the e*Gate Integrator Installation CD-ROM (...\setup\addons\gmeek\Readme.txt).*

You can supply a fully functional e*Way without a front-end wizard, but doing so requires your end users to use a command-line interface to invoke the builder and install the ETD. For more information on using the command-line interface, see **“Task 6: Creating and Registering the ETD Using the Command Line” on page 148**.

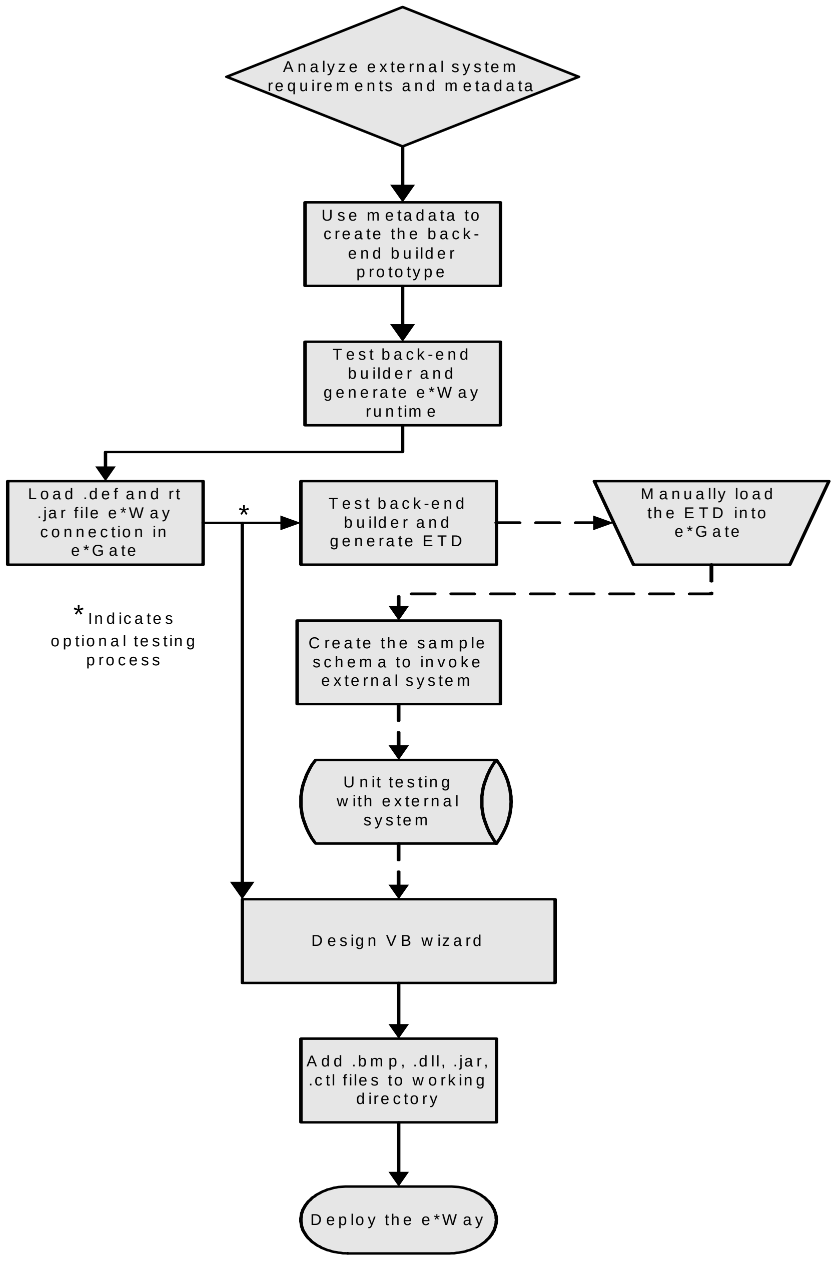
To provide a front end, you can create the ETD builder wizard using either heavyweight Visual Basic or lightweight Visual Basic. Both approaches are discussed in this chapter, and the sample e*Way instructions in **Chapter 9** provides step-by-step procedures for each approach.

For best practices about the ETD builder, see **“Working With the Back-end Builder” on page 252**.

8.2 ETD Builder Development Process Overview

Figure 23 shows the process of creating an ETD builder and lists the chapters that correspond to each step in the ETD builder development process.

Figure 23 ETD Builder Development Process


Analyze external system requirements and metadata
Use metadata to create the back-end builder prototype
Test back-end builder and generate e*Way runtime
Load .def and rt .jar file e*Way connection in e*Gate
*
Test back-end builder and generate ETD
Manually load the ETD into e*Gate
* Indicates optional testing process
Create the sample schema to invoke external system
Unit testing with external system
Design VB wizard
Add .bmp, .dll, .jar, .ctl files to working directory
Deploy the e*Way

- See **Chapter 5**
- See this chapter
- See this chapter
- See **Chapter 9**
- See **Chapter 9**
- See **Appendix C**
- See this chapter
- See **Chapter 9**
- See **Chapter 13**

8.2.1 ETD Builder Components

To develop an e*Way with an ETD builder component, you must obtain the metadata that describes the objects of the external system. Once this is accomplished, the ETD builder can create the ETD from the metadata that corresponds to the external system.

For example, the e*Gate end user may want to use an ETD representing an Account object in a billing system. In order to represent this object as an ETD, the e*Way developer must obtain the metadata describing the objects of the billing system.

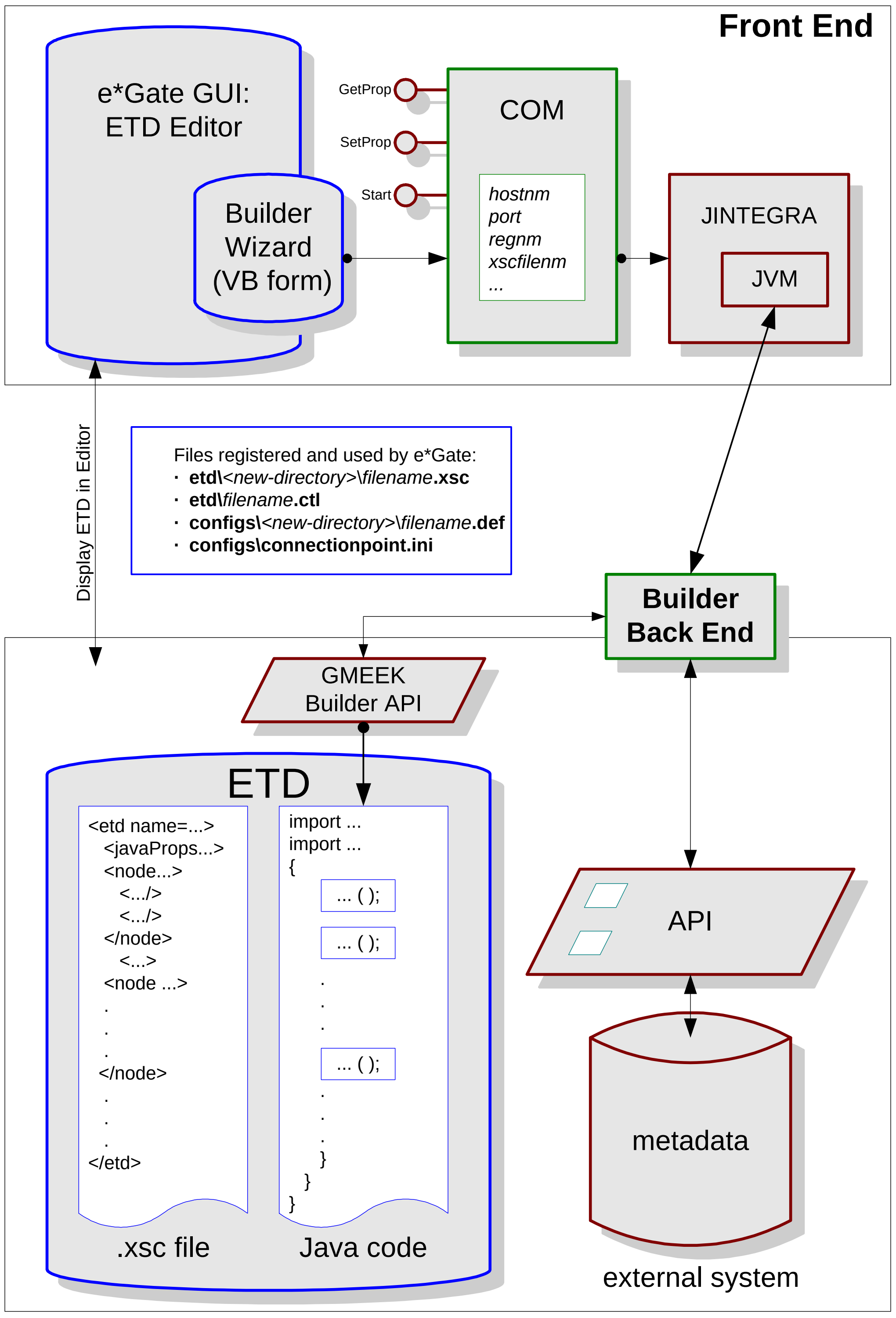
An ETD builder consists of:

- A component that obtains information about the objects defined by the e*Gate end user.
- A component which obtains the metadata from the external system and translates the metadata to the ETD.

8.2.2 ETD Builder Architecture

Figure 24 shows the ETD builder components and their relationships.

Figure 24 ETD Builder Architecture



8.2.3 What the ETD Builder Does for End Users

When end users of your e*Way run the ETD builder you supply, it does the following:

- It determines the type of input required from the user and, through the front-end wizard, prompts the user to supply all necessary information.
- If required, it creates new directories that are specific to the ETD.
- It creates an ETD-specific **.xsc** file. This makes the ETD visible to the ETD Editor.
- It creates an ETD-specific **.java** files, compiles them, and writes them to a **.jar** file.
- It creates an ETD-specific **.ctl** file. This makes the ETD available to the Collaboration Rules Editor.
- It creates an ETD-specific **.def** file. This defines default configuration parameter values and allows the end user to create **.cfg** files using the Configuration Editor.
- It runs the **installEWAY** automation command file. This modifies the **connectionpoint.ini** file (which allows the e*Way Connection to be visible to the e*Gate Configuration Editor), and it registers the new and modified files with the e*Gate Registry.

8.2.4 How an ETD Builder Operates

When the end user initiates the creation of a new ETD, the ETD Editor invokes the WizardManager. The WizardManager reads the e*Gate Registry's **addon.log** file and uses the information in that file to retrieve a list of wizards that have been installed by the end user's Participating Host, and to download the applicable wizard icons from the e*Gate Registry to the local machine.

The ETD Editor displays these wizard icons in the **New Event Type Definition** dialog box. When the user selects and launches a wizard, the WizardManager downloads the corresponding wizard DLL from the e*Gate Registry (unless it was already downloaded) and instantiates a wizard object that implements the ***Wizard*** interface.

To pass information such as username, port, and password, the ETD Editor uses the wizard's **setProperty()** method. Next, the Editor invokes the wizard's **start()** method.

The wizard collects and validates information from the user, interacts with the Registry, and calls the back-end builder. The back-end builder creates the **.xsc** and **.jar** files for the ETD. After the files are built, the wizard invokes the e*Gate Registry API to install them to e*Gate. Finally, the wizard's **start()** method returns control to the ETD Editor.

At this point, the ETD Editor can invoke the wizard's **getProperty()** method to learn further information about the wizard's end result. Finally, the ETD is displayed in the five-panel window, and the user can view or modify it using the ETD Editor.

The remainder of this chapter focuses on the purpose and function of the four e*Way components you must provide to your end users:

1. **The e*Way Run-Time Environment** on page 95.
2. **e*Gate Deployment Scripts** on page 96.
3. **Back-end Converter for the ETD Builder** on page 96.

4 **Front-end Wizards for the ETD Builder** on page 99.

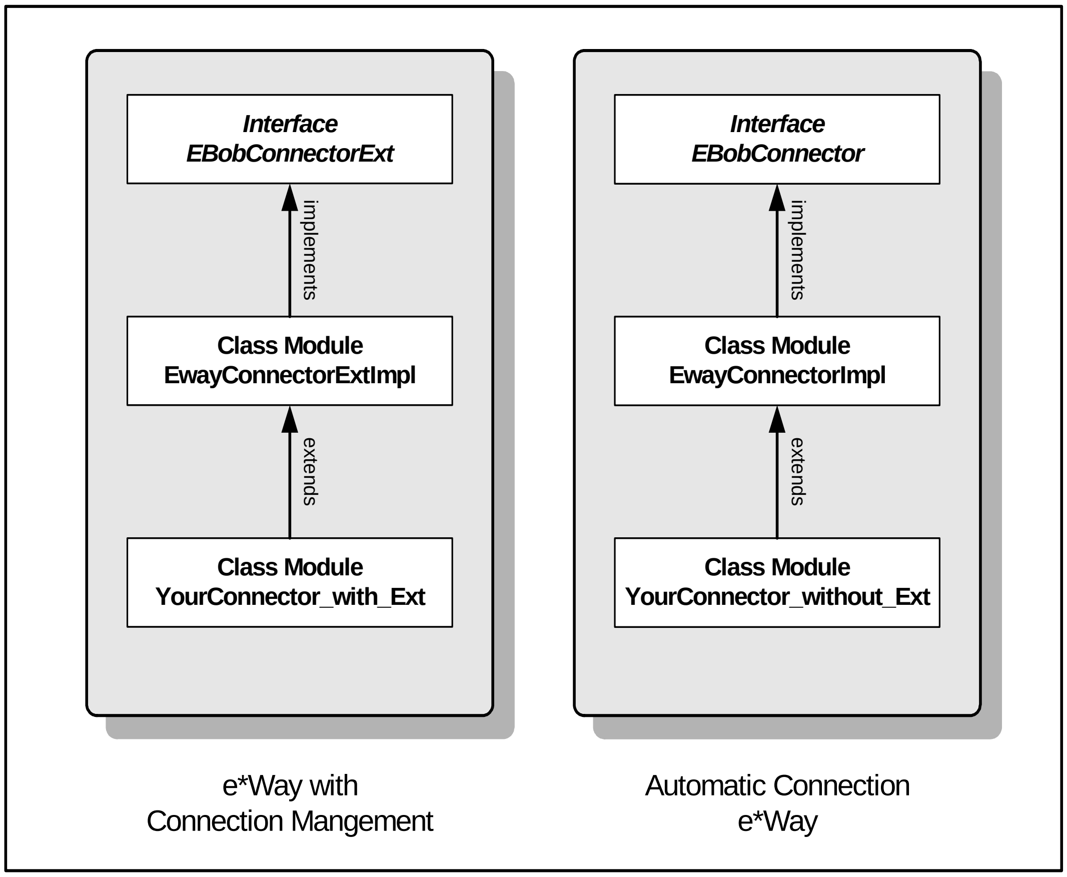
8.3 The e*Way Run-Time Environment

The run-time environment is a **.jar** file that packages the e*Way's **Connector** classes. The **Connector** classes contain the code that handles connections with the external system. Using the builder API, you can choose between two types of handling connections with the external system:

- Connectors that support Connection Management.
- Automatic connectors, which do not support Connection Management.

These connector types are shown in Figure 25.

Figure 25 Connector Types: Connection Management and Automatic Connection



- For e*Gate version 4.5.2 or later, always use Connection Management in your connectors. In practice, this means implementing the **EBobConnectorExt** interface.
- For e*Gate version 4.5.1, always use Automatic Connection in your connectors. In practice, this means implementing the **EBobConnector** interface.

The APIs provided with the kit create the code for **EwayConnectorExtImpl** or **EwayConnectorImpl**, and you normally will not need to modify them directly. Instead, you modify the skeleton connector code generated, adding appropriate logic for connecting to your external system. In this example, you would add the logic either to **YourConnector_with_Ext** (for an e*Way that will run on e*Gate version 4.5.2 or later, or else to **YourConnector_without_Ext** (for an e*Way that will run on e*Gate version 4.5.1).

For a step-by-step procedure for creating a run-time environment for the sample e*Way, see **Task 3: Building the e*Way and e*Way Connection** on page 125.

8.4 e*Gate Deployment Scripts

Along with the e*Way runtime **.jar** file, the builder API generates deployment scripts which are used to commit the e*Gate runtime **.jar** to the e*Gate Registry. The builder generates the scripts listed in Table 7.

Table 7 e*Gate Deployment Scripts

File Name	Description
connectionpoint.ini	Contains extra lines to append to **connectionpoint.ini** in the e*Gate Registry.
*<eWayName>***.def**	Contains the connection configuration parameters.
*<eWayName>***wizard.ctl**	Contains a list of file names used by the builder wizard.
stcew*<eWayName>***.ctl**	Contains a list of files to be loaded from the e*Way working directory to the e*Gate Registry.
*<ETDName>***etd.ctl**	This file is not generated by default. To manually load the ETD to the Registry, you must specify this file from the command line to invoke the back-end builder.
ETDWizards.ini	Contains extra lines to append the e*Gate **ETD Wizards.ini** to the e*Gate Registry.
addon.log	Contains extra lines to append the **addon.log** to the e*Gate Registry.

8.5 Back-end Converter for the ETD Builder

As a developer of an ETD builder, you are responsible for supplying its back-end converter, which includes:

- All necessary base classes.
- Java code for *modeling* the metadata of the external system. (End users populate the model when they run the builder.) The modeling is done using the *builder API.*
- The files required to integrate the builder into the end user's e*Gate system.

- A way of registering the files with the end user's e*Gate system.

If you expect end users to build ETDs through a command-line interface, you will need to supply them with the information they need for their system, such as directory and file names and instructions for customizing, compiling, and running various scripts that you furnish. These scripts are generated by the back-end converter.

For a step-by-step procedure for creating a back-end converter for the sample e*Way, see **Task 2: Creating the Back-end for the ETD Builder** on page 124.

8.5.1 Understanding the Builder API

Building a back-end requires you to have good knowledge of the builder API and the metadata in the external system. The samples are meant to reinforce your knowledge of the builder API and help you understand design patterns for a builder using APIs. For example:

- The sample discussed in **Chapter 9** is designed so that the run-time builder and the ETD builder are both in the **com.stc.eways.samples.gmeek.builder.apiDemo** class.
- In the sample code file RmiAccountTester.java, the **installEway()** method becomes the run-time builder, and the **compile()** methods become the ETD builder. See the source code listing of **GmeekDemoBuilder.java** on page 112.

The following subsections provide examples of typical design patterns for a builder using APIs.

To create a root node and specify connector properties

```
(1)    // The following call creates a new root node.
(2)    GmeekETDRootNode  aEtd= new GmeekETDRootNode(etd_nm,xscfname,etd_pkg_nm);

(3)    // The following call informs the root node of the
       // type of connector.
(4)    GmeekConnectorModel aCnctr = aEtd.getConnector(
                           EWAY_CONNECTOR_NAME,EWAY_CONNECTOR_TYPE,EWAY_PACKAGE_NAME);

(5)    // The following calls tell the connector model which
       // connection parameters to use. Repeat everything that
       // you specified for the connector in the installEWAY method.
(6)    aCnctr.setConnectionProperty("Rmi","Host","localhost");
(7)    aCnctr.setConnectionProperty("Rmi","Port","11990");
(8)    aCnctr.setConnectionProperty("Rmi","Name","RmiDemoSvr");
```

To create a internal template with subnodes

```
(1)    // The following call adds an internal template.

(2)    GmeekETDNode TheValueNode = aEtd.addInternalTemplate("TheValue4");

(3)    // The following calls add subnodes to this internal template.

(4)    TheValueNode.addSimpleTypeUsrField("stringValue",
                           GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

(5)    TheValueNode.addSimpleTypeUsrField("shortValue",
                           GmeekTreeNode.TreeNodeType.FIELD_TYPE_INT);

(6)    TheValueNode.addSimpleTypeUsrField("longValue",
                           GmeekTreeNode.TreeNodeType.FIELD_TYPE_LONG);
```

```
TheValueNode.addSimpleTypeUsrField("intValue",
                    GmeekTreeNode.TreeNodeType.FIELD_TYPE_INT);

TheValueNode.addSimpleTypeUsrField("doubleValue",
                    GmeekTreeNode.TreeNodeType.FIELD_TYPE_DOUBLE);
```

To use an internal template

```
// The following calls add a repeating subnode (Account Reps)
// to the root node, ...
GmeekETDNode anode = aEtd.addInnerNode("ACCOUNT_REPS");
anode.setMinMaxOccur(1,GmeekTreeNode.TreeNodeOccurrence.UNBOUNDED);

// ... but this repeating subnode uses the internal template.
anode.setInternalTemplateName("TheValue4");
```

To add a repeating subnode

```
// The following calls add a repeating subnode (Shipping Addr)
// to the root node.

GmeekETDNode ShippingAddrNode=aEtd.addInnerNode("SHIPPING_ADDR");

ShippingAddrNode.setMinMaxOccur(
                 1,GmeekTreeNode.TreeNodeOccurrence.UNBOUNDED);

ShippingAddrNode.addSimpleTypeUsrField(
      "SHIP_ALIAS",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

ShippingAddrNode.addSimpleTypeUsrField(
       "SHIP_NAME",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

ShippingAddrNode.addSimpleTypeUsrField(
    "SHIP_ADDRESS",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

ShippingAddrNode.addSimpleTypeUsrField(
   "SHIP_ADDRESS_2",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

ShippingAddrNode.addSimpleTypeUsrField(
   "SHIP_MAIL_STOP",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

ShippingAddrNode.addSimpleTypeUsrField(
        "SHIP_CITY",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

ShippingAddrNode.addSimpleTypeUsrField(
        "SHIP_STATE",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

ShippingAddrNode.addSimpleTypeUsrField(
        "SHIP_ZIP",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
```

8.5.2 Creating and Deploying an ETD by Command-Line Interface

The ETD builder component can only be used with e*Gate version 4.5.2 or later. Users of e*Gate version 4.5.1 must use the command line to build the ETD. This section explains how you (and your end users, if you do not supply a front end) can set up a script that:

1. Defines an appropriate classpath.
2. Compiles the **.java** files using this classpath.
3. Packages the results into an appropriately named **.jar** file.

4. Copies the files to an accumulator location containing e*Gate deployment files.
5. Loads the ETD to the e*Gate Registry based on the generated **.ctl** file by:
   - A Using the back-end builder with the option `<etdName>.ctl=Yes` to generate the **.ctl** file.

Important: *You must generate the **.ctl** file. The **.ctl** file contains a list of the **.xsc**, and **.jar** files, and the location in the e*Gate registry to commit those files.*

   - B Using the following **stcregutil** command:

```
stcregutil -rh host-name -rs schema-name -un user-name
-up password -fc . -ctl etdName.ctl
```

This registers the ETD (the **.xsc** and **.jar** files) into e*Gate.

stcinstd

If your Participating Host and Registry Host are running on different computers, you must have `stcinstd` running against the schema and Registry Host in use while running the `stcregutil` command.

To run *stcinstd* against the schema and the Registry Host

1. Type the following text at the command line:

```
stcinstd -rh <host> -rs <schema> -un <username> -up <password> -ss
```

2. Press **ENTER**.

Note: *In the **stcinstd** command line as shown, the flag **-ss** is optional and indicates to run the host as a service.*

For a complete explanation on using the e*Gate command line, including the **stcinstd** command, see the *e*Gate Integrator System Administration and Operations Guide.*

For more information on building the ETD from the command line, see **Task 6: Creating and Registering the ETD Using the Command Line** on page 148.

8.6 Front-end Wizards for the ETD Builder

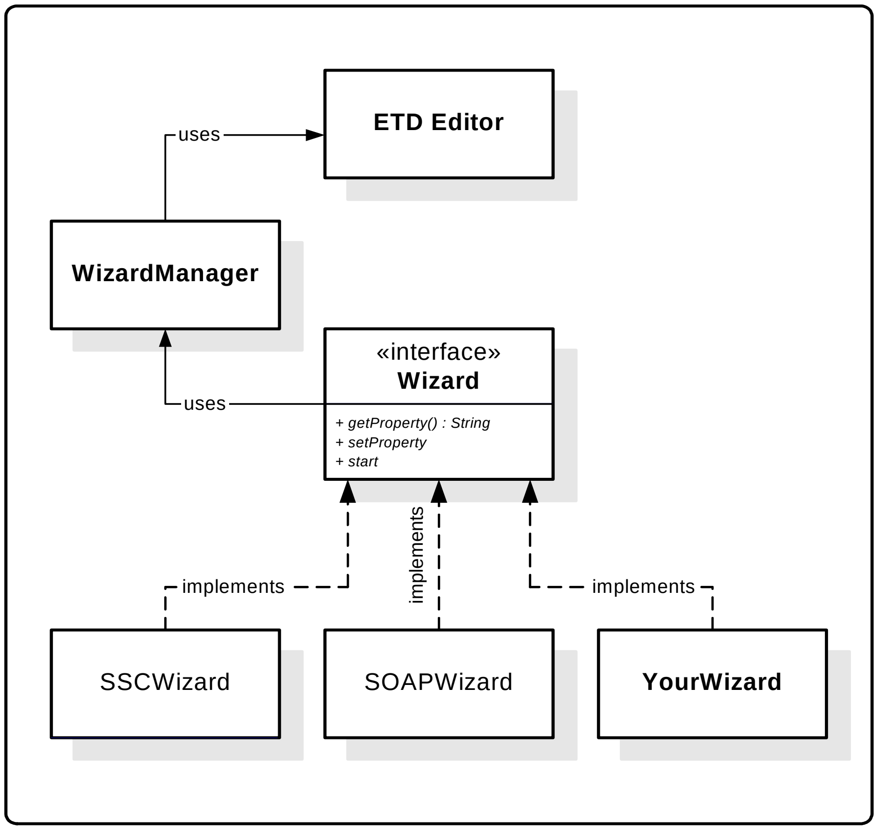
Important: *If you provide a builder wizard, you must inform users about the ESR required to run the wizard on an e*Gate version 4.5.2 system.*

8.6.1 Overview of ETD Builder Wizards

An ETD builder wizard is a self-contained application that is called by the e*Gate ETD Editor (a Visual Basic application). It uses JINTEGRA (the Visual Basic-Java bridge supplied by Oracle) to bridge between Visual Basic and Java components, and also uses the Java API to invoke the back-end converter. **Figure 24 on page 93** shows the overall architecture supporting the operation of an ETD builder wizard.

Figure 26 diagrams the relationship between the ETD Editor and the builder wizards.

Figure 26 Relationships Between the ETD Editor and the Builder Wizards



The ETD Editor invokes the wizard through three COM interfaces:

- The **SetProp** interface is used by the ETD Editor to communicate property names and values to the wizard.
- The **GetProp** interface is used by the ETD Editor to glean property names and values that the wizard has set.
- The **Start** interface causes the wizard to begin running.

The **SetProp** and **GetProp** interfaces use some standard properties as shown inTable 8:

Table 8 Standard Properties for the SetProp and GetProp COM Interfaces

Property Name	Purpose
▪ user name ▪ port ▪ password	Together, these three properties allow the wizard to communicate with the e*Gate Registry.

Table 8 Standard Properties for the SetProp and GetProp COM Interfaces

Property Name	Purpose
▪ .xsc file path	This property tells the wizard where to put the generated **.xsc** and **.jar** files.

8.6.2 Using Heavyweight or Lightweight Visual Basic to Create an ETD Builder Wizard

The ETD builder wizard is written in heavyweight or lightweight Visual Basic, according to your preferences and resources. However, you cannot use both. It is recommended that you decide on the type of Visual Basic and use that type throughout the entire process of developing an ETD builder wizard. If you have a schema that used one kind of wizard to create an ETD, you will not be able to create a new ETD or regenerate an old ETD using the new type of wizard.

There is a slight difference in architecture between the two types of Visual Basic. Compare **Figure 27 on page 102** to **Figure 28 on page 105**.

Traditionally, ETD builder wizards are developed in heavyweight Visual Basic. This approach is presented in this chapter; see **"Using Heavyweight Visual Basic to Create an ETD Builder Wizard" on page 102**.

Developers who are less familiar with Visual Basic (or prefer Java) can instead develop the wizard using lightweight Visual Basic. Lightweight Visual Basic minimizes the amount of Visual Basic coding required. This approach is also presented in this chapter; see **"Using Lightweight Visual Basic to Create an ETD Builder Wizard" on page 104**.

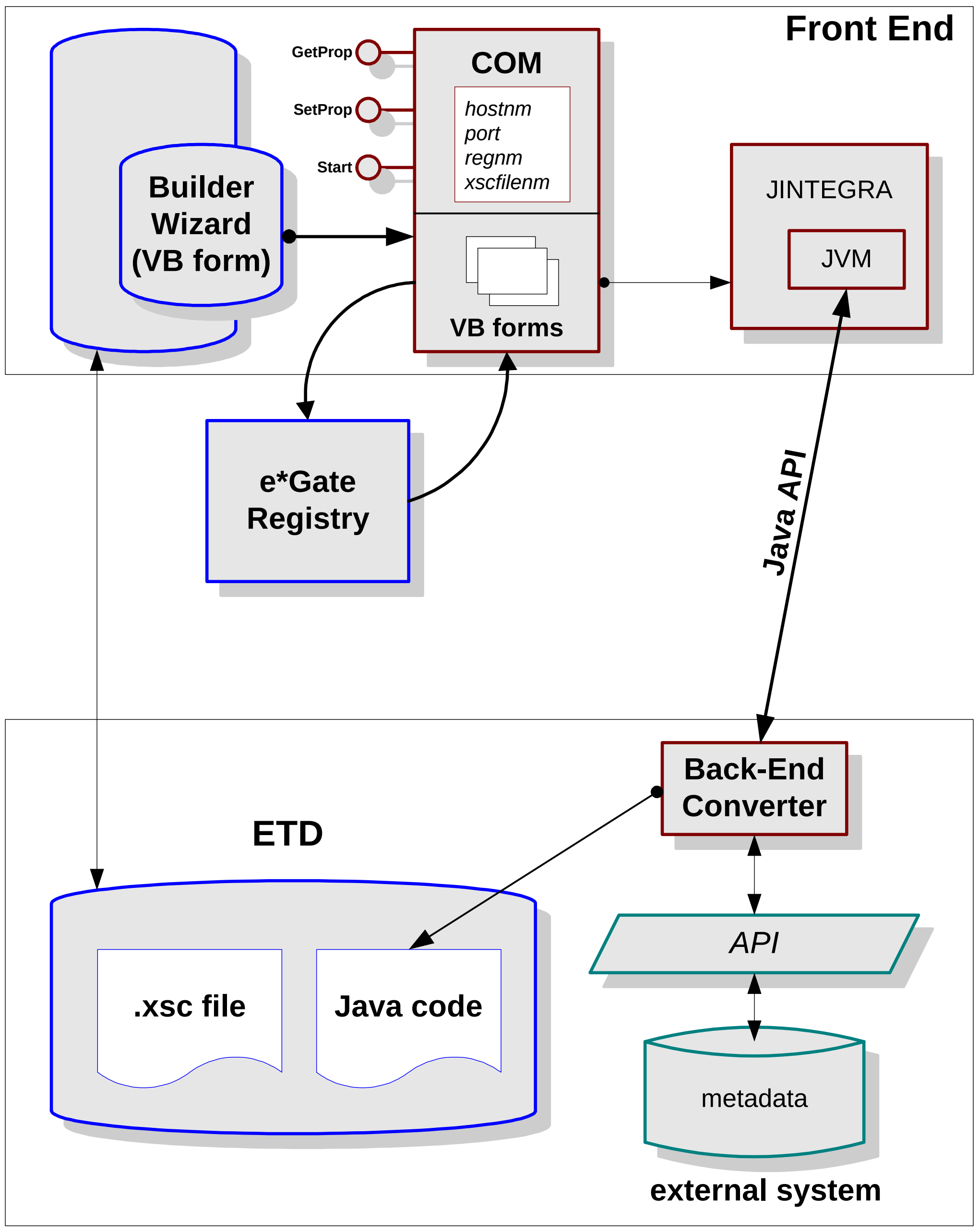
Regardless of the Visual Basic type you use, after you have created the ETD builder wizard, you must create the files that deploy the wizard to e*Gate and validate the results; see **"Deploying and Validating an ETD Builder Wizard" on page 108**.

For a step-by-step procedure for creating front-end wizards for the sample e*Way, see **"Task 4: Creating and Deploying an ETD Builder Wizard" on page 132**.

Using Heavyweight Visual Basic to Create an ETD Builder Wizard

Figure 27 illustrates the architecture of a wizard written in heavyweight Visual Basic.

Figure 27 ETD Builder Wizard Using Heavyweight Visual Basic



Writing your ETD builder wizard using heavyweight Visual Basic gives it a similar look and feel to the builder wizards supplied by Oracle. With this approach, you can capture user input, such as the root node name and package name for the ETD, and store those parameters as Visual Basic variables.

Using heavyweight Visual Basic allows you to quickly create a new wizard by re-using the Visual Basic code from the wizard template supplied with the kit, such as:

- **Forms:** frm_RegistryFileDialog, frmWizard;
- **Modules:** modConstants, STC_Global_Functions;
- **Class Modules:** class_eGateRegistry, Converter, STC_Jvm.

In the class module **Converter**, the public functions **GetProp()**, **GetPropKeys()**, **SetProp()**, and **Start()** are implemented by all wizards. These are the common COM interfaces that the ETD Editor uses to communicate with wizards. In all cases, the last line in the **Start()** method launches the wizard, as shown below:

```
frmWizard.Show vbModal
```

Registry and Classpath

Because the heavyweight Visual Basic approach requires handling the Registry calls from within the wizard, **class_eGateRegistry** is provided to allow you to obtain an object reference to the e*Gate Registry.

Each ETD builder requires you to set up a specific classpath. To obtain the classpath data, the wizard must query the Registry and read the corresponding **.ctl** file in the **etd/** directory. For example, in the Chapter 9 sample, this file is **etd/gmeekdemoewaywizard.ctl**.

This **.ctl** file, which was previously stored in the Registry as part of the Generic Multi-Mode e*Way Extension Kit e*Way installation, contains all the **.jar** information for the back-end to run. The wizard then builds a classpath string containing the **.jar** files specified in the **.ctl** file. After building the classpath, the wizard invokes a JVM to which the classpath is provided so that the back-end can run correctly. When the ETD builder returns, the corresponding **.xsc** file is created, and are displayed by the ETD Editor.

Note: ***.ctl*** *file names must be lowercase.*

Except for the classpath building and the hand-off to the JVM, very little handshaking is required among the ETD Editor, the wizard, and the back-end. The algorithm for parsing the **.ctl** file and building the classpath string is provided in the sample. It is recommended that you reuse the algorithm provided.

Invoking the Back-end Builder

A heavyweight Visual Basic wizard requires JINTEGRA to invoke the back-end Java builder.

To invoke the back-end builder, do the following:

1. **Private m_objJvm As New STC_Jvm**: This creates a new STC_Jvm object. All wizards have this surrogate STC_Jvm class.
2. **lRet = m_objJvm.SetClasspath(m_classPath)**: This sets up the JVM classpath so that the back-end can run properly.
3. **lRet = m_objJvm.Start(JINTEGRACLASS)**: This starts the underlying JVM, through ExecCmdNoWait().

4. **objGmeek = m_objJvm.GetJavaClass(GMEEKBACKENDETDBUILDERCLASS)**: This causes the builder classes in the back-end to be loaded to the JVM.

5. **objGmeek.compile strEtdName, "ACCOUNT", m_strXSCFile, strPkgName**: This calls methods specific to the back-end, generates the **.xsc** and **.jar** files, and places them in the designated directory. No complicated data structures are passed over JVM; instead, the **.xsc** and **.jar** files for the ETD are stored in a pre-arranged location for the ETD Editor to display.

ActiveX DLL

The final product is an ActiveX DLL. This DLL is copied to the working directory for your e*Way. In the **Chapter 9** sample, it is named **GmeekWizard.dll**, and it is located in the **installEWAY\GmeekDemoEway\** directory. Also, since the **.ctl** file has a list of all e*Way components that need to be loaded into eGate Registry, this DLL is referenced in the corresponding **stc**<*eWayName*>**.ctl** file. For example, the file **GmeekWizard.dll** is referenced in the **stcewgmeekdemoeway.ctl**.

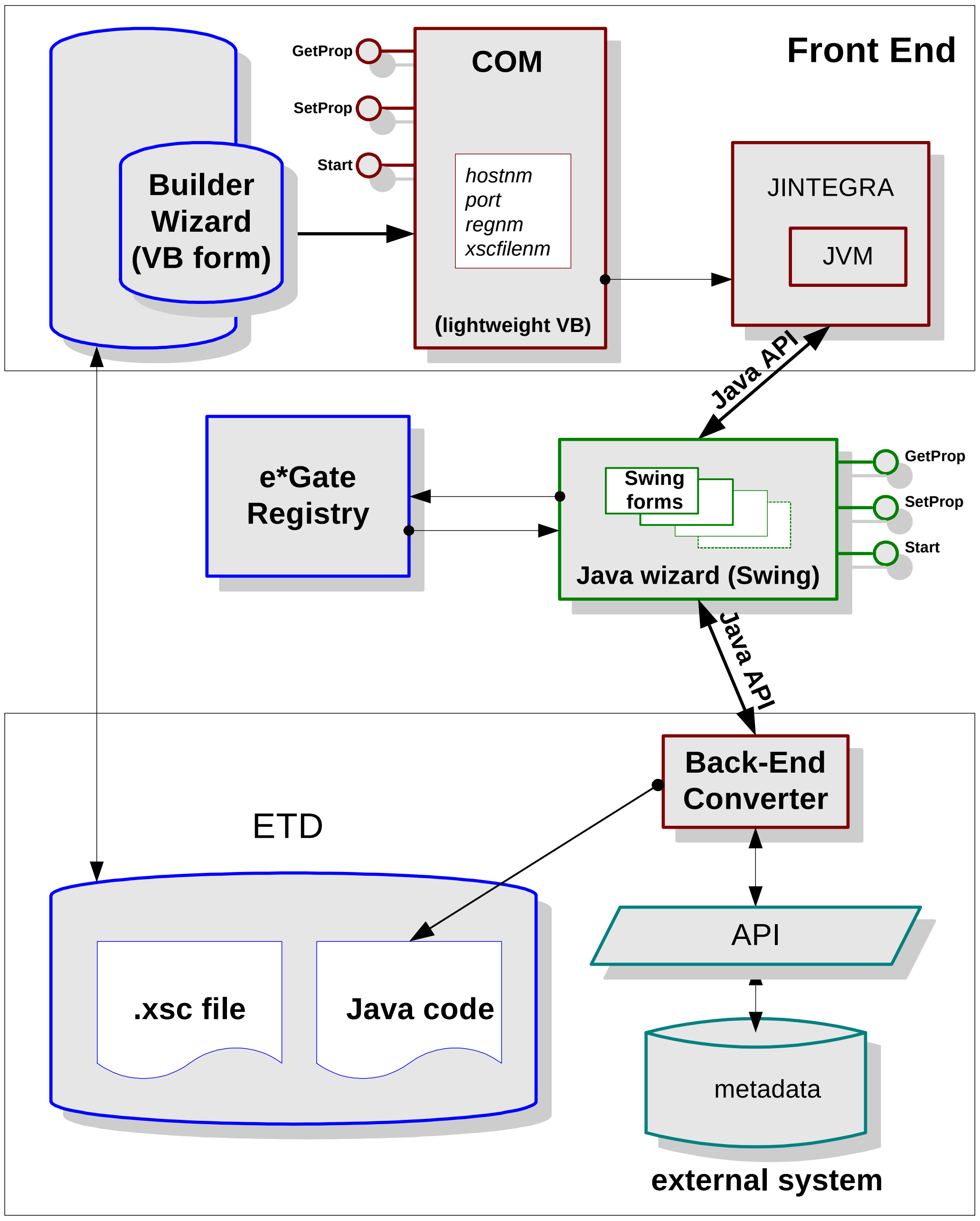
Using Lightweight Visual Basic to Create an ETD Builder Wizard

Most Java e*Way programmers prefer to code in Java as much as possible. Since the wizard is only used to capture user input, it can easily be written in Java using lightweight Visual Basic. **Figure 28 on page 105** illustrates the architecture of a wizard written using lightweight Visual Basic and Java Swing. Java Swing is used for developing Java GUI API's.

Unlike a heavy weight Visual Basic wizard, in which you invoke the back-end Java builder directly through JINTEGRA bridge, a lightweight Visual Basic wizard invokes a Java Swing GUI.

Using a lightweight Visual Basic wizard, the back-end builder in not invoked directly. Instead, the Java Swing wizard is invoked, which captures user input, validates user input, and eventually invokes the back-end builder.

Figure 28 ETD Builder Wizard Using Lightweight Visual Basic



Even if lightweight is used, a Visual Basic component is still required to communicate with the ETD Editor. The lightweight Visual Basic wizard exposes the following methods:

- **GetProp()**
- **SetProp()**
- **GetPropKeys()**
- **Start()**

Note: *Rather than storing all the properties in Java, Oracle recommends storing the properties as a Dictionary in Visual Basic. This avoids passing complicated data structure over JINTEGRA.*

When the ETD editor calls **GetProp()**, **SetProp()** and **GetPropKeys()**, the ETD editor retrieves the value from the lightweight Visual Basic wizard, not from the Java code. The property value is passed to Java only when the **Start()** method is called by the editor.

Registry and Classpath

Each ETD builder requires you to set up a specific classpath. To obtain the classpath data, the wizard must query the Registry and reads the corresponding **.ctl** file in the **etd/** directory. For example, in the **Chapter 9** sample, this file is **etd/ gmeekdemoewaywizard.ctl**.

This **.ctl** file, which was previously stored in the Registry as part of the Generic Multi-Mode e*Way Extension Kit e*Way installation, contains all the **.jar** information for the back-end to run. The wizard then builds a classpath string containing the **.jar** files specified in the **.ctl** file. After building the classpath, the wizard invokes a JVM and provides this classpath to the JVM so that the back-end can run correctly. When the ETD builder returns, the corresponding .**xsc** file is created, and is displayed by the ETD Editor.

Note: ***.ctl** file names must be lowercase.*

Except for the classpath building and the hand-off to the JVM, very little handshaking is required among the ETD Editor, the wizard, and the back-end. The algorithm for parsing the **.ctl** file and building the classpath string is provided in the sample. It is recommended that you reuse the algorithm provided.

Invoking the Back-end Builder

The following example shows how the wizard invokes the back-end builder in Visual Basic:

```
Public Function Start() As Long
...
Set objGmeek2 = m_objJvm.GetJavaClass(GMEEKBACKENDETDWIZARDCLASS)
If Not objGmeek2 Is Nothing Then
Dim KeyArray As Variant
Dim i As Long
KeyArray = m_Props.Keys
For i = LBound(KeyArray) To UBound(KeyArray)
objGmeek2.SetProp KeyArray(i), m_Props.Item(KeyArray(i))
Next i
ret=objGmeek2.Start
...
m_objJvm.Terminate
set m_objJvm=Nothing
Else
...
start=ret
...
```

The Java Swing GUI class implements the following methods:

- public void SetProp(String PropName, String PropValue);

- public int Start() throws Exception;

In the sample in Chapter 9, the lightweight Visual Basic wizard is compiled into **GmeekWizardLight.dll**, and the Java Swing portion is compiled into **GmeekWizard.jar**. **GmeekWizardLight.dll** invokes **SetProp()** and **Start()** in **GmeekWizard.jar**.

Append the following lines to the **ETDWizards.ini** file in the e*Gate Registry:

```
?GmeekDemoEway=GmeekDemoEway Wizard,
GmeekWizardLight.bmp,GmeekWizardLight.Converter,GmeekWizardLight.dll,
```

The last line instructs the ETD editor to launch the lightweight Visual Basic wizard.

ActiveX DLL

The final product is an ActiveX DLL. This DLL is copied to the working directory for your e*Way. In the Chapter 9 sample, it is named **GmeekWizardLight.dll**, and it is located in the **installEWAY\GmeekDemoEway\** directory. Also, since the **.ctl** file has a list of all e*Way components that need to be loaded into eGate Registry, this DLL is referenced in the corresponding **stc**<*eWayName*>**.ctl** file. For example, the file **GmeekWizardLight.dll** is referenced in the **stcewgmeekdemoeway.ctl**.

Note: *In the lightweight Visual Basic wizard, the Java wizard **.jar** file must be added to **stcew<eWayName>.ctl** so that the Java wizard can commit to the e*Gate Registry.*

8.6.3 Wizard Icons

Note: *This section addresses features only available for e*Gate version 4.5.2 or later.*

For both heavyweight Visual Basic and lightweight Visual Basic, you must create an icon file for the wizard. The icon file enables the Editor to display the icon in its list of builder wizards. You can use any tool to generate this icon, but it must be a **.bmp** file. In the sample, the icon is named **GmeekWizard.bmp**. Again, as with the DLL, since the **.bmp** file must be committed to the Registry, the corresponding **stc**<*eWayName*>**.ctl** file must contain the icon's file name.

In addition, the icon's file name must match the name in the **ETDWizards.ini** file. Each entry in **ETDWizards.ini** must follow this format:

```
<wizardName>, <wizardName>.bmp, <wizardName>.Converter
```

For example, in the sample, after building both the lightweight and heavyweight wizards, the file contains the following:

```
GmeekWizard,      GmeekWizard.bmp,   GmeekWizard.Converter

GmeekWizardLight, GmeekWizardLight.bmp, GmeekWizardLight.Converter
```

The icon's file name in the **addon.log** and **ETDWizards.ini** files must be identical. For more information on the **addon.log** file, see **"How an ETD Builder Operates" on page 94**. When the code in the sample generates the scripts, it provides a default name for the wizard DLL.

Important: *If you create the wizard DLL with a different name, you must change the name in both the* ***addon.log*** *and the* ***ETDWizards.ini*** *files.*

8.6.4 Deploying and Validating an ETD Builder Wizard

The ETD builder wizard is deployed with the e*Way runtime environment. All DLL's and **.jar** files used by the ETD builder wizard must be specified in the **stcew<eWayName>.ctl** file. These DLL and **.jar** files are copied to the e*Way working directory through the use of **installEWAY.bat**.

After installing the e*Way, the ETD builder wizard is validated by invoking the ETD Editor. The new wizard icon appears in the New Event Type Definition window. Double click the wizard icon to validate that the ETD builder launches successfully.

8.7 Sample Code for the Builder API

This section lists and describes the source code files involved in exercising the builder API, as used in Chapter 9. The sample code for the builder API is listed under GmeekDemoBuilder.java on page 112.

- **"RmiDemoSvr"**: Demonstrates using a Remote Method Invocation (RMI) server to simulate a generic external system.
- **"RmiDemoSvrIntf.java"**: The RMI remote interface determines the server functionality that a remote object is allowed to call.
- **"GmeekDemoBuilder.java"**: Sample program that demonstrates the use of the builder API.

RmiDemoSvr

A sample external system is provided in the sample code directory **builder/rmiDemoSvr**. The sample external system is an RMI server with the lookup name **RmiDemoSvr**.

```
package com.stc.eways.samples.gmeek.builder.rmiDemoSvr;
import java.rmi.*;
/** Remote interface specifying methods that must be provided by the server.
  *
  * @version $Revision: 1.1.2.1 $
  */
public interface RmiDemoSvrIntf extends java.rmi.Remote
{
  public String  sayEcho(String myName) throws RemoteException;
  public boolean createAccount(java.util.Map  data) throws RemoteException;
  public boolean deleteAccount(java.util.Map criteria) throws RemoteException;
  public boolean updateAccount(java.util.Map criteria) throws RemoteException;
  public java.util.HashMap retrieveAccount(java.util.HashMap criteria) throws
  RemoteException;
  public java.util.List retrieveAllAccountId() throws RemoteException;
}
```

RmiDemoSvrIntf.java

In the same directory, the implementation class is in **RmiDemoSvrIntf.java**. The ACCOUNT data is represented as HashMap and saved to a file named **account.data**.

```
package com.stc.eways.samples.gmeek.builder.rmiDemoSvr;

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.MalformedURLException;
import java.io.*;

/** Unicast remote object implementing RmiGmeek2Test interface.
 *
 * @version $Revision: 1.1.2.1 $
 */
public class RmiDemoSvrImpl extends java.rmi.server.UnicastRemoteObject implements RmiDemoSvrIntf
{
    File theFile = null;
    /** Constructs RmiDemoSvrImpl object and exports it on default port.
     */
    public RmiDemoSvrImpl() throws RemoteException
    {
        super();
        theFile = new File("account.data");
        try
        {
            if (!theFile.exists())
            {
                theFile.createNewFile();
            }
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }

    /** Constructs RmiDemoSvrImpl object and exports it on specified port.
     * @param port The port for exporting
     */
    public RmiDemoSvrImpl(int port) throws RemoteException
    {
        super(port);
    }

    /** Register RmiDemoSvrImpl object with the RMI registry.
     *
     * @param name     name identifying the service in the RMI registry
     * @param create   create local registry if necessary
     *
     * @throw RemoteException if cannot be exported or bound to RMI registry
     * @throw MalformedURLException if name cannot be used to construct a valid URL
     * @throw IllegalArgumentException if null passed as name
     */
    public static void registerToRegistry(String name, Remote obj, boolean create) throws
    RemoteException, MalformedURLException
    {
        if (name == null)
        {
            throw new IllegalArgumentException("registration name can not be null");
        }

        try
        {
            Naming.rebind(name, obj);
        }
        catch (RemoteException ex)
        {
            if (create)
            {
                Registry r = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
                r.rebind(name, obj);
            }
            else
            {
                throw ex;
            }
        }
    }

    /** return a string saying hello back to the client
     *
     */
    public String sayEcho(String myName) throws RemoteException
    {
        return "\nHello "+ myName + "!!\n";
    }

    /** Create an account object in the output file and add it to the list
     *  of accounts
     *
     * @param  data  map of accounts
     */
    public boolean createAccount(java.util.Map data) throws RemoteException
    {
        try
        {
            java.util.Vector allAccounts = new java.util.Vector();
            {
                FileInputStream fileinstrm = new FileInputStream(theFile);
                if (fileinstrm.available() > 0)
                {
                    ObjectInputStream pin = new ObjectInputStream(fileinstrm);
                    allAccounts = (java.util.Vector)pin.readObject();
                    pin.close();
                }
            };

            allAccounts.add(data);
            {
                FileOutputStream fileonstrm = new FileOutputStream(theFile);
                ObjectOutputStream pout = new ObjectOutputStream(fileonstrm);
                pout.writeObject(allAccounts);
                pout.flush();
                pout.close();
            }
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            return false;
        }
        return true;
    }

    /** Delete an account object from the output file and remove it from
     *  the list of accounts
     *
     * @param  data  map of accounts
     */
    public boolean deleteAccount(java.util.Map criteria) throws RemoteException
    {
        try
        {
            FileInputStream fileinstrm = new FileInputStream(theFile);
            java.util.Vector allAccounts = new java.util.Vector();

            if (fileinstrm.available() > 0)
            {
                ObjectInputStream pin = new ObjectInputStream(fileinstrm);
                allAccounts = (java.util.Vector)pin.readObject();
                pin.close();
            }
            else
                return false;

            for(int i =0; i< allAccounts.size();i++)
            {
                java.util.HashMap account = (java.util.HashMap)allAccounts.get(i);
                String queryAccountID = (String)criteria.get("ACCOUNT_ID");
                if (queryAccountID.compareToIgnoreCase((String)account.get("ACCOUNT_ID"))
                 == 0)
                {
                    allAccounts.remove(i);
                    return true;
                }
            }
            {
                FileOutputStream fileonstrm = new FileOutputStream(theFile);
                ObjectOutputStream pout = new ObjectOutputStream(fileonstrm);
                pout.writeObject(allAccounts);
                pout.flush();
                pout.close();
            }
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
        return false;
    }

    /** Update an account object with the matching account id
     *
     * @param  data  map of accounts
     */
    public boolean updateAccount(java.util.Map criteria) throws RemoteException
    {
            boolean updated = false;
            try
            {
                FileInputStream fileinstrm = new FileInputStream(theFile);
                java.util.Vector allAccounts = new java.util.Vector();

                if (fileinstrm.available() > 0)
                {
                    ObjectInputStream pin = new ObjectInputStream(fileinstrm);
                    allAccounts = (java.util.Vector)pin.readObject();
                    pin.close();
                }
                else
                    return false;

                for(int i =0; i< allAccounts.size();i++)
                {
                    java.util.HashMap account = (java.util.HashMap)allAccounts.get(i);
                    String queryAccountID = (String)criteria.get("ACCOUNT_ID");
             If (queryAccountID.compareToIgnoreCase(
            (String)account.get("ACCOUNT_ID")) == 0)
                    {
                        allAccounts.remove(i);
                        allAccounts.add(criteria);
                        updated = true;
                    }
                }

                if(updated)
                {
                    FileOutputStream fileonstrm = new FileOutputStream(theFile);
                    ObjectOutputStream pout = new ObjectOutputStream(fileonstrm);
                    pout.writeObject(allAccounts);
                    pout.flush();
                    pout.close();
                }

            }
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
            return updated;
        }

        /** Return an account object with the matching account id
         *
         * @param  data  map of accounts
         */
        public java.util.HashMap retrieveAccount(java.util.HashMap criteria) throws
        RemoteException
        {
            try
            {
                FileInputStream fileinstrm = new FileInputStream(theFile);
                java.util.Vector allAccounts = new java.util.Vector();
                if (fileinstrm.available() > 0)
                {
                    ObjectInputStream pin = new ObjectInputStream(fileinstrm);
                    allAccounts = (java.util.Vector)pin.readObject();
                    pin.close();
                }

                for (int i =0; i< allAccounts.size();i++)
                {
                    java.util.HashMap account = (java.util.HashMap)allAccounts.get(i);
                    String queryAccountID = (String)criteria.get("ACCOUNT_ID");
                    if (queryAccountID.compareToIgnoreCase(
                    (String)account.get("ACCOUNT_ID")) == 0)
                    {
                        return account;
                    }
                }
            }
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
            return null;
        }

        /** Retrieve all accounts in the list
         *
         * @param  data  map of accounts
         */
        public java.util.List retrieveAllAccountId() throws RemoteException
        {
            java.util.ArrayList theList = new java.util.ArrayList();
            try
            {
                FileInputStream fileinstrm = new FileInputStream(theFile);
                java.util.Vector allAccounts = new java.util.Vector();
```

```
(268)
(269)               if (fileinstrm.available() > 0)
(270)               {
(271)                   ObjectInputStream pin = new ObjectInputStream(fileinstrm);
(272)                   allAccounts = (java.util.Vector)pin.readObject();
(273)                   pin.close();
(274)               }
(275)
(276)               for (int i =0; i< allAccounts.size();i++)
(277)               {
(278)                   java.util.HashMap account = (java.util.HashMap)allAccounts.get(i);
(279)                   String queryAccountID = (String)account.get("ACCOUNT_ID");
(280)                   theList.add(queryAccountID);
(281)               }
(282)           }
(283)           catch (Exception ex)
(284)           {
(285)               ex.printStackTrace();
(286)           }
(287)           return theList;
(288)       }
(289)   }
```

GmeekDemoBuilder.java

GmeekDemoBuilder.java is a sample program that demonstrates the use of the builder API.

The import statements (3) through (5) shown below are required. They import the e*Way-specific classes that are used in a typical ETD builder and the gmeek.model classes that constitute the builder API.

```
(1)     package com.stc.eways.samples.gmeek.builder.apiDemo;
(2)
(3)     import java.io.*;
(4)     import com.stc.eways.common.builder.generator.*;
(5)     import com.stc.eways.gmeek.model.*;
(6)
(7)     /** Sample builder program that uses the GMEEK builder API.
(8)      *  After this program is compiled, the resulting .class file
(9)      *  must be archived into the (eWayName)ETDBuilder.jar file.
(10)     *
(11)     * @version $Revision: 1.1.2.4 $
(12)     */
(13)    public class GmeekDemoBuilder
(14)    {
(15)
(16)        // All ETD builders should define the following variables for
(17)        // e*Way name, package name, connector name, and connector type:
(18)        //
(19)        //Generate eway for eGate452 and above
(20)        static String EWAY_NAME = "GmeekDemoEway";
(21)        static String EWAY_PACKAGE_NAME = "com.stc.eways.GmeekDemoEway";
(22)        static String EWAY_CONNECTOR_NAME = "DemoRmi";
(23)        static String EWAY_CONNECTOR_TYPE =
        GmeekConnectorModel.ConnectorType.CONNECTOR_SUPPORT_CONNECTION_MANAGER;
(24)
(25)        //Generate eWay for  eGate451 only, no connection management
(26)        //static String EWAY_NAME = "GmeekDemoEway_451";
(27)        //static String EWAY_PACKAGE_NAME = "com.stc.eways.GmeekDemoEway_451";
(28)        //static String EWAY_CONNECTOR_NAME = "DemoRmi_451";
(29)        //static String EWAY_CONNECTOR_TYPE =
        GmeekConnectorModel.ConnectorType.CONNECTOR_NO_CONNECTION_MANAGER;
(30)
(31)        public GmeekDemoBuilder()
(32)        {
(33)        }
(34)
(35)        /**
(36)         * Generates the e*Way install scripts, e*Way run-time
(37)         * Java source, and compile scripts.
(38)         *
(39)         * @param eway_install_dir  Specifies the directory to contain
(40)         *   the generated e*Way run-time source code.
(41)         *
(42)         *   Both of the e*Way run-time source files are skeleton code only.
(43)         *   You will need to modify them to implement the connection logic.
(44)         *
(45)         *   The (eWayName)ETDBuilder.jar is placed into this directory.
(46)         *   Remember that you must also put any third-party .jar files into
(47)         *   this same directory, so that the install script can pick up
(48)         *   the .jar files and commit them to the e*Gate Registry.
(49)         *
(50)         * @param gen_props  Specifies a <code>java.util.Properties</code>.
(51)         *   If any of the files to be generated already exist, and if
```

```
     *   you want to regenerate the file, you need to specify the
     *   file name as the key in gen_props (the value is always "Yes").
     *   Otherwise, any files that already exist are not regenerated.
     *
     * @throws Exception
     */
    public void installEway(String eway_install_dir,java.util.Properties gen_props ) throws Exception
    {
       File afile;
       try{
        afile = new File(eway_install_dir);
       }
       catch (NullPointerException ex)
       {
           throw new Exception("null eway working directory");
       }

        if (afile == null)
        {
            throw new Exception("eway working directory is not valid");
        }


        //if the installEWAY directory doesn't exist, create it.
       if (!afile.exists())
            afile.mkdir();

       try{
        // Start a GmeekEwayModel to store all information needed
        // to generate run-time code for the e*Way.
        //
        // EWAY_NAME and EWAY_PACKAGE_NAME are defined as global
        // static variables, to ensure that all methods in this
        // class use the same values.
        GmeekEwayModel  aEway= new GmeekEwayModel(EWAY_NAME,EWAY_PACKAGE_NAME);

        //for CVS
        aEway.setVersion("$Revision: 1.1.2.4 $");


        // The following call causes GmeekEwayModel create a connector
        // for the e*Way, specifying the connector name and type.
        // EWAY_CONNECTOR_NAME is defined as a static variable, to
        // ensure that all methods in this class use the same values.
        //
        GmeekConnectorModel aCnctr = aEway.getConnector(EWAY_CONNECTOR_NAME,EWAY_CONNECTOR_TYPE);

        aCnctr.setVersion("$Revision: 1.1.2.4 $");

        // The following calls add all packages this connector will use.
        //
        aCnctr.addImportPackage("java.util.Properties");
        aCnctr.addImportPackage("java.net.UnknownHostException");
        aCnctr.addImportPackage("com.stc.eways.util.*");
        aCnctr.addImportPackage("com.stc.common.registry.RepositoryDirectories");
        aCnctr.addImportPackage("java.rmi.Naming");
        aCnctr.addImportPackage("com.stc.eways.samples.gmeek.builder.rmiDemoSvr.*");

        // The following calls provide the connector model with all the
        // connection properties, so that it can generate skeleton code
        // in the connector source.
        // These properties are also used in generating the .def file
        // (default configuration file) for the e*Way Connection.
        //
        aCnctr.setConnectionProperty("Rmi","Host","localhost");
        aCnctr.setConnectionProperty("Rmi","Port","11990");
        aCnctr.setConnectionProperty("Rmi","Name","RmiDemoSvr");

        // The following call causes the model to emit connector source
        // code. Parameter gen_props is of type java.util.Properties:
        // - If any of the files to be generated already exists, and if
        //   you want to regenerate the file, you must specify the file
        //   name as the key in gen_props (the value is always "Yes").
        // - If you do not specify any key value in gen_props, the
        //   following method will not overwrite any pre-existing files.
        //
        aEway.emitEwayRunTime(eway_install_dir,gen_props);
       }
       catch (Exception ex)
       {
           ex.printStackTrace();
           File[] file_lst = afile.listFiles();
           for (int i = 0; i<file_lst.length;i++)
               file_lst[i].delete();
           // afile.delete();
           System.err.println("everything failed remove the eway working directory");
       }

    }

    /**
     * This method compiles the code without generating the files
     * etd.ctl, compile.sh, and compile.bat.
     *
     * Call this version of the method if you will be running the
     * builder from an ETD builder wizard.
     *
     * @param etd_nm       Specifies the root node name of the ETD.
     *                     For most ETD builders, this parameter is
     *                     input by the user.
     *
     * @param etd_typ_nm   Specifies which type of ETD: ACCOUNT,
     *                     PAYROLL, etc.
     *                     This is unique to the ETD being built
     *                     as a sample.
     *
     * @param xscfname     Specifies the full path name of the
     *                     generated .xsc file.
     *                     The .jar file will be put in the same
     *                     directory and will have the same prefix.
     *
     * @param etd_pkg_nm   Specifies the package name of the ETD.
     *                     For almost all ETD builders, this parameter
     *                     is input by the user.
     */
    public void compile(String etd_nm,String etd_type_nm,String xscfname, String etd_pkg_nm) throws Exception
    {
        this.compile(etd_nm,etd_type_nm,xscfname,etd_pkg_nm,null);
    }
    /**
     *This method will generate etd.ctl, compile,short and compile.bat files to
     *help you compile and install the etd from command line.
     *
     * Call this version of the method if you will be installing
     * the ETD from the command line.
     *
     * @param etd_nm       Specifies the root node name of the ETD.
     *                     For most ETD builders, this parameter is
     *                     input by the user.
     *
     * @param etd_typ_nm   Specifies which type of ETD: ACCOUNT,
     *                     PAYROLL, etc.
     *                     This is unique to the ETD being built
     *                     as a sample.
     *
     * @param xscfname     Specifies the full path name of the
     *                     generated .xsc file.
     *                     The .jar file will be put in the same
     *                     directory and will have the same prefix.
     *
     * @param etd_pkg_nm   Specifies the package name of the ETD.
     *                     For almost all ETD builders, this parameter
     *                     is input by the user.
     *
     * @param props        Specifies the list of files to generate
     *                     for building the ETD in command mode:
     *                     etd.ctl, compile.bat, and compile.sh
     */
    public void compile(String etd_nm,String etd_type_nm,String xscfname, String etd_pkg_nm,java.util.Properties props) throws Exception
    {
        try
        {
            System.out.println("etd_nm:"+etd_nm+" xscfname:"+xscfname+" pkg name:"+etd_pkg_nm);

            // The following call creates a new root node.
            GmeekETDRootNode  aEtd= new GmeekETDRootNode(etd_nm,xscfname,etd_pkg_nm);

            aEtd.setVersion("$Revision: 1.1.2.4 $");


            // The following call informs the root node of the
            // type of connector.
            //
            GmeekConnectorModel aCnctr = aEtd.getConnector(EWAY_CONNECTOR_NAME,EWAY_CONNECTOR_TYPE,EWAY_PACKAGE_NAME);

            // The following calls inform the connector model which
            // connection parameters to use. Repeat everything that
            // you specified for the connector in the installEWAY method.
            //
            aCnctr.setConnectionProperty("Rmi","Host","localhost");
            aCnctr.setConnectionProperty("Rmi","Port","11990");
            aCnctr.setConnectionProperty("Rmi","Name","RmiDemoSvr");

            // The following calls cause metadata to be output to a
            // text file for viewing purposes only. The output is
            // not used to build the ETD.
            //
            FileOutputStream afostrm = new FileOutputStream("Account.something");
            PrintStream aprnttsrm = new PrintStream(afostrm);

            // The following calls define the import packages for the ETD.
            aEtd.addImportPackage("java.util.*");
            aEtd.addImportPackage("java.rmi.Naming");

            // The following call adds the "retrieveAccount" method
            // to the ETD.
            //
            GmeekETDMethod aMethod =
aEtd.addMethod("retrieveAccount",GmeekTreeNode.TreeNodeAccessType.PUBLIC_ACCESS,"void");

            // The following call adds the method body.
            aMethod.appendMethodBody("try { \n" +
            " java.util.HashMap creteria = new java.util.HashMap(); \n"+
            " creteria.put(\"ACCOUNT_ID\",this.getACCOUNT_ID()); \n" +
            " java.util.Map accountMap = myConnector.getRemoteRef().retrieveAccount(creteria); \n" +
            " if(accountMap.containsKey(\"ACCOUNT_NAME\")) { \n " +
            "  String temp=(String)accountMap.get(\"ACCOUNT_NAME\"); \n" +
            "  this.setACCOUNT_NAME(temp); \n" +
            "  } \n" +
            " if(accountMap.containsKey(\"ACCOUNT_TYPE\")) { \n" +
            "   String temp=(String)accountMap.get(\"ACCOUNT_TYPE\") ; \n" +
            "   this.setACCOUNT_TYPE(temp); \n" +
            " } \n" +
            " if(accountMap.containsKey(\"ACCOUNT_REPS\")) { \n" +
            "    java.util.Vector temp=(java.util.Vector)accountMap.get(\"ACCOUNT_REPS\"); \n" +
            "     for (int i=0; i < temp.size(); i++) { \n" +
            "     this.getACCOUNT_REPS(i).setstringValue((String)temp.get(i)); \n" +
            "      } \n" +
            " } \n" +
            " if(accountMap.containsKey(\"ADDRESS\")) { \n" +
            "   String temp=(String)accountMap.get(\"ADDRESS\"); \n" +
            "   this.setADDRESS(temp); \n" +
            "  } \n" +
            " if(accountMap.containsKey(\"ADDRESS_2\")) { \n" +
            "   String temp=(String)accountMap.get(\"ADDRESS_2\"); \n" +
            "   this.setADDRESS_2(temp); \n" +
            " } \n " +
            " if(accountMap.containsKey(\"BILLING_ADDR\")) { \n" +
            "   java.util.Vector temp=(java.util.Vector)accountMap.get(\"BILLING_ADDR\"); \n" +
            "   for (int i=0; i < temp.size(); i++) { \n" +
            "     java.util.HashMap temp1 = (java.util.HashMap)temp.get(i); \n " +
            "     if(temp1.containsKey(\"BILL_ADDRESS\")) \n" +
           "       this.getBILLING_ADDR(i).setBILL_ADDRESS((String)temp1.get(\"BILL_ADDRESS\")); \n"
+
            "     if(temp1.containsKey(\"BILL_ADDRESS_2\")) \n " +
            "       this.getBILLING_ADDR(i).setBILL_ADDRESS_2((String)temp1.get(\"BILL_ADDRESS_2\"));
\n " +
            "     if(temp1.containsKey(\"BILL_ALIAS\")) \n" +
            "       this.getBILLING_ADDR(i).setBILL_ALIAS((String)temp1.get(\"BILL_ALIAS\")); \n " +
            "   } \n " +
            "  } \n" +
            " }catch (java.rmi.RemoteException ex) { \n " +
            "   throw new CollabDataException(\"Exception caught.\", ex); \n " +
            " } \n "
            );

            // The following calls add the other three methods --
            // "createAccount()", "deleteAccount()", and "updateAccount()".
            //
            aMethod =
aEtd.addMethod("createAccount",GmeekTreeNode.TreeNodeAccessType.PUBLIC_ACCESS,"void");
            aMethod =
aEtd.addMethod("deleteAccount",GmeekTreeNode.TreeNodeAccessType.PUBLIC_ACCESS,"void");
            aMethod =
aEtd.addMethod("updateAccount",GmeekTreeNode.TreeNodeAccessType.PUBLIC_ACCESS,"void");

            // The following calls add simple subnodes.
            aEtd.addSimpleTypeUsrField("ACCOUNT_NAME",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("ACCOUNT_ID",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("ACCOUNT_TYPE",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("COMPANY_NAME",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("INDUSTRY",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("CONTACT",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("ADDRESS",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("ADDRESS_2",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("CITY",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("STATE",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("ZIP",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("COUNTRY",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("PHONE",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("FAX",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("MAIL_STOP",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            aEtd.addSimpleTypeUsrField("EMAIL",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

            // The following call adds an internal template.
            GmeekETDNode TheValueNode = aEtd.addInternalTemplate("TheValue4");

            // The following calls add subnodes to this internal template.

TheValueNode.addSimpleTypeUsrField("stringValue",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

TheValueNode.addSimpleTypeUsrField("shortValue",GmeekTreeNode.TreeNodeType.FIELD_TYPE_INT);
TheValueNode.addSimpleTypeUsrField("longValue",GmeekTreeNode.TreeNodeType.FIELD_TYPE_LONG);
            TheValueNode.addSimpleTypeUsrField("intValue",GmeekTreeNode.TreeNodeType.FIELD_TYPE_INT);
TheValueNode.addSimpleTypeUsrField("doubleValue",GmeekTreeNode.TreeNodeType.FIELD_TYPE_DOUBLE);

            // The following calls add a repeating subnode (Shipping Addr)
            // to the root node.
            //
            GmeekETDNode ShippingAddrNode=aEtd.addInnerNode("SHIPPING_ADDR");
            ShippingAddrNode.setMinMaxOccur(1,GmeekTreeNode.TreeNodeOccurrence.UNBOUNDED);

ShippingAddrNode.addSimpleTypeUsrField("SHIP_ALIAS",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
ShippingAddrNode.addSimpleTypeUsrField("SHIP_NAME",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
ShippingAddrNode.addSimpleTypeUsrField("SHIP_ADDRESS",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
ShippingAddrNode.addSimpleTypeUsrField("SHIP_ADDRESS_2",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING)
;
ShippingAddrNode.addSimpleTypeUsrField("SHIP_MAIL_STOP",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING)
;
ShippingAddrNode.addSimpleTypeUsrField("SHIP_CITY",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
ShippingAddrNode.addSimpleTypeUsrField("SHIP_STATE",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
ShippingAddrNode.addSimpleTypeUsrField("SHIP_ZIP",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

            // The following calls add a repeating and optional subnode (Billing Addr)
            // to the root node.
            //
            GmeekETDNode BillAddrNode=aEtd.addInnerNode("BILLING_ADDR");
            BillAddrNode.setMinMaxOccur(0,GmeekTreeNode.TreeNodeOccurrence.UNBOUNDED);
BillAddrNode.addSimpleTypeUsrField("BILL_ALIAS",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
BillAddrNode.addSimpleTypeUsrField("BILL_NAME",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
BillAddrNode.addSimpleTypeUsrField("BILL_ADDRESS",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
BillAddrNode.addSimpleTypeUsrField("BILL_ADDRESS_2",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
BillAddrNode.addSimpleTypeUsrField("BILL_CITY",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
BillAddrNode.addSimpleTypeUsrField("BILL_STATE",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
BillAddrNode.addSimpleTypeUsrField("BILL_ZIP",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
BillAddrNode.addSimpleTypeUsrField("BILL_COUNTRY",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

            // The following calls add a optional subnode (Payment)
            // to the root node.
            //
            GmeekETDNode PaymentNode=aEtd.addInnerNode("PAYMENT");
            PaymentNode.setMinMaxOccur(0,1);
PaymentNode.addSimpleTypeUsrField("PAYMENT_NAME",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
PaymentNode.addSimpleTypeUsrField("DEPARTMENT",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
PaymentNode.addSimpleTypeUsrField("CARD_NUMBER",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            PaymentNode.addSimpleTypeUsrField("STATE",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
PaymentNode.addSimpleTypeUsrField("PMTTYPE_ID",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
PaymentNode.addSimpleTypeUsrField("PURCHASE_LIMIT",GmeekTreeNode.TreeNodeType.FIELD_TYPE_DOUBLE);
            PaymentNode.addSimpleTypeUsrField("ADDRESS",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            PaymentNode.addSimpleTypeUsrField("CITY",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
PaymentNode.addSimpleTypeUsrField("STORE_NAME",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
PaymentNode.addSimpleTypeUsrField("FIRST_NAME",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

            // The following calls add a must have subnode (Order)
            // to the root node.
            //
            GmeekETDNode OrderNode=aEtd.addInnerNode("ORDER");
            OrderNode.setMinMaxOccur(1,1);
OrderNode.addSimpleTypeUsrField("ORDER_NAME",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
OrderNode.addSimpleTypeUsrField("DEPARTMENT",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
            OrderNode.addSimpleTypeUsrField("ORDER_ID",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
OrderNode.addSimpleTypeUsrField("ORDER_PARTY_ADDRESS",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
OrderNode.addSimpleTypeUsrField("ORDER_PARTY_CITY",GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);

            // The following calls add a repeating subnode (Account Reps)
            // to the root node, ...
            //
            GmeekETDNode anode = aEtd.addInnerNode("ACCOUNT_REPS");
            anode.setMinMaxOccur(1,GmeekTreeNode.TreeNodeOccurrence.UNBOUNDED);
            // ... but this repeating subnode uses the internal template.
            anode.setInternalTemplateName("TheValue4");

            // The following calls add a repeating optional subnode (Trans Aliases)
            // to the root node, ...
            //
            anode = aEtd.addInnerNode("TRANS_ALIASES");
            anode.setMinMaxOccur(0,GmeekTreeNode.TreeNodeOccurrence.UNBOUNDED);
            // ... but this repeating subnode uses the internal template.
            anode.setInternalTemplateName("TheValue4");

            // The following calls add a nonrepeating optional subnode (OET Names)
            // to the root node, ...
            //
            anode = aEtd.addInnerNode("OET_NAMES");
            anode.setMinMaxOccur(0,1);
            // ... but this repeating subnode uses the internal template.
            anode.setInternalTemplateName("TheValue4");


            // The following calls add a nonrepeating must have subnode (Visitor Names)
            // to the root node, ...
            //
            anode = aEtd.addInnerNode("VisitorNames");
            anode.setMinMaxOccur(1,1);
            // ... but this repeating subnode uses the internal template.
            anode.setInternalTemplateName("TheValue4");

            // The following calls add a single subnode (Org Entity Types)
            // to the root node ...
            //
            GmeekETDNode orgEntityTypesNode = aEtd.addInnerNode("OrgEntityTypes");
            // ... but this repeating subnode uses the internal template.
            orgEntityTypesNode.setInternalTemplateName("TheValue4");


            // The following calls write the metadata you just entered
            // to a human-readable file.
            //
            aEtd.emitNesting(aprnttsrm);
            aprnttsrm.flush();
            aprnttsrm.close();

            // The following call takes all the ETD information that
            // was captured by the model and emits it to .java and
            // .xsc source.
            //
            aEtd.emitJavaAndXsc(props);

            // The following call compiles and archives (jars) the
            // Java source, thus generating the run-time package.
            //
            aEtd.generateRuntimePackage();

        } catch (Exception ex)
        {
            File axscFile = new File(xscfname);
            if(axscFile != null)
            {
                File xscDir = axscFile.getParentFile();
                if (xscDir != null)
                {
                    File[] allfiles = xscDir.listFiles();
                    try
                    {
                        for (int i=0;i < allfiles.length;i++)
                        {
                            allfiles[i].delete();
                        }
                    }catch (Exception f)
                    {
                        f.printStackTrace();
                    }
                }
            }
            ex.printStackTrace();
            System.err.println("remove every partialy generated file");

            // Generate exception for caller of this routine to
            // determine cause of problem.
            //
           throw (new Exception("xsc files not generated"));
        }


    } //

    public static void main(String [] args ) throws Exception
```

```
  {
    if (args.length < 2)
    {
      System.out.println("Usage: java com.stc.eways.samples.gmeek.builder.apiDemo.GmeekDemoBuilder -install <dirname> or \n");
      System.out.println("Usage: java com.stc.eways.samples.gmeek.builder.apiDemo.GmeekDemoBuilder -compile etdname etd_type_name etdxscfile etd_package_name");
      return;
    }
    if (args[0].compareToIgnoreCase("-install" ) == 0 )
    {
      GmeekDemoBuilder builder = new GmeekDemoBuilder();
      java.util.Properties props = new java.util.Properties();
      if (args.length >= 2)
      {

        for(int i=2; i < args.length;i++)
        {
          int idx = args[i].indexOf("=");
          if(idx > 0)
          {
            String key=args[i].substring(0,idx);
            String val = args[i].substring(idx+1);
            System.out.println(args[i]+" "+key+" "+val);
            props.setProperty(key,val);
          }

        }
      }
      builder.installEway(args[1],props);
    }
    else if (args[0].compareToIgnoreCase("-compile" ) == 0 )
    {
      GmeekDemoBuilder builder = new GmeekDemoBuilder();


      if (args.length >= 5)
      {
        java.util.Properties props = new java.util.Properties();
        for(int i=5; i < args.length;i++)
        {
          int idx = args[i].indexOf("=");
          if(idx > 0)
          {
            String key=args[i].substring(0,idx);
            String val = args[i].substring(idx+1);
            System.out.println(args[i]+" "+key+" "+val);
            props.setProperty(key,val);
          }

        }
        builder.compile(args[1],args[2],args[3],args[4],props);
      }
      else
        builder.compile(args[1],args[2],args[3],args[4]);



    }


  }
}


```

Chapter 9

Developing an e*Way Using the Builder API

9.1 Overview

Chapters seven through nine describe how to write the Java classes, **.xsc** files, configuration files, and deployment files required to produce an ETD which corresponds to an e*Way connection. The ETD produced in this process has predefined nodes and methods that correspond to an external system.

This chapter guides you through the steps needed to create a sample e*Way that allows end users to build ETDs dynamically, based on metadata from an external system. This sample e*Way is much more complex than the other sample e*Ways described in following chapters, because it requires you to simulate an external system, develop an e*Way Connection tailored to that system, and develop and deploy an *ETD builder*. The sample ETD builder uses the *builder API* to generate the Java source code and XSC that constitute the ETD. This architecture is illustrated in **Figure 24 on page 93**.

As noted in **Chapter 8**, you must provide your end users with these four components:

1. A prepackaged **run-time environment** for the e*Way: <*eWayName*>**rt.jar**
2. **e*Gate deployment files**, including a **.def** file to define configuration parameters and default values and allow end users to create and save custom configurations; and an **.ini** file to make the e*Way Connection visible to e*Gate.
3. A **back-end converter** for the e*Way's ETD builder: <*eWayName*>**ETDbuilder.jar**
4. A **front-end wizard** for the ETD builder.

Note: *The **front-end wizard** for the ETD builder is only available for e*Gate version 4.5.2 or later. If your e*Way will operate in e*Gate version 4.5.1, you must provide instructions for running the ETD builder from the command line. For more information, see* **"Creating and Deploying an ETD by Command-Line Interface" on page 98**.

9.1.1 Using the Sample e*Ways

This chapter describes the steps required to create a sample e*Way for versions 4.5.1 and 4.5.2 or later of e*Gate. The steps to create the sample e*Ways are the same for all versions with the following exception for e*Gate version 4.5.1:

- the file names for e*Gate version 4.5.1 end with"_451".

Upon completing the sample e*Way, you can deliver your end users an e*Way that comprises:

1. A prepackaged run-time environment for the e*Way: **GmeekDemoEwayrt.jar**. The e*Gate version 4.5.2 or later sample e*Way is named **GmeekDemoEwaySample**. Each sample e*Way's run-time **.jar** file is a package containing the classes and source for the e*Way Connection.
2. The following set of e*Gate deployment files:
   - **GmeekDemoEway.def**: A file that defines default configuration parameters for the e*Ways of the GmeekDemoEway type. The **.def** file makes parameters visible to the e*Gate Configuration Editor and allows end users to create customized configurations for the e*Way.
   - **connectionpoint.ini**: A fragment to be incorporated into the end user's **connectionpoint.ini** file that makes the e*Way Connections of type GmeekDemoEway visible to e*Gate.
   - **gmeekdemoeway.ctl**: Contains a list of files to be loaded from the e*Way working directory to the e*Gate Registry.
   - **gmeekdemoewaywizard.ctl**: Contains a list of files used by the builder wizard.

Note: ***.ctl** file names must be lowercase. The Visual Basic Collaboration Editor will not download any files that are uppercase.*

3. A back-end converter for the ETD builder: **GmeekDemoEwayETDbuilder.jar**. The back-end **.jar** file is a package containing all classes needed to build the ETD. If you do not supply an ETD builder wizard, your end users must use this **.jar** file to invoke the builder through a command-line interface.
4. Either one of two GUI programs that provide a front-end wizard for the ETD builder, built with either heavyweight Visual Basic or lightweight Visual Basic. This feature is only supported in e*Gate version 4.5.2 or later.

Important: *When running the sample schema for the first time, we do recommend that you do not modify the source files. We only recommend that you modify the source files once you have become familiar with the e*Way development process.*

Tasks for Completing the Sample e*Ways

The procedures in this chapter are organized into the following general tasks:

- **Task 1: One-Time Setup Steps** on page 122. In this task, you perform one-time steps to load and set up the product and to set up the RMI server that simulates an external system.
- **Task 2: Creating the Back-end for the ETD Builder** on page 124. In this task, you browse the sample code that exercises the builder API, and then run the compile script.
- **Task 3: Building the e*Way and e*Way Connection** on page 125. In this task, you create and package files needed by the e*Way and the e*Way Connection, and then register the files in e*Gate.

- **Task 4: Creating and Deploying an ETD Builder Wizard** on page 132. In this task, you create a builder wizard (both approaches are presented—heavyweight Visual Basic or lightweight Visual Basic) and then register the wizard with e*Gate.

Note: *Using a kit-built ETD builder wizard on e*Gate Integrator version 4.5.2 requires an ESR. Refer to the* ***Readme.txt*** *file for more information.*

- **Task 5: Testing the Wizard with a Stand-alone Visual Basic Tester** on page 145. In this task, you verify that end users of your product can do the following within e*Gate:
  - Create and configure a **GmeekDemoEway** type of e*Way Connection.
  - Use an ETD builder wizard to create one or more ETDs of type **AccountETD**.
  - Use these ETDs in a simple Collaboration Rule.
- **Task 6: Creating and Registering the ETD Using the Command Line** on page 148. In this task, you exercise the back-end only and verify that files are correctly built and registered.

Note: *If using e*Gate Integrator version 4.5.1, you are required to use the approach detailed in Task 6 to build the ETD. Tasks 4 and 5 apply only to e*Gate version 4.5.2 or later.*

- **Task 7: Testing Outside of the e*Gate Environment** on page 151. In this task, you validate the APIs outside of e*Gate.
- **Task 8: Understanding the Sample Implemented in a Schema** on page 153. In this task, you load the sample schema to see how the sample components appear and behave in the e*Gate environment.

9.2 Task 1: One-Time Setup Steps

The following sub-tasks are needed to set up your system to create the sample:

- **Pre-installation** on page 122
- **One-Time Setup Steps** on page 123
- **One-Time Setup Steps for the RMI Server** on page 124

9.2.1 Pre-installation

Ensure that your machine has Java SDK 1.6.0_19 installed on it and that you have access to files on an e*Gate Participating Host at level 4.5.1 or later.

Note: *Remember that an ESR is required to run the ETD Builder wizard on e*Gate version 4.5.2. You will do this in a later task. If using e*Gate version 4.5.1, see* **"Task 6: Creating and Registering the ETD Using the Command Line" on page 148**.

1 Locate where your Java SDK is installed. If it is not in **C:\jdk1.3.1_02\**, you will need to open certain batch files or shell scripts in the following steps and modify them so that they point to the correct Java SDK directory.

2 Create two parent directories on your machine, one to hold the Oracle-supplied **.jar** files you need to reference, and one to hold the files you will develop. The samples assume they are named as in Table 9.

Table 9 Development Directories

Directory to Create	Purpose
C:\gmeekjars\bin\java C:\gmeekjars\classes C:\gmeekjars\ThirdParty\sun	These directories hold **.jar** files supplied by Oracle that you will need to reference when creating the e*Way and ETD builder.
C:\gmeeksdk	Holds your development work in progress. Keep your code here so that it will not be prematurely mingled with run-time files.

3 Copy specific **.jar** files from your e*Gate Participating Host (**<eGate>\client**) into a subdirectory of your **C:\gmeekjars\** directory as shown in Table 10.

Table 10 Files to Copy from e*Gate Participating Host to C:\gmeekjars

e*Gate File to Copy (from <eGate>\client\)	Destination
bin\java\gnu-regexp-1.1.1.jar	C:\gmeekjars\bin\java

Table 10 Files to Copy from e*Gate Participating Host to C:\gmeekjars

e*Gate File to Copy (from <eGate>\client\)	Destination
bin\java\jcscomp.jar	C:\gmeekjars\bin\java
bin\java\xerces.jar	C:\gmeekjars\bin\java
classes**stcjcs.jar**	C:\gmeekjars\classes
classes**stcutil.jar**	C:\gmeekjars\classes
ThirdParty\sun\jta.jar	C:\gmeekjars\ThirdParty\sun
ThirdParty\sun\jms.jar	C:\gmeekjars\ThirdParty\sun

9.2.2 One-Time Setup Steps

The following steps only need to be done once, when you initially install the Generic Multi-Mode e*Way Extension Kit. If you did not do the setup steps during your initial installation, do the following steps.

1 Locate the following file on the e*Gate Integrator Installation CD-ROM:

setup\addons\gmeek\common.taz

2 Decompress **common.taz** and extract its **.jar** files into **C:\gmeekjars\**:

- classes\stcewcommonbuilder.jar
- classes\stcgmeek.jar

3 Locate the following directory on the e*Gate Integrator Installation CD-ROM:

samples\sdk\gmeek

4 Extract the files from **samples\sdk\gmeek\gmeek.taz** into **C:\gmeeksdk\**, which creates directories and files, some of which are shown in the following list:

- C:\gmeeksdk\gmeek\
- C:\gmeeksdk\gmeek\classes\
- C:\gmeeksdk\gmeek\installETD\
- C:\gmeeksdk\gmeek\installEWAY\
- C:\gmeeksdk\gmeek\testdata\

For a complete list of files and directories in **gmeek.taz**, see **Table 3 on page 26**.

5 Extract the files from **samples\sdk\gmeek\VBGmeekWizard.zip** into directory **C:\gmeeksdk\Wizards\**. You will use these files to create an ETD builder wizard if using e*Gate version 4.5.2 or later.

6 If using e*Gate version 4.5.1, copy **C:\gmeeksdk\gmeek\classes\stcexception.jar** to **C:\gmeekjars\classes** and to <**eGate>\client\classes**.

If using e*Gate version 4.5.2 or later, copy **eGate\Server\registry\repository\default\classes\stcexception.jar** to **C:\gmeekjars\classes** and to <**eGate>\client\classes** (if not already in this directory).

9.2.3 One-Time Setup Steps for the RMI Server

The following steps only need to be done once to set up an RMI server.You will set up the RMI server only for the purposes of the sample. The RMI server is used to simulate an external system. For an overview and more information on the RMI server, see **Appendix C**.

1. Change the current directory to **C:\gmeeksdk\gmeek\builder\rmiDemoSvr\**.
2. If your environment differs from the assumptions noted in the pre-installation steps on page 122, edit the **compile** script and make appropriate substitutions.
3. Run the .**compile** script.

Results

Running the compile script creates the following directories and files. If there are pre-existing copies, they are overwritten.

- File

  ```
  .\RmiDemoSvr.jar
  ```
- Directory

  ```
  .\com\stc\eways\samples\gmeek\builder\rmiDemoSvr\
  ```
- The following **.class** files within the ...\com\stc\...\rmiDemoSvr\ directory:
  - RmiDemoSvrIntf.class
  - RmiDemoClient.class
  - RmiDemoSvrImpl.class
  - RmiDemoSvrImpl_Skel.class
  - RmiDemoSvrImpl_Stub.class
  - RmiDemoSvr.class

These files allow you to run the RMI Server to simulate an external accounting system.

9.3 Task 2: Creating the Back-end for the ETD Builder

1. Change directories to **C:\gmeeksdk\gmeek\builder\apiDemo\**.
2. If your environment differs from the assumptions noted in the pre-installation steps on page 122, edit the **compile** script and make appropriate substitutions.
3. Browse the sample code in **GmeekDemoBuilder.java** to understand the usage pattern. For a listing and explanation, see **"GmeekDemoBuilder.java" on page 112**.

This Java source code uses default values and settings as shown in Table 11.

Table 11 As-Shipped Settings in RmiAccounTester.java

Item	Original Setting or Value
e*Way name	GmeekDemoEway
Package name	com.stc.eways.GmeekDemoEway
e*Way Connection prefix	DemoRmiConnector
RMI property named "Host"	"localhost"
RMI property named "Port"	"11990"
RMI property named "Name"	"RmiDemoSvr"

You can make changes to validate or test the sample, but changing any of the preceding will necessitate corresponding changes in other files.

4 Run the .**compile** script.

Results

Running the compile script creates the following directories and files. If there were pre-existing copies, they are overwritten.

- Directory

  `.\com\stc\eways\samples\gmeek\builder\apiDemo`
- The following **.class** file within the ...\com\stc\...\apiDemo\ directory:
  - **GmeekDemoBuilder.class**

  This class creates an "external accounting system" simulated by the RMI Server.
- The following **.jar** file is rebuilt, making it generic:
  - ..**GmeekDemoEwayETDbuilder.jar**
- The rebuilt **.jar** file overwrites the as-shipped copy located in
  - ..\..\installEWAY\GmeekDemoEway\GmeekDemoEwayETDbuilder.jar

  This **.jar** file allows you to test the back-end of ETD builder against the system simulated by the RMI server.

9.4 Task 3: Building the e*Way and e*Way Connection

In this section, you will accomplish the following sub-tasks:

1. Run the **runapidemo** script to set up the e*Gate files associated with the e*Way (including the corresponding e*Way Connection).
2. Make changes to source files and then run the **compile** script to build the e*Way.
3. Make changes to **.ctl** files and then run the **installEWAY** script to install the files to the e*Gate system.

4 Verify that the e*Way and e*Way Connection were successfully installed.

Completion steps and results are provided for each sub-tasks below.

Note: *In general, you install e*Ways and e*Way Connections to the **default** schema, to make them available to all schemas. Accordingly, in this example, you will install the e*Way Connection Type named **GmeekDemoEway** to the schema named **default**.*

To set up the e*Gate files associated with the e*Way and e*Way connection

If you have made changes to any source files of the GmeekDemoEway sample, you must do the following steps. If you have not made any changes to any source files, skip ahead to step 2 of "To register the e*Way and e*Way Connection with e*Gate" on page 127.

1 If appropriate, back up or move previously built versions (including the as-loaded versions) of files in **C:\gmeeksdk\gmeek\installEWAY\GmeekDemoEway\**.

2 Change the current directory to **C:\gmeeksdk\gmeek\builder\apiDemo\**.

3 If your environment differs from the assumptions noted in the pre-installation steps on page 122, edit the **runapidemo** script and make appropriate substitutions.

4 Run the .**runapidemo** script.

Results

As a result of running the **runapidemo** script, the following e*Gate files are created in **gmeek\installEWAY\GmeekDemoEway**:

- **compile.bat** and **compile.sh**—Scripts that will be used to build the e*Way and e*Way Connection.
- **connectionpoint.ini**—Fragment to be manually incorporated in the e*Way Connections master file.
- **EwayConnectorExtImpl.java**—Base class for the e*Way Connection.
- **GmeekDemoEway.def**—Default configuration parameters for the e*Way Connection.
- **gmeekdemoewaywizard.ctl**—The wizard uses this control file to load the class path for the JVM.
- **stcewgmeekdemoeway.ctl**—Control file to tell e*Gate how to register the e*Way.
- **DemoRmiConnector.java**—Class for the e*Way Connection (skeleton code only).

Note: *The files listed above are generated the first time you run the **apidemo** script. To regenerate these files you must specify the file name with a value in the appropriate batch files, as follows:*

```
connectionpoint.ini=Yes
```

To build the e*Way and e*Way connection

1. Change the current directory to **..\..\installEWAY\GmeekDemoEway\** (**..\..\installEWAY\GmeekDemoEway_4.5.1** for e*Gate version 4.5.1).

   The kit provides a compile script (**compile.bat** on Windows; **compile.sh** on UNIX) to set CLASSPATH information and to create a **.jar** file for the compiled .java files upon completion.

```
set GMEEK_EXTRACTDIR=C:\gmeekjars
set JAVA_PATH=C:\jdk1.3.1_02\bin
set
MYCLASSPATH="%GMEEK_EXTRACTDIR%\classes\stcjcs.jar;%GMEEK_EXTRACTDIR%\c
lasses\stcexception.jar;%GMEEK_EXTRACTDIR%\classes\stcutil.jar;YourThir
dParty.jar;%CLASSPATH%"

%JAVA_PATH%\javac -classpath %MYCLASSPATH% -d . *.java
@REM jar up the classes
%JAVA_PATH%\jar cvf GmeekDemoEwayrt.jar GmeekDemoEway\*.class
@REM jar up the source file to allow for debugger to use
@REM copy *.java GmeekDemoEway\.
@REM %JAVA_PATH%\jar uvf GmeekDemoEwayrt.jar GmeekDemoEway\*.java
@REM del GmeekDemoEway\*.java
del GmeekDemoEway\*.class
```

As needed, make the following changes to reflect your environment:

1. Specify the correct location where the files were extracted, if you did not follow the pre-installation steps. For example:

   `set GMEEK_EXTRACTDIR=C:\gmeekjars`

2. Specify the correct path location for your JDK. For example:

   `JAVA_PATH=C:\jdk1.6.0\bin`

3. When creating e*Way Connections from scratch, modify the directory locations in the **compile** script as needed.

4. Run the **.\compile** script.

Results

Running the compile script creates the following directories and files. Any pre-existing copies will have been overwritten.

- File **.\GmeekDemoEwayrt.jar**
- Directory **.\com\stc\eways\GmeekDemoEway**

To register the e*Way and e*Way Connection with e*Gate

1. Edit and modify each of the **.ctl** files—**stcewgmeekdemoeway.ctl** and **gmeekdemoewaywizard.ctl** —by doing one or more of the following:

   - If your e*Way requires one or more third-party files, add them under the Third Party **.jar** files with the appropriate concatenation of **.jar** files and their containing directories.
   - Use *forward* slashes (/), even on Windows systems. For example:

     `their.jar,ThirdParty,FILETYPE_BINTEXT`

```
another.jar,ThirdParty/xml,FILETYPE_BINTEXT
jdom.jar,RelativePath/jdom/jdom-b7,FILETYPE_BINTEXT
andyetanother.jar,/AbsolutePath/To/Directory,FILETYPE_BINTEXT
```

- If needed for the RMI simulation in this sample, be sure to substitute the name of the file that you created in step 3 on page 124—that is, **RmiDemoSvr.jar**—for the placeholder **YourThirdParty.jar**. For example:

```
RmiDemoSvr.jar,ThirdParty,FILETYPE_BINTEXT
```

2 Change to **C:\gmeeksdk\gmeek\installEWAY\**

3 Run the **installEWAY** script with the appropriate parameters. For this sample, to install the e*Way named **GmeekDemoEway** to the schema named **default**, do the following:

- On Windows, open a Command Prompt, change directories to **gmeek\installEWAY**, and enter the following command:

```
.\installEWAY -e GmeekDemoEway -s default -h localhost -g C:\eGate
```

- On UNIX, open a shell, change directories to **gmeek/installEWAY**, and enter the following command:

```
./installEWAY.sh -e GmeekDemoEway -s default -h localhost -g <eGate>
```

- Table 12 shows the usage and parameters for Windows and UNIX.

Table 12 Usage and Parameters for the **installEWAY** script

Operating System	Usage	Parameters
Windows	installEWAY [params] [param args	-g: eGate root directory (default = \eGate) -e: eWay name (required) -s: eGate schema (required) -h: eGate registry host (default = localhost) -r: eGate registry port (default = 23001) -u: eGate registry user (default=Administrator) -p: eGate registry user password (default=STC) -?: help screen.
UNIX	installEWAY.sh [params] [param args]	-g: eGate root directory -e: eWay name -s: eGate schema -h: eGate registry host -r: eGate registry port -u: eGate registry username -p: eGate registry password -v: verbose

Note: *If your Participating Host and Registry Host are running on different computers, you must have `stcinstd` running against the schema and Registry Host in use while running the `stcregutil` command.*

1. Type the following text at the command line:

```
stcinstd -rh <host> -rs <schema> -un <username> -up <password> -ss
```

2. Press **ENTER**.

Note: *In the **stcinstd** command line as shown, the flag **-ss** is optional and means to run the host as a service.*

For a complete explanation on using the e*Gate command line, including the **stcinstd** command, see the *e*Gate Integrator System Administration and Operations Guide.*

Results

The results of running the **installEWAY** command are as follows:

- The file **connectionpoint.ini** is retrieved from the Registry Host, backed up, modified, and appended with an entry for the new e*Way (if not already present), and the Registry version is updated.
- The files specified in **stcew**<*eWayStem*>**.ctl** are registered with the Registry Host. For example, **stcewgmeekdemoeway.ctl** registers the following.
  - Third-party **.jar** files and directories. For the sample, this is simulated by:
    - **ThirdParty\RmiDemoSvr.jar**
  - The **.def** (default configuration-parameter) file for the GmeekDemoEway e*Way:
    - **configs\GmeekDemoEway\GmeekDemoEway.def**
  - The run-time **.jar** file for the GmeekDemoEway e*Way Connection:
    - **classes\GmeekDemoEwayrt.jar**
  - The **.ctl** file associated with the ETD builder wizard for GmeekDemoEway:
    - **etd\gmeekdemoewaywizard.ctl**
  - The **.jar** file for the back-end of the ETD builder for GmeekDemoEway:
    - **classes\GmeekDemoEwayETDbuilder.jar**
  - The **.jar**, **.dll**, and **.bmp** files for the ETD builder wizard:
    - **classes\GmeekWizard.jar**: the Swing GUI jar file. It is not needed unless using lightweight Visual Basic.
    - **classes\GmeekWizard.dll**: the heavyweight Visual Basic Active X DLL.
    - **classes\GmeekWizard.bmp**: the bitmap icon.
  - **stcewgmeekdemoeway.ctl**
- The results are written to a log file, **C:\GmeekDemoEway_install_log.txt**.

To validate the new e*Way Connection within e*Gate

1 Start the e*Gate Schema Designer, log in to a Registry Host, and open or create a schema. If necessary, activate the **Components** tab at the bottom of the Navigator (left) pane.

2 In the Navigator pane, click the **e*Way Connections** folder.

3 Click the icon in the tool palette to create a new e*Way Connection.

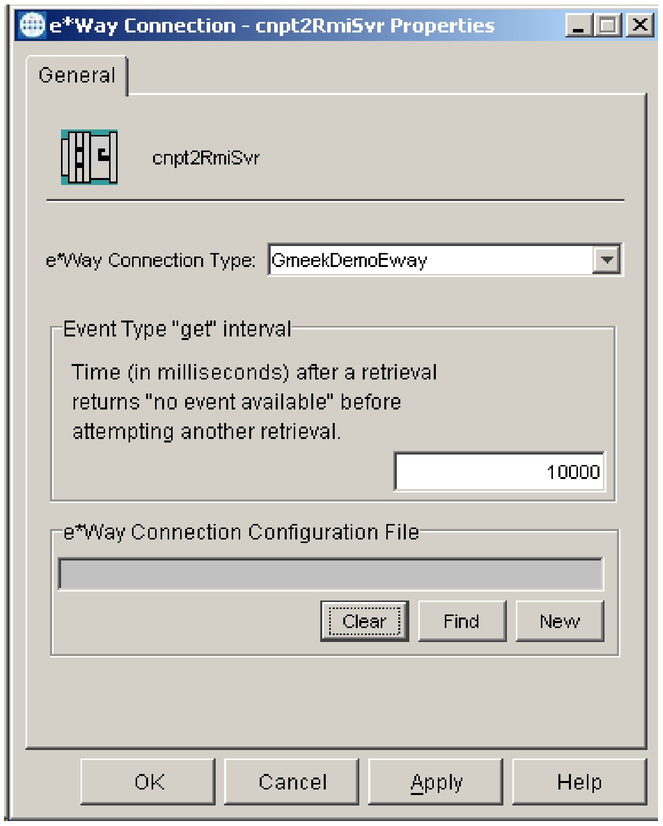
4 In the **New e*Way Connection** dialog box, enter a name (such as eWC_Gmeek), and then click **OK**.

5 In the component pane, right-click the new e*Way Connection and, on the context menu, click **Properties**.

The **e*Way Connection Properties** dialog box appears.

6 Select **GmeekDemoEway** from the **e*Way Connection Type** menu, as shown in Figure 29.

Figure 29 e*Way Connection Properties Dialog Box



cnpt2RmiSvr

e*Way Connection Type: GmeekDemoEway

Event Type "get" interval

Time (in milliseconds) after a retrieval returns "no event available" before attempting another retrieval.

10000

e*Way Connection Configuration File

This verifies that the e*Way Connection for **GmeekDemoEway** was successfully registered to the **connectionpoint.ini** file.

7 In the **e*Way Connection Type** list, select **GmeekDemoEway** and click **New**.

The Configuration Editor appears. Make sure that the parameter sections, parameter names, and default values match the way they were set up in the **GmeekDemoEway.def** file.

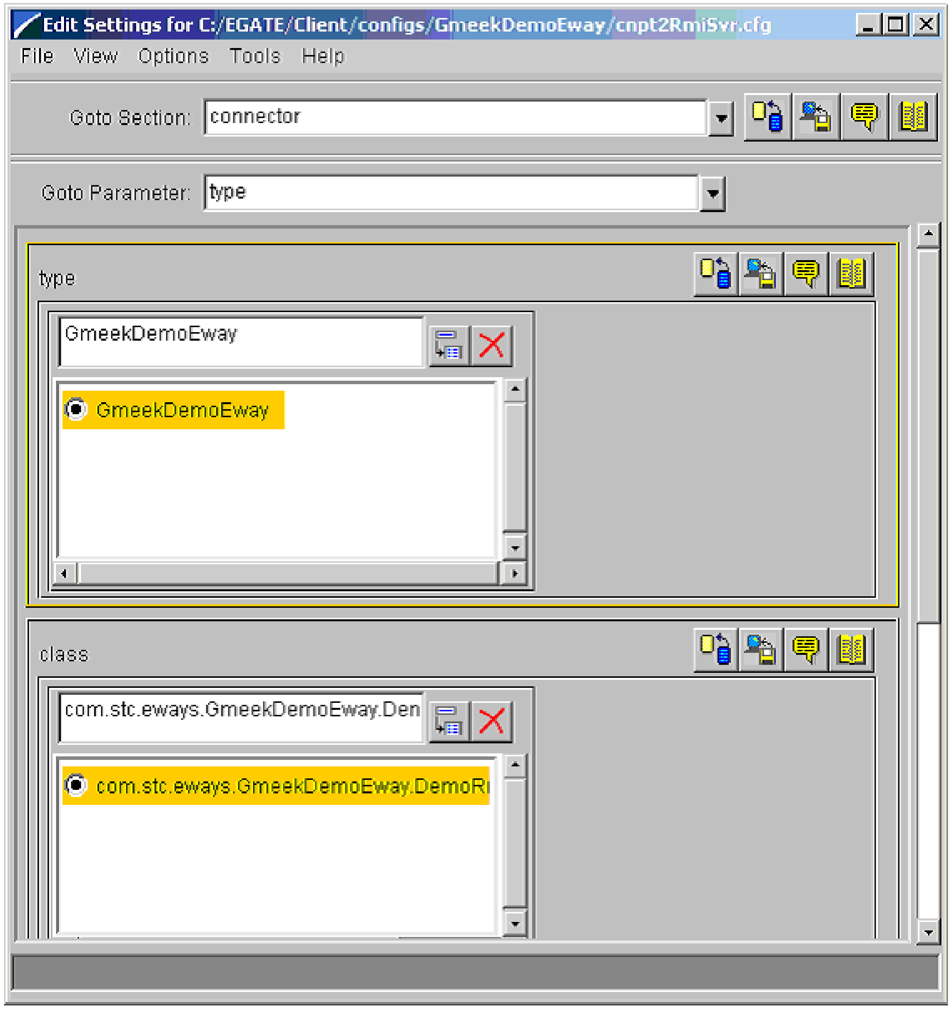
8 On the **File** menu, click **Save As**, and then enter **cnpt2RmiSvr** to save this all-default configuration in a new **.cfg** file named **cnpt2RmiSvr.cfg**.

9 Make changes to parameters as necessary, such as changing the connection management from Automatic Connection to Connection Management. When you are done, on the **File** menu, click **Save**.

Results

You created a new e*Way Connection named **cnpt2RmiSvr**, of type **GmeekDemoEway**, configured to use the parameters stored in **GmeekDemoEway.def**. See Figure 30.

Figure 30 Results of Validating the e*Way Connection



Edit Settings for C:/EGATE/Client/configs/GmeekDemoEway/cnpt2RmiSvr.cfg

Goto Section: connector

Goto Parameter: type

type: GmeekDemoEway

class: com.stc.eways.GmeekDemoEway.DemoRi

From now on, any Multi-Mode e*Way that uses this e*Way Connection is called a **GmeekDemoEway** type of e*Way. Because you installed the files to the **default** schema, you are able to create and deploy a **GmeekDemoEway** type of e*Way in any schema.

9.5 Task 4: Creating and Deploying an ETD Builder Wizard

Traditionally, ETD builder wizards are developed in Visual Basic. This approach ("heavyweight Visual Basic") is one of two presented in this sample. See **"Using Heavyweight Visual Basic to Create an ETD Builder Wizard"**.

Developers who are less familiar with Visual Basic can code most of the wizard using native Java GUI tools. This approach ("lightweight Visual Basic") is also presented in this sample. See **"Using Lightweight Visual Basic to Create an ETD Builder Wizard"**.

You must decide whether to use heavyweight Visual Basic or lightweight Visual Basic during the initial planning phase of creating your e*Way. We recommend that you use either only heavyweight Visual Basic or only lightweight Visual Basic for the lifetime of the e*Way.

Regardless of whether you use heavyweight Visual Basic or lightweight Visual Basic to create the ETD builder wizard you must create the files that deploy the wizard to e*Gate and validate the results. For more information about creating these files, see **"To compile the wizard into an Active X DLL"**.

9.5.1 Using Heavyweight Visual Basic to Create an ETD Builder Wizard

In this section, you will modify a generic Visual Basic wizard template, tailoring it to an ETD builder wizard that corresponds to a simulated external accounting system. You will then register the ETD builder wizard with e*Gate and validate it within e*Gate using the ETD Editor of the Schema Designer GUI.

Although in the other e*Way samples you may install a specific ETD to a specific schema, in this case you should install ETD builder wizards to the **default** schema in order to make them available to all end users. An end user can only use a builder wizard to create an ETD within a particular schema.

To create an ETD builder wizard

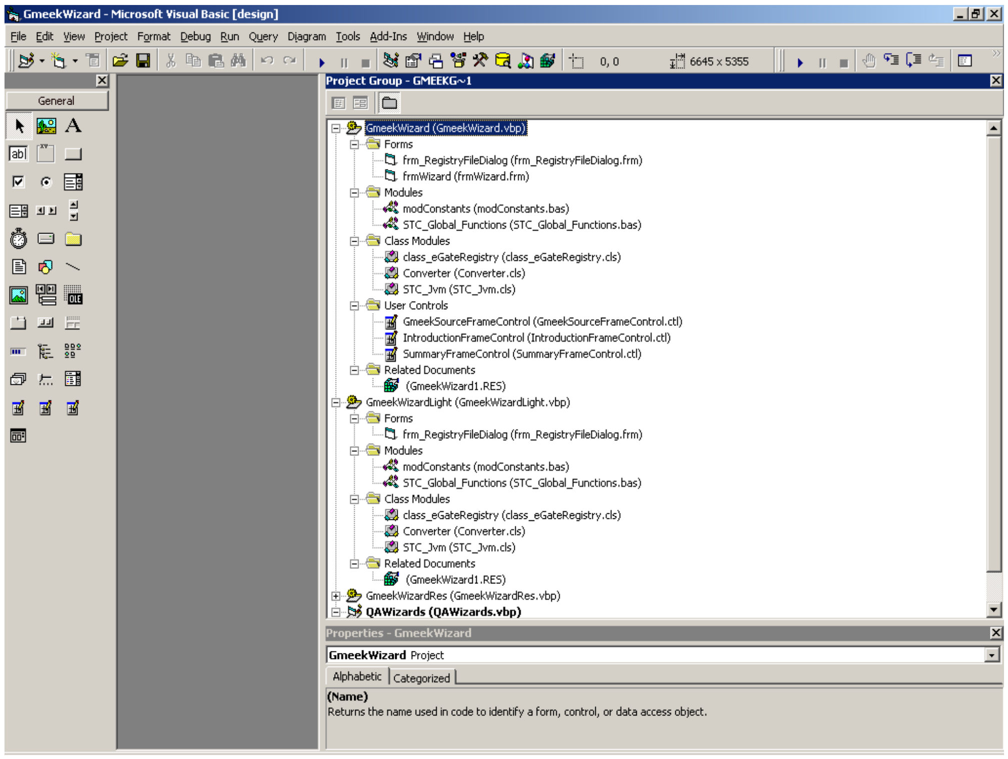
1 Extract and copy the files on the e*Gate Integrator Installation CD-ROM in **...\samples\sdk\gmeek\** to a working directory

2 Open a new Visual Basic session by double-clicking the following file:

```
C:\gmeeksdk\Wizards\GmeekGroup.Visual Basicg
```

3 The Microsoft Visual Basic wizard appears, as shown in Figure 31.

Figure 31 Gmeek Wizard in Microsoft Visual Basic

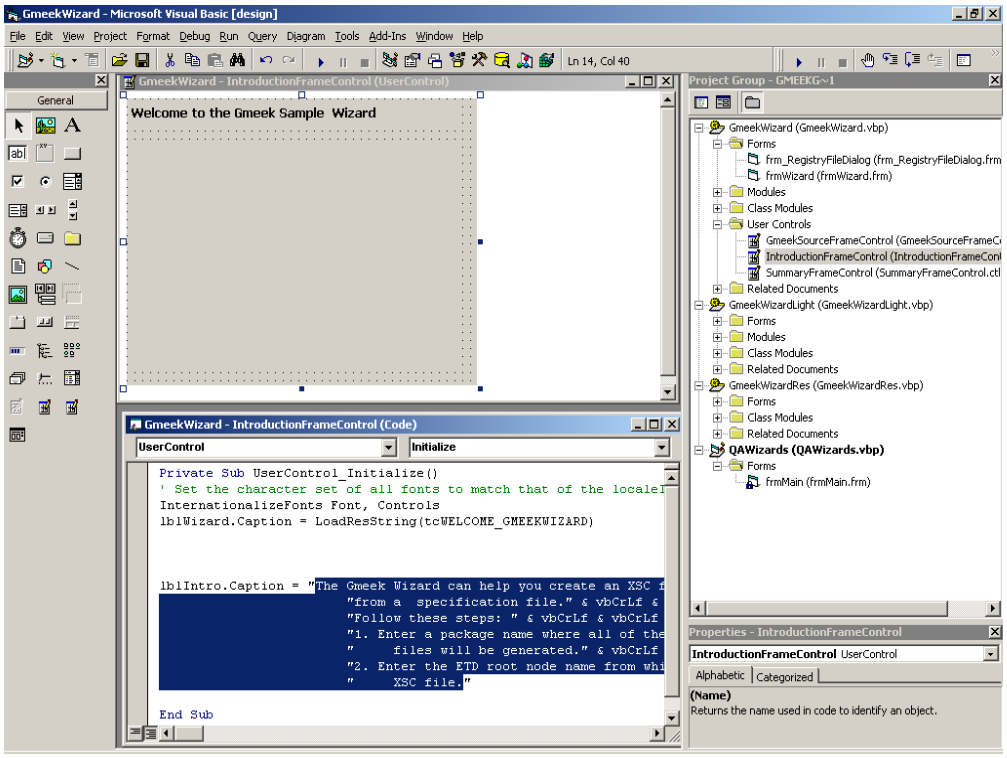

GmeekWizard - Microsoft Visual Basic [design]
File Edit View Project Format Debug Run Query Diagram Tools Add-Ins Window Help
0, 0
6645 x 5355
General
Project Group - GMEEKG~1

GmeekWizard (GmeekWizard.vbp)
Forms
frm_RegistryFileDialog (frm_RegistryFileDialog.frm)
frmWizard (frmWizard.frm)
Modules
modConstants (modConstants.bas)
STC_Global_Functions (STC_Global_Functions.bas)
Class Modules
class_eGateRegistry (class_eGateRegistry.cls)
Converter (Converter.cls)
STC_Jvm (STC_Jvm.cls)
User Controls
GmeekSourceFrameControl (GmeekSourceFrameControl.ctl)
IntroductionFrameControl (IntroductionFrameControl.ctl)
SummaryFrameControl (SummaryFrameControl.ctl)
Related Documents
(GmeekWizard1.RES)
GmeekWizardLight (GmeekWizardLight.vbp)
Forms
frm_RegistryFileDialog (frm_RegistryFileDialog.frm)
Modules
modConstants (modConstants.bas)
STC_Global_Functions (STC_Global_Functions.bas)
Class Modules
class_eGateRegistry (class_eGateRegistry.cls)
Converter (Converter.cls)
STC_Jvm (STC_Jvm.cls)
Related Documents
(GmeekWizard1.RES)
GmeekWizardRes (GmeekWizardRes.vbp)
QAWizards (QAWizards.vbp)

Properties - GmeekWizard
GmeekWizard Project
Alphabetic | Categorized
(Name)
Returns the name used in code to identify a form, control, or data access object.

4 In the **User Controls** folder, click **IntroductionFrameControl**. The screen shown in Figure 32 appears.

Figure 32 Gmeek Wizard in Microsoft Visual Basic: IntroductionFrameControl


GmeekWizard - Microsoft Visual Basic [design]
File Edit View Project Format Debug Run Query Diagram Tools Add-Ins Window Help
Ln 14, Col 40
General
GmeekWizard - IntroductionFrameControl (UserControl)
Welcome to the Gmeek Sample Wizard
Project Group - GMEEKG~1
GmeekWizard (GmeekWizard.vbp)
Forms
frm_RegistryFileDialog (frm_RegistryFileDialog.frm
frmWizard (frmWizard.frm)
Modules
Class Modules
User Controls
GmeekSourceFrameControl (GmeekSourceFrameC
IntroductionFrameControl (IntroductionFrameCon
SummaryFrameControl (SummaryFrameControl.ctl
Related Documents
GmeekWizardLight (GmeekWizardLight.vbp)
Forms
Modules
Class Modules
Related Documents
GmeekWizardRes (GmeekWizardRes.vbp)
Forms
Class Modules
Related Documents
QAWizards (QAWizards.vbp)
Forms
frmMain (frmMain.frm)
GmeekWizard - IntroductionFrameControl (Code)
UserControl
Initialize
Properties - IntroductionFrameControl
IntroductionFrameControl UserControl
Alphabetic
Categorized
(Name)
Returns the name used in code to identify an object.

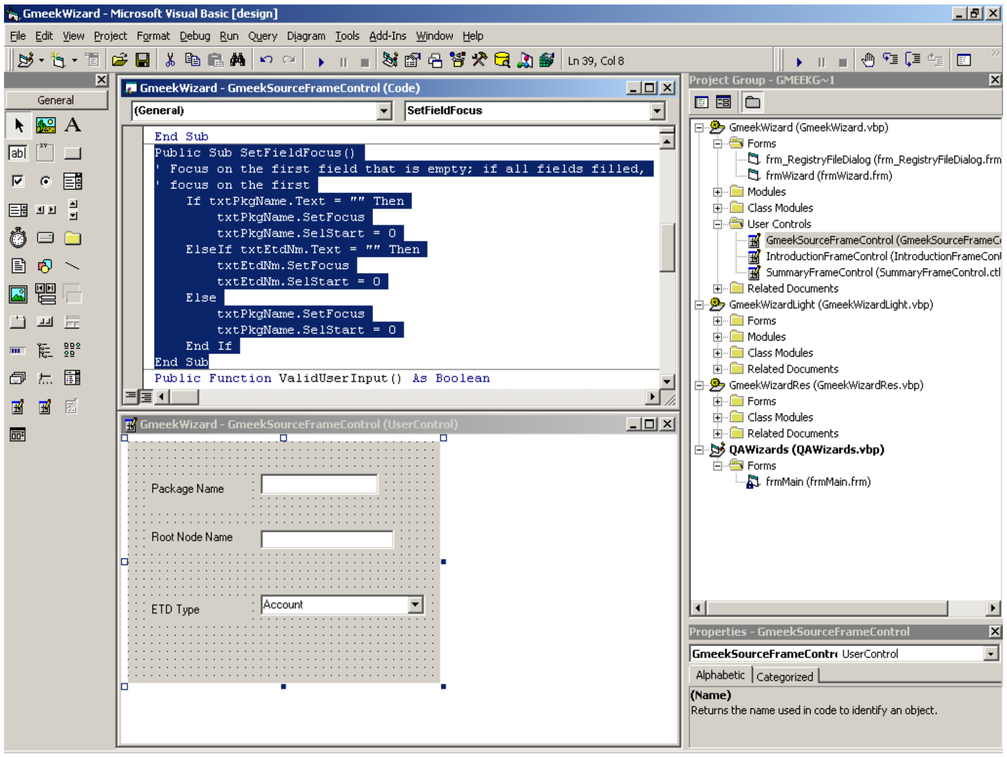
```
Private Sub UserControl_Initialize()
' Set the character set of all fonts to match that of the localeI
InternationalizeFonts Font, Controls
lblWizard.Caption = LoadResString(tcWELCOME_GMEEKWIZARD)



lblIntro.Caption = "The Gmeek Wizard can help you create an XSC f
                   "from a  specification file." & vbCrLf &
                   "Follow these steps: " & vbCrLf & vbCrLf
                   "1. Enter a package name where all of the
                   "     files will be generated." & vbCrLf
                   "2. Enter the ETD root node name from whi
                   "     XSC file."

End Sub
```

5 If you want to edit the introductory text of the wizard, highlight the existing text and type in the new text. See the highlighted area in Figure 32.

Note: *All changes made to text are automatically saved in Microsoft Visual Basic.*

6 In the **User Controls** folder, click **GmeekSourceFrameControl**. The screen shown in Figure 33 appears.

Figure 33 Gmeek Wizard in Microsoft Visual Basic: Gmeek SourceFrameControl


GmeekWizard - Microsoft Visual Basic [design]
File Edit View Project Format Debug Run Query Diagram Tools Add-Ins Window Help
Ln 39, Col 8
General
GmeekWizard - GmeekSourceFrameControl (Code)
(General)
SetFieldFocus
GmeekWizard - GmeekSourceFrameControl (UserControl)
Project Group - GMEEKG~1
GmeekWizard (GmeekWizard.vbp)
Forms
frm_RegistryFileDialog (frm_RegistryFileDialog.frm
frmWizard (frmWizard.frm)
Modules
Class Modules
User Controls
GmeekSourceFrameControl (GmeekSourceFrameC
IntroductionFrameControl (IntroductionFrameCon
SummaryFrameControl (SummaryFrameControl.ctl
Related Documents
GmeekWizardLight (GmeekWizardLight.vbp)
Forms
Modules
Class Modules
Related Documents
GmeekWizardRes (GmeekWizardRes.vbp)
Forms
Class Modules
Related Documents
QAWizards (QAWizards.vbp)
Forms
frmMain (frmMain.frm)
Properties - GmeekSourceFrameControl
GmeekSourceFrameContr UserControl
Alphabetic
Categorized

```
End Sub
Public Sub SetFieldFocus()
' Focus on the first field that is empty; if all fields filled,
' focus on the first
    If txtPkgName.Text = "" Then
        txtPkgName.SetFocus
        txtPkgName.SelStart = 0
    ElseIf txtEtdNm.Text = "" Then
        txtEtdNm.SetFocus
        txtEtdNm.SelStart = 0
    Else
        txtPkgName.SetFocus
        txtPkgName.SelStart = 0
    End If
End Sub
Public Function ValidUserInput() As Boolean
```

Package Name

Root Node Name

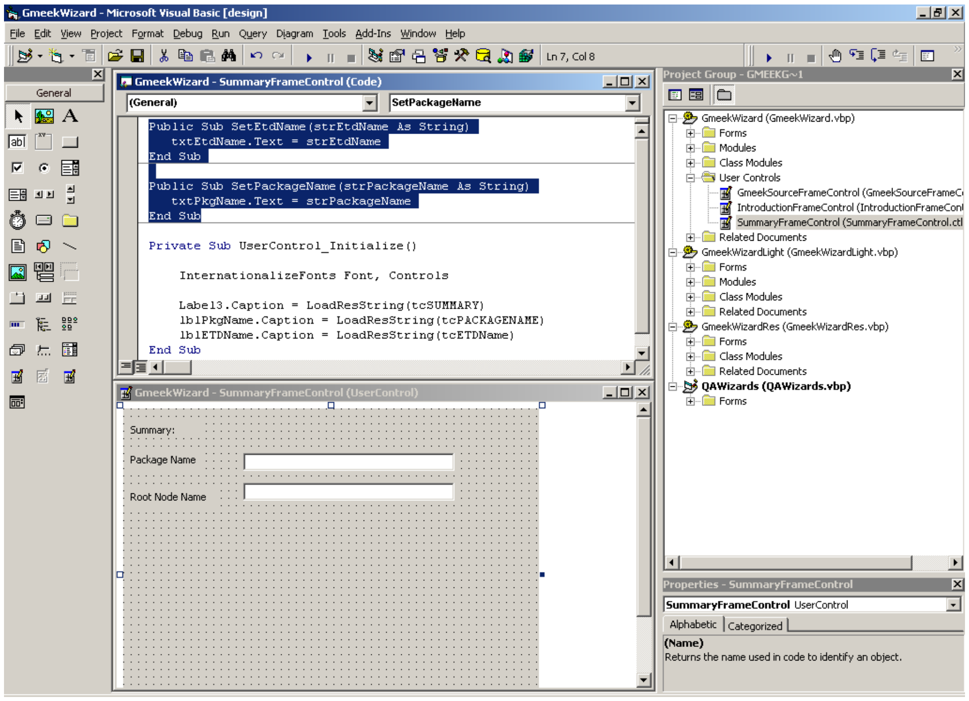
ETD Type: Account

(Name)
Returns the name used in code to identify an object.

The behavior of the three text boxes is defined in the function SetFieldFocus. If you require more boxes to capture additional user input, use the SetFieldFocus function to add them.

7 In the **User Controls** folder, click **SummaryFrameControl**. The screen shown in Figure 34 appears.

Figure 34 Gmeek Wizard in Microsoft Visual Basic: SummaryFrameControl


GmeekWizard - Microsoft Visual Basic [design]
File Edit View Project Format Debug Run Query Diagram Tools Add-Ins Window Help
Ln 7, Col 8
General
GmeekWizard - SummaryFrameControl (Code)
(General)
SetPackageName
GmeekWizard - SummaryFrameControl (UserControl)
Project Group - GMEEKG~1
GmeekWizard (GmeekWizard.vbp)
Forms
Modules
Class Modules
User Controls
GmeekSourceFrameControl (GmeekSourceFrameC
IntroductionFrameControl (IntroductionFrameCon
SummaryFrameControl (SummaryFrameControl.ctl
Related Documents
GmeekWizardLight (GmeekWizardLight.vbp)
Forms
Modules
Class Modules
Related Documents
GmeekWizardRes (GmeekWizardRes.vbp)
Forms
Class Modules
Related Documents
QAWizards (QAWizards.vbp)
Forms
Properties - SummaryFrameControl
SummaryFrameControl UserControl
Alphabetic
Categorized

```
Public Sub SetEtdName(strEtdName As String)
   txtEtdName.Text = strEtdName
End Sub

Public Sub SetPackageName(strPackageName As String)
   txtPkgName.Text = strPackageName
End Sub

Private Sub UserControl_Initialize()

    InternationalizeFonts Font, Controls

    Label3.Caption = LoadResString(tcSUMMARY)
    lblPkgName.Caption = LoadResString(tcPACKAGENAME)
    lblETDName.Caption = LoadResString(tcETDName)
End Sub
```

Summary:
Package Name
Root Node Name

(Name)
Returns the name used in code to identify an object.

In Figure 34, the highlighted area shows a summary of the captured user input. The Visual Basic Sub displays the values captured from **Gmeek SourceFrameControl**.

Generic wizard template forms in Microsoft Visual Basic

In the **Forms** folder, the **frmWizard** contains the controls tailored to your wizard. The following three controls are embedded in this frame:

- GmeekSourceFrameControl
- IntroductionFrameControl
- SummaryFrameControl

To create a new wizard using the template, modify the Public Sub SetStep(nStep As Integer, nDirection As Integer) in **frmWizard**. Public Sub SetStep(nStep As Integer, nDirection As Integer) defines how to navigate to different user controls. If you add a user control, you must put the correct navigation rule here.

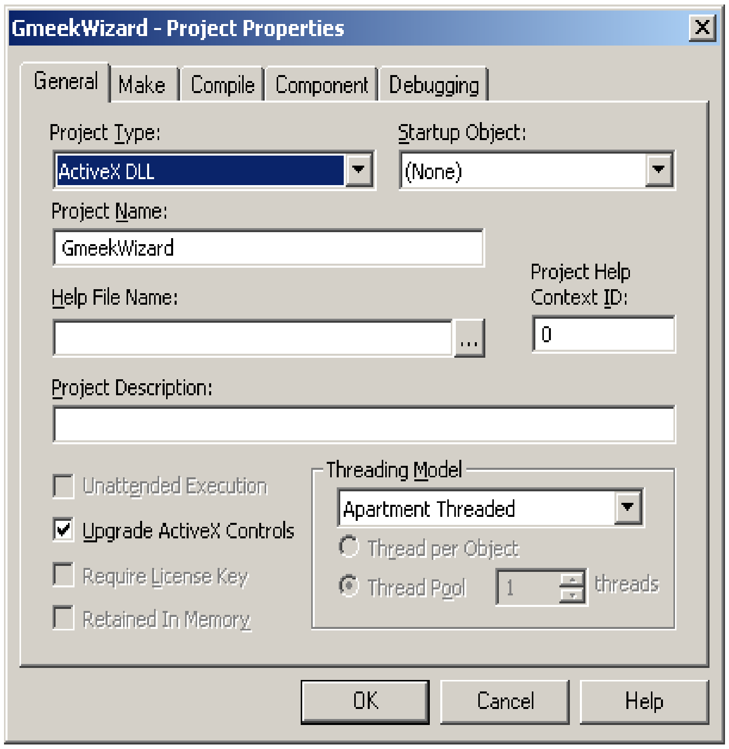
The navigation logic among the three controls is defined in the following Sub:

```
Public Sub SetStep(nStep As Integer, nDirection As Integer)
    Dim retVal As Long
    Select Case nStep
        Case STEP_INTRO
           IntroductionFrameControl1.Visible = True
           GmeekSourceFrameControl1.Visible = False
           SummaryFrameControl1.Visible = False

           frmWizard.Caption = LoadResString(tcGMEEKWIZARD_INTRO)
           cmdFinish.Enabled = False
           cmdFinish.Visible = False
           cmdNav(3).Visible = True

           If nDirection = DIR_BACK Then
              ' set focus to Next button
              cmdNav(3).SetFocus
           End If

        Case STEP_SELECT
           IntroductionFrameControl1.Visible = False
           SummaryFrameControl1.Visible = False
           GmeekSourceFrameControl1.Visible = True
           GmeekSourceFrameControl1.SetFocus

           frmWizard.Caption = LoadResString(tcGMEEK_WIZARD_STEP1)
           cmdFinish.Enabled = False
           cmdFinish.Visible = False
           cmdNav(3).Visible = True

           ' set focus on control field
           GmeekSourceFrameControl1.SetFieldFocus

       Case STEP_SUMMARY
           IntroductionFrameControl1.Visible = False
           GmeekSourceFrameControl1.Visible = False
           SummaryFrameControl1.Visible = True
           cmdFinish.Enabled = True
           cmdFinish.Visible = True
           cmdNav(3).Visible = False
           frmWizard.Caption = LoadResString(tcGMEEK_WIZARD_STEP2)
    End Select

    SetNavBtns nStep

End Sub

```

To compile the wizard into an Active X DLL

1. Right-click and select Project Properties.
2. Set the GmeekWizard project type as an Active X DLL, as shown in Figure 35.

Figure 35 Wizard Project Properties


GmeekWizard - Project Properties

General | Make | Compile | Component | Debugging

Project Type: ActiveX DLL
Startup Object: (None)
Project Name: GmeekWizard
Help File Name:
Project Help Context ID: 0
Project Description:

- [ ] Unattended Execution
- [x] Upgrade ActiveX Controls
- [ ] Require License Key
- [ ] Retained In Memory

Threading Model: Apartment Threaded
Thread per Object
[✓ Thread Pool] 1 threads

3 To compile the wizard, from the File menu click **File**, and select **Make GmeekWizard.dll**.

4 Copy the **GmeekWizard.dll** to the **GmeekDemoEway** working directory.

The **GmeekWizard.dll** is loaded into e*Gate when the **installEWAY** script is run.

Note: *Make sure that the* ***.dll*** *and* ***.bmp*** *files are copied to the e*Way working directory. In the sample, the working directory is* ***gmeek/installEWAY/GmeekDemoEway****.*

9.5.2 Using Lightweight Visual Basic to Create an ETD Builder Wizard

In this section, you will create an ETD builder wizard using lightweight Visual Basic, tailoring it to an ETD builder wizard that corresponds to a simulated external accounting system. You will then register the ETD builder wizard with e*Gate and validate it within e*Gate using the ETD Editor of the Schema Designer GUI.

Although in the other e*Way samples you may install a specific ETD to a specific schema, in this case you should install ETD builder wizards to the **default** schema in order to make them available to all end users. An end user can only use a builder wizard to create an ETD within a particular schema.

The sample includes a lightweight Visual Basic wizard project, called **GmeekWizardLight**. The advantages of using the lightweight Visual Basic approach are:

- it creates wizards with less Visual Basic code.
- since the wizard is written in Java, there is neither a frmWizard Visual Basic Form, nor any Visual Basic graphical user controls.

Creating the Java Wizard

The ETD Editor launches the lightweight Visual Basic wizard by invoking the Start() method in lightweight Visual Basic. The Start() method then calls the corresponding Start() method of the Java wizard.

The Java wizard is written using any of the Java foundation classes such as Swing or AWT. All the Java files are compiled and packaged into **GmeekWizard.jar**.

The ETD Editor expects the return values listed in Table 13. You must use the same set of return values when you code the Java wizard.

Table 13 Return Values

Return Value	Definition
0	Success
-1	Not implemented
-2	Cancelled
-8	General Failure

Important: *If the Java wizard does not return the correct value to the ETD Editor, an error may occur.*

To create a Java ETD builder wizard launched from lightweight Visual Basic

1. In the sample code, the Java wizard implements the following interface:

```
public interface JConverterIntf {
    public void SetProp(String PropName, String PropValue);
    public int Start() throws Exception;
}
```

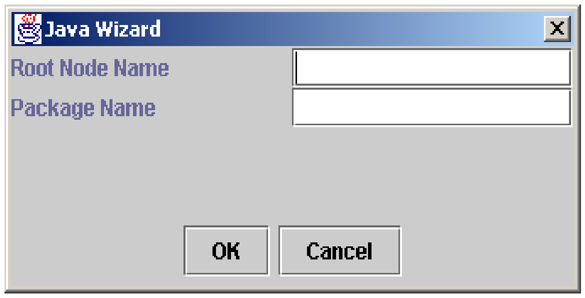
2. Invoking the Start() method starts the JDialog window.

Note: *The JDialog class is the main class for creating a dialog window. Use this class to create a custom dialog.*

3. Use the JDialog class for the Java wizard.
4. Call the back-end Compile() method. If all the input values are valid, the back-end builder will generate the **.xsc** and **.jar** files and assign them to the designated directory.

The sample Java wizard uses classes from Swing Java. This sample Java wizard dialog appears as shown in Figure 36.

Figure 36 Example of Customized Java Wizard Dialog



Using the lightweight Visual Basic Converter class, the execution is transferred from Visual Basic to the Java wizard when the Start() method is called.

The following sample code shows the Start() method in lightweight Visual Basic.

```
Public Function Start() As Long
...
If Not m_objJvm Is Nothing Then
Set objGmeek =  m_objJvm.GetJavaClass(GMEEKBACKENDETDWIZARDCLASS)
If Not objGmeek Is Nothing Then
Dim KeyArray As Variant
Dim i As Long
KeyArray = m_Props.Keys
For i = LBound(KeyArray) To UBound(KeyArray)
                    objGmeek.SetProp KeyArray(i),
m_Props.Item(KeyArray(i))
Next i
...
ret=objGmeek.Start
...
m_objJvm.Terminate
set m_objJvm=Nothing
Else
End If...
...
start=ret
...
```

The **objGmeek.Start** call transfers the execution to the Java wizard.

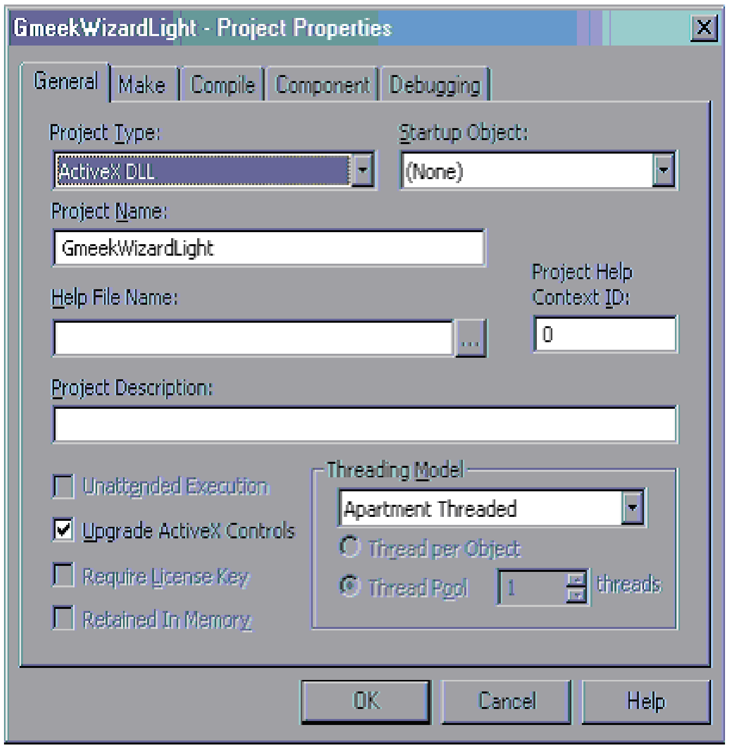
Note: *The sample Java wizard includes a Main() method to enable running a stand-alone test.*

To compile the wizard into an Active X DLL

1 Right-click and select Project Properties.

2 Set the GmeekWizardLight project type as an Active X DLL, as shown in Figure 37.

Figure 37 Wizard Project Properties


GmeekWizardLight - Project Properties

Project Type: ActiveX DLL
Startup Object: (None)
Project Name: GmeekWizardLight
Help File Name:
Project Help Context ID: 0
Project Description:

[ ] Unattended Execution
[✓] Upgrade ActiveX Controls
[ ] Require License Key
[ ] Retained In Memory

Threading Model: Apartment Threaded
( ) Thread per Object
(●) Thread Pool 1 threads

3 To compile the wizard, from the File menu click **File**, and select **Make GmeekWizardLight.dll**.

4 Copy the **GmeekWizardLight.dll** to the **GmeekDemoEway** working directory.

Note: *Make sure that the **.dll**, **.bmp**, and **.jar** files are copied to the e*Way working directory. In the sample, the working directory is **gmeek/installEWAY/GmeekDemoEway**.*

The **GmeekWizardLight.dll** is loaded into e*Gate when the **installEWAY** script is run. (as long as the **stcewgmeekdemoeway.ctl** file is modified accordingly). For more information, see **"Modifying stcewgmeekdemoeway.ctl" on page 141**.

9.5.3 Deploying the ETD wizard in e*Gate

Whether using heavyweight Visual Basic or lightweight Visual Basic, to deploy the ETD wizard in e*Gate you must modify both the **stcewgmeekdemoeway.ctl** file and the **ETDWizards.ini** file. In addition, you must include all **.dll** and **.bmp** files.

Modifying stcewgmeekdemoeway.ctl

In the sample, the generated **stcewgmeekdemoeway.ctl** without any modifications looks like the following:

```
##-----------------------------------------------------------------
#  stcewgmeekdemoeway.ctl (The CTL used to install your eway
connection components)
```

```
#
# This CTL file is used to install your EWAY Runtime and builder in
eGate.
##---------------------------------------------------------------------
stcexception.jar,classes,FILETYPE_BINTEXT
##---------------------------------------------------------------------
#   Any Third party JAR files used
#    replace the YourThirdParty.jar with real name of the jar file
#    You can specify the path in ThirdParty directory
#    e.g. jdom.jar,ThirdParty/jdom/jdom-b7,FILETYPE_BINTEXT
##---------------------------------------------------------------------
RmiDemoSvr.jar,ThirdParty,FILETYPE_BINTEXT
##---------------------------------------------------------------------
#   The DEF file (if used) associated with your eway connection
##---------------------------------------------------------------------
GmeekDemoEway.def,configs/GmeekDemoEway,FILETYPE_ASCIITEXT
##---------------------------------------------------------------------
#  The runtime jar file (if used) associated with your eway connector
##---------------------------------------------------------------------
GmeekDemoEwayrt.jar,classes,FILETYPE_BINTEXT
##---------------------------------------------------------------------
#   The wizard ctl file (if used) associated with your eway ETD
builder
##---------------------------------------------------------------------
gmeekdemoewaywizard.ctl,etd,FILETYPE_ASCIITEXT
##---------------------------------------------------------------------
#   The backend builder jar file (if used) associated with your eway
ETD builder
##---------------------------------------------------------------------
GmeekDemoEwayETDbuilder.jar,classes,FILETYPE_BINTEXT
##---------------------------------------------------------------------
#   The Visual Basic ETD wizard BMP file (if used) associated with
your eway ETD builder
##---------------------------------------------------------------------
GmeekWizard.bmp,bin/WizardIcons,FILETYPE_BINTEXT
##---------------------------------------------------------------------
#   The Visual Basic ETD wizard DLL file (if used) associated with
your eway ETD builder
##---------------------------------------------------------------------
GmeekWizard.dll,bin/win32,FILETYPE_BINTEXT
##---------------------------------------------------------------------
#   The Java ETD wizard jar file (if used) associated with your eway
ETD builder
#   This Java Swing based wizard will work with a light weight Visual
Basic wizard
##---------------------------------------------------------------------
GmeekWizard.jar,classes,FILETYPE_BINTEXT
```

Important: *Make sure the* ***.bmp*** *and* ***.dll*** *file s are copied to the e*Way working directory.*

To modify the stcewgmeekdemoeway.ctl file using heavyweight Visual Basic

1 Replace the following line:

```
#GmeekWizard.bmp,bin/WizardIcons,FILETYPE_BINTEXT
```

with

```
GmeekWizard.bmp,bin/WizardIcons,FILETYPE_BINTEXT
```

2 Replace the following line:

```
#GmeekWizard.dll,bin/win32,FILETYPE_BINTEXT
```

with

```
GmeekWizard.dll,bin/win32,FILETYPE_BINTEXT
```

To modify the stcewgmeekdemoeway.ctl file using lightweight Visual Basic

1 Replace the following line:

```
#GmeekWizard.bmp,bin/WizardIcons,FILETYPE_BINTEXT
```

with

```
GmeekWizardLight.bmp,bin/WizardIcons,FILETYPE_BINTEXT
```

2 Replace the following line:

```
#GmeekWizard.dll,bin/win32,FILETYPE_BINTEXT
```

with

```
GmeekWizardLight.dll,bin/win32,FILETYPE_BINTEXT
```

3 Replace the following line:

```
#GmeekWizard.jar,classes,FILETYPE_BINTEXT
```

with

```
GmeekWizard.jar,classes,FILETYPE_BINTEXT
```

Modifying ETDWizards.ini

In the sample, the generated **ETDWizards.ini** file looks like the following:

```
?GmeekDemoEway=GmeekDemoEway Wizard,
GmeekDemoEwayWizard.bmp,GmeekDemoEwayWizard.Converter,GmeekDemoEwayWizard.dll,
```

To modify the ETDWizards.ini file using heavyweight Visual Basic

1 In the generated **ETDWizards.ini** file, replace the following line making sure it is all in one line without a line break:

```
?GmeekDemoEway=GmeekDemoEway Wizard,
GmeekDemoEwayWizard.bmp,GmeekDemoEwayWizard.Converter,GmeekDemoEwayWizard.dll
```

with

```
?GmeekDemoEway=Demo Eway,  GmeekWizard.bmp, GmeekWizard.Converter,GmeekWizard.dll,
```

To modify the ETDWizards.ini file using lightweight Visual Basic

1 In the generated **ETDWizards.ini** file, replace the following line making sure it is all in one line without a line break:

```
?GmeekDemoEway=GmeekDemoEway Wizard,
GmeekDemoEwayWizard.bmp,GmeekDemoEwayWizard.Converter,GmeekDemoEwayWizard.dll
```

with

```
?GmeekDemoEway=Demo Eway Light,  GmeekWizardLight.bmp,
GmeekWizardLight.Converter,GmeekWizardLight.dll,
```

Deploying the ETD Wizard in e*Gate

1 If you have finished creating other e*Way components, then you must run the **installEWAY** script. The **installEWAY** script loads the wizard into e*Gate.

2

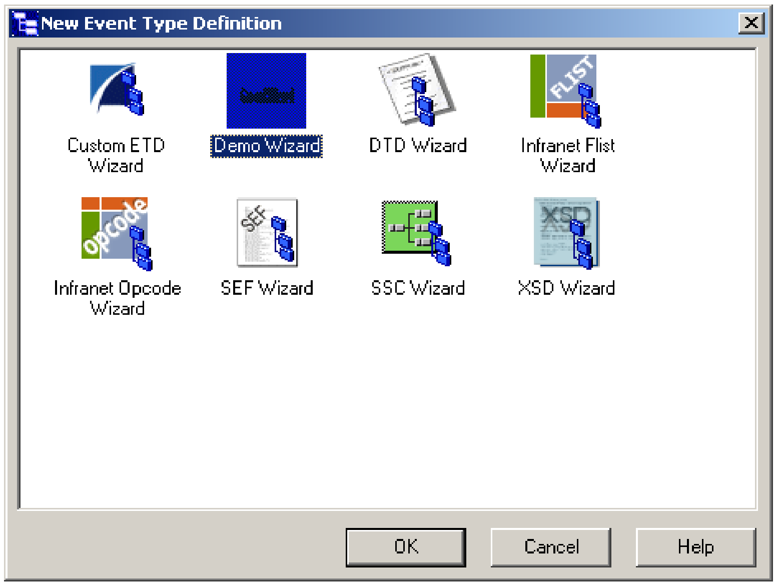
3 You must manually append the **ETDWizards.ini** in your working directory to the **ETDWizards.ini** in the Registry in:

<egate>\server\registry\repository\default\install

If using heavyweight Visual Basic

A The Demo Wizard icon appears in the Event Type Definition dialog, as shown in Figure 38.

Figure 38 Heavyweight Visual Basic Wizard Icon

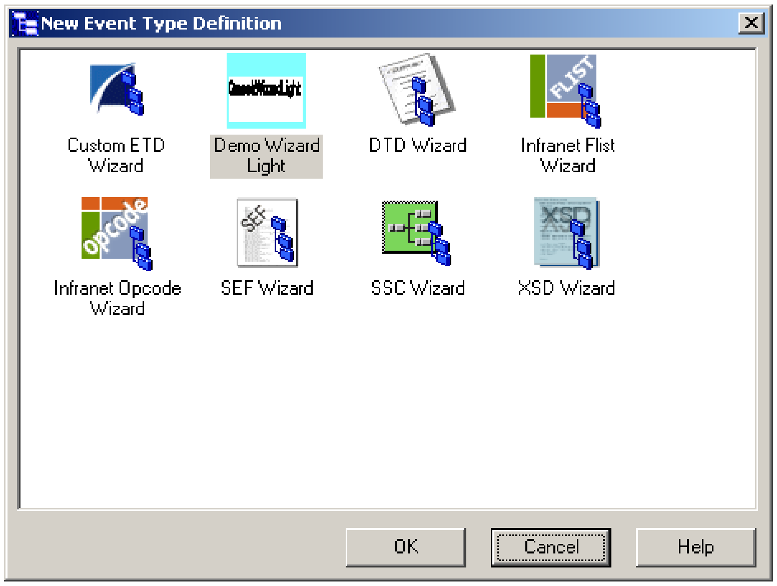


B Click the **Demo Wizard** icon to launch the wizard created in heavyweight Visual Basic.

If using lightweight Visual Basic

A The Demo Wizard Light icon appears in the Event Type Definition dialog, as shown in Figure 39.

Figure 39 Lightweight Visual Basic Wizard Icon



B Click the **Demo Wizard Light** icon to launch the wizard created in lightweight Visual Basic.

9.6 Task 5: Testing the Wizard with a Stand-alone Visual Basic Tester

The Wizard Visual Basic sample includes a stand-alone Visual Basic tester **QAWizards** project. You can use this project to test your ETD wizards outside of the ETD Editor. However, you still must commit the **.dll** and **.bmp** files to the e*Gate Registry.

The Visual Basic tester runs the wizards and displays the **.xsc** file. It reads its own **ETDWizards.ini** file, which is different in format from the **ETDWizards.ini** file in the e*Gate Registry.

The **ETDWizards.ini** file is included in the sample Visual Basic project. The ETDWizards have the following contents:

- GmeekWizard
- GmeekWizard.bmp
- GmeekWizard.Converter
- GmeekWizardLight
- GmeekWizardLight.bmp
- GmeekWizardLight.Converter

To validate the new ETD builder wizard using the stand-alone Visual Basic tester

1 In **C:\gmeeksdk\Wizards**, double click on **GmeekGroup.vbg**.

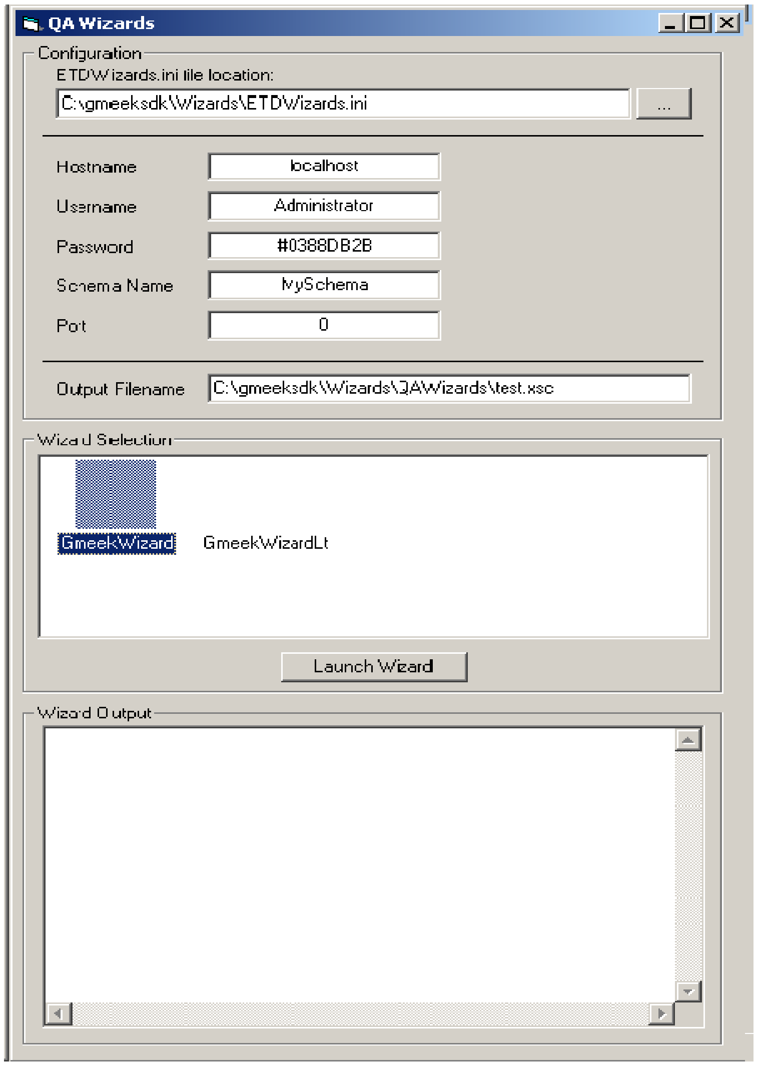
2 Set the QAWizards Start Up project, and select the **QAWizards.exe**.

3 From the Run menu, click **Start**.

4 From the ETDWizards.ini file location menu, select the **ETDWizards.ini** file that you extracted in the **gmeeksdk/wizards** directory.

Figure 40 shows the QAWizards stand-alone tester.

Figure 40 QA Wizards Stand-alone Tester



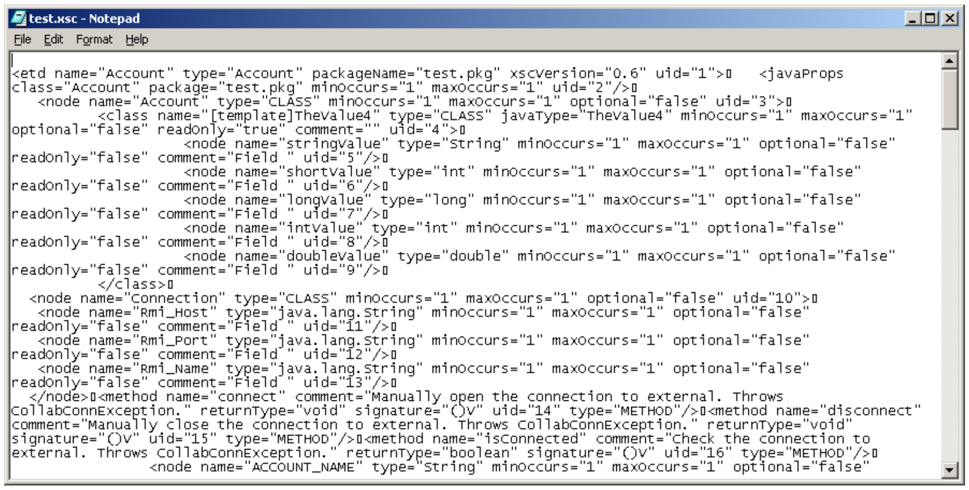
ETDWizards.ini file location: C:\gmeeksdk\Wizards\ETDWizards.ini

Hostname: localhost
Username: Administrator
Password: #0388DB2B
Schema Name: MySchema
Port: 0

Output Filename: C:\gmeeksdk\Wizards\QAWizards\test.xsc

5 Select a wizard and click **Launch Wizard**. Launching the wizard generates the **.xsc** file, shown in Figure 41.

Figure 41 Generated **.xsc** in Notepad



```
<etd name="Account" type="Account" packageName="test.pkg" xscVersion="0.6" uid="1">▯    <javaProps
class="Account" package="test.pkg" minOccurs="1" maxOccurs="1" uid="2"/>▯
   <node name="Account" type="CLASS" minOccurs="1" maxOccurs="1" optional="false" uid="3">▯
          <class name="[template]TheValue4" type="CLASS" javaType="TheValue4" minOccurs="1" maxOccurs="1"
optional="false" readOnly="true" comment="" uid="4">▯
                    <node name="stringValue" type="String" minOccurs="1" maxOccurs="1" optional="false"
readOnly="false" comment="Field " uid="5"/>▯
                    <node name="shortValue" type="int" minOccurs="1" maxOccurs="1" optional="false"
readOnly="false" comment="Field " uid="6"/>▯
                    <node name="longValue" type="long" minOccurs="1" maxOccurs="1" optional="false"
readOnly="false" comment="Field " uid="7"/>▯
                    <node name="intValue" type="int" minOccurs="1" maxOccurs="1" optional="false"
readOnly="false" comment="Field " uid="8"/>▯
                    <node name="doubleValue" type="double" minOccurs="1" maxOccurs="1" optional="false"
readOnly="false" comment="Field " uid="9"/>▯
          </class>▯
  <node name="Connection" type="CLASS" minOccurs="1" maxOccurs="1" optional="false" uid="10">▯
   <node name="Rmi_Host" type="java.lang.String" minOccurs="1" maxOccurs="1" optional="false"
readOnly="false" comment="Field " uid="11"/>▯
   <node name="Rmi_Port" type="java.lang.String" minOccurs="1" maxOccurs="1" optional="false"
readOnly="false" comment="Field " uid="12"/>▯
   <node name="Rmi_Name" type="java.lang.String" minOccurs="1" maxOccurs="1" optional="false"
readOnly="false" comment="Field " uid="13"/>▯
  </node>▯<method name="connect" comment="Manually open the connection to external. Throws
CollabConnException." returnType="void" signature="()V" uid="14" type="METHOD"/>▯<method name="disconnect"
comment="Manually close the connection to external. Throws CollabConnException." returnType="void"
signature="()V" uid="15" type="METHOD"/>▯<method name="isConnected" comment="Check the connection to
external. Throws CollabConnException." returnType="boolean" signature="()V" uid="16" type="METHOD"/>▯
                <node name="ACCOUNT_NAME" type="String" minOccurs="1" maxOccurs="1" optional="false"
```

The QAWizards merely displays the **.xsc** file without actually committing the ETD (**.xsc** file) to the e*Gate Registry. You can view the ETD source Java code in the **.jar** file in the **C:\wizards\QAWizards\** directory.

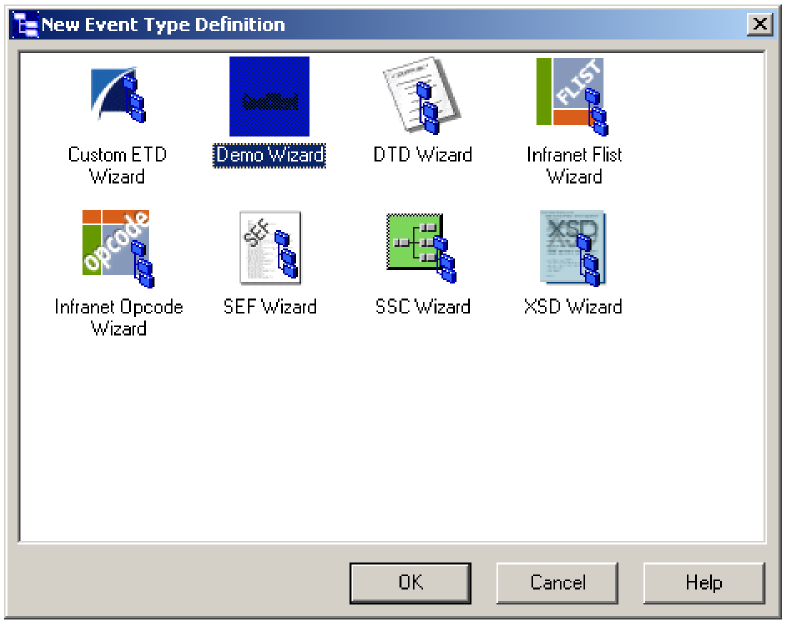
6 By using Notepad or a similar text editor, verify that the **.xsc** file was generated correctly.

To validate the new ETD builder within e*Gate

1 Start e*Gate Schema Designer, log in to the Registry Host, and open a schema. If necessary, activate the Components tab at the bottom of the Navigator (left) pane.

2 In the New Event Type Definition dialog, double-click the Demo Wizard icon, as shown in Figure 42.

Figure 42 New Event Type Definition Dialog



3 Follow the wizard instructions to:

A Enter a root node name.

B Enter a package name where the ETD Editor can place all the Java generated classes associated with the ETD.

C Click **OK** and **Finish** to accept the names and open the **ETD Editor** dialog.

Results

Using the builder wizard, you create an ETD in e*Gate that captures all the data in the original data source.

9.7 Task 6: Creating and Registering the ETD Using the Command Line

Note: *This section describes using the command line, not the ETD builder wizard, to build the ETD. This is a required task if using e*Gate version 4.5.1. If you are using e*Gate version 4.5.2 or later, we recommend using the ETD builder wizard to build the ETD.*

In this task, you will use the **runapidemo1** script to create the **Account** ETD and then register the files to the e*Gate system.

Although you installed e*Ways and e*Way Connections to the **default** schema to make them available to all schemas in **"Task 3: Building the e*Way and e*Way Connection" on page 125**, in this case, you must install the ETD to a *specific* schema. In this example, you install the ETD named **Account**.

To set up the e*Gate files associated with the e*Way and e*Way connection

1. Change directories to **C:\gmeeksdk\gmeek\builder\apiDemo\**.
2. If your environment differs from the assumptions noted in the pre-installation steps on page 122, edit the **runapidemo1** script and make appropriate substitutions.
3. Run the **.\runapidemo1** script.

Results

As a result of running the **runapidemo1** script, the following are created:

- .\Account.something—a human-readable text version of the metadata that was extracted. This file is for viewing purposes only; it is not used by any other process.
- ..\..\installETD\ETD4Account\—Directory for files constituting the **Account** ETD:
  - **Account.ctl**—Control file to tell e*Gate how to register and manage the ETD.
  - **Account.jar**—Archive file for classes and the Java source code referenced by the ETD.
  - **Account.xsc**—To allow the ETD to be viewed in e*Gate using the ETD Editor.

To register the ETD with e*Gate

1. Change directories to **C:\gmeeksdk\gmeek\installETD\ETD4Account\**.
2. Enter the following command, all on one line, making the appropriate substitutions as needed for Registry Host, user name, user password, and remote schema:

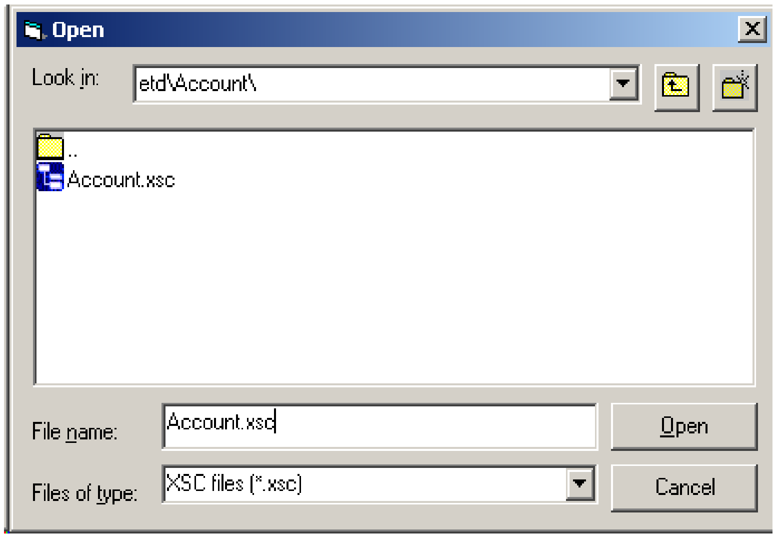
```
stcregutil -rh <host-name> -rs <schema-name> -un <user-name>
-up <password> -fc . -ctl Account.ctl
```

To validate the new ETD within e*Gate

1. Open the schema used in step 2 of **"To register the ETD with e*Gate"**.
2. Open the ETD Editor.
3. Click **File**, **Open**.
4. Click the Account folder.

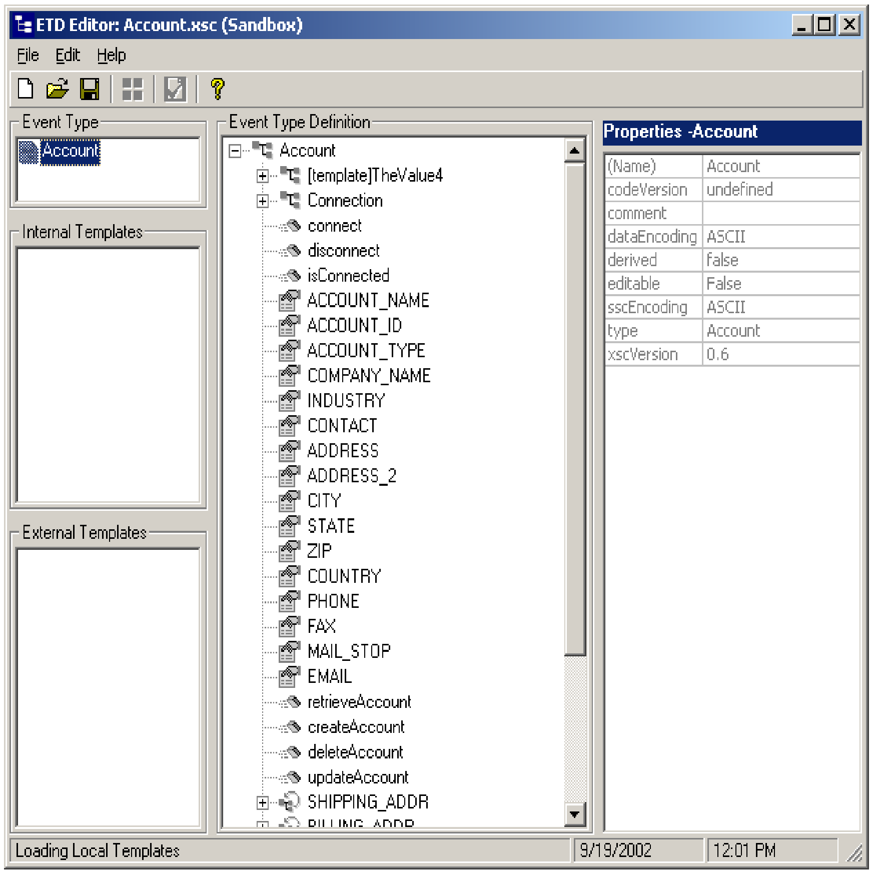
The ETD shown in Figure 43 appears.

Figure 43 Event Type Definition Selection Dialog



Select the **Account.xsc** file. The ETD Editor dialog shown in Figure 44 appears.

Figure 44 **Account.xsc** Viewed in the ETD Editor



(Name)	Account
codeVersion	undefined
comment	
dataEncoding	ASCII
derived	false
editable	False
sscEncoding	ASCII
type	Account
xscVersion	0.6

Verify the following:

- The root node contains fields for holding account name, account ID, and so forth.
- The ETD contains subnodes for holding shipping address, billing address, and payment information, as well as an internal template used by six reference nodes.
- The ETD supplies methods **createAccount()**, **deleteAccount()**, **updateAccount()**, and **retrieveAccount()** that e*Gate end users can invoke to access corresponding APIs in the external system.

9.8 Task 7: Testing Outside of the e*Gate Environment

The **GmeekDemoEway** must be tested outside of the e*Gate environment. This section describes how to validate the APIs outside of e*Gate. The **RmiDemoClient.java** and **RmiDemoSvr.java** files contain the source code for testing the APIs, and the **runRmitest** and **runrmiclient** scripts are provided to run the client and the server.

Running the runRmitest Script for the Rmi Server

To compile the Rmi Server

1. Do one of the following:
   - Invoke the javac:
     ```
     javac rmiDemoSvr.java
     ```
   - Alternatively, you can use the **compile.bat** (or, on UNIX, **compile.sh**) file supplied in **gmeek/builder/rmiDemoSvr**.
2. After the **.java** files have been compiled, run the script by opening a Command Prompt and entering the command **runRmitest**.

Do one of the following:

- On Windows, open a Command Prompt, change to the correct directory and then run the **runRmitest.bat** file by entering the following:
  ```
  .\runRmitest
  ```
- On UNIX, open a shell, change to the correct directory, and then run the **runRmitest.sh** file by entering the following:
  ```
  ./runRmitest.sh
  ```

Note: *Initially, there should not be any output.*

RmiDemoClient.java

An excerpt from the RmiDemoClient.java file is listed as follows.

```
...
  public static void main(String args[])
  throws Exception
  {
```

```
            String url;

            if (args.length == 3 )
            {
              url = new String("//"+args[0]+":"+args[1]+"/RmiDemoSvr");
            }
            else if (args.length == 2)
            {
              url = new String("//"+args[0]+"/RmiDemoSvr");
            }
            else
            {
              RmiDemoClient.usage();
              return;
            }

            // RMI registry lookup
            //
            RmiDemoSvrIntf echoRef = (RmiDemoSvrIntf)Naming.lookup(url);

            // Call echo method on server
            //
            System.out.println(echoRef.sayEcho(args[args.length-1]));

            // Retrive accounts from server
            //
            java.util.List alllist = echoRef.retrieveAllAccountId();

            System.out.println("total number of accounts: "+alllist.size());

            for (int i = 0; i < alllist.size(); i++)
            {
              System.out.println("ACCOUNT_ID: "+(String)alllist.get(i));
            }
        ...
```

Running the runrmiclient Script for the Rmi Server

Once the **.java** files have been compiled, run the **runTester.bat** file by opening a Command Prompt and entering the command **runTester**. This populates the RMI server with data.

Do one of the following:

- On Windows, open a Command Prompt, change to the correct directory and then run the **runrmiclient.bat** file by entering the following:

  ```
  .\runrmiclient
  ```

- On UNIX, open a shell, change to the correct directory, and then run the **runrmiclient.sh** file by entering the following:

  ```
  ./runrmiclient.sh
  ```

The output in the client for runrmiclient should resemble the following:

```
(1)
(2)   Hello lynn!!
(3)
(4)   total number of accounts: 0
(5)   create account: {ACCOUNT_TYPE=saving, ACCOUNT_ID=1000, ZIP=91006,
      CITY=Monrovia, STATE=CA, ADDRESS=103 Main St.}
(6)   ...
```

Note: *You must leave the RMI server running so that the* ***GmeekDemoEway*** *sample schema can query the RMI server for data.*

9.8.1 Rerunning the installEWAY Script

If you modify the source files, including the ETD builder wizard files, or if you modify the **.ctl** file, you must:

- rerun the **installEWAY** script.
- re-validate the sample files within e*Gate.

For instructions on rerunning the **installEWAY** script, see **Table 12 on page 128**.

9.9 Task 8: Understanding the Sample Implemented in a Schema

This section:

- Describes how to load the sample schema included in **GmeekDemoEwaySample.zip**, or for e*Gate version 4.5.1, **GmeekDemoEwaySample_4.5.1.zip**.
- Shows you how the sample components appear to the end user.
- Shows you how the sample components behave in the e*Gate environment.

9.9.1 Importing the GmeekDemoEwaySample.zip Schema

To import the sample schema into e*Gate

1. Start the e*Gate Schema Designer GUI.
2. When the Schema Designer prompts you to log in, select the host that you specified during installation, and enter your password.
3. You are then prompted to select a schema. Click **New**. The New Schema dialog box opens. (Schemas can also be imported or opened from the e*Gate File menu by selecting **New Schema** or **Open Schema**.)
4. Enter a name for the new Schema, for example, **GmeekDemoEwaySample_Schema**, or any name as desired.
5. To import the sample schema select **Create from Export**, and use **Find** to locate and select the **GmeekDemoEwaySample.zip** (or for e*Gate version 4.5.1, **GmeekDemoEwaySample_451.zip**) file on the e*Gate Integrator Installation CD-ROM or from the location it was copied in earlier.

   GmeekDemoEwaySample.zip (or **GmeekDemoEwaySample_451.zip**) is the file supplied in the sample directory.

The e*Gate Schema Designer opens to the new schema. You are now ready to make any configuration changes that may be necessary for this sample schema to run on your specific system.

9.9.2 Sample Data INDATA

If you have not already extracted the **INDATA.zip** file into a temporary directory, follow steps 1 and 2. If you have already done this step, skip steps 1 and 2 and begin with step 3.

1. Extract the **INDATA.zip** file into a temporary directory.
2. Copy the extracted **Account.fin** file into the **C:\INDATA** directory.

Note: *If you want or need to use a location other than **C:\INDATA** (for example, on UNIX), you must change the string "\INDATA" to the correct location in your e*Gate schema's ewAccount configuration.*

The presence of a file with extension **.fin** triggers the file e*Way to read its contents while the e*Way is running.

The **INDATA.zip** file contains the **Account.fin** file, as follows.

- **Account.fin**:

```
1000
```

Note: *If the RMI server is not already running, you must run the runRmitest script, and run the runrmiclient script. For details on running the runRmitest script, see* **"Running the runRmitest Script for the Rmi Server" on page 151**. *For details on running the runrmiclient script, see* **"Running the runrmiclient Script for the Rmi Server" on page 152**.

3. From a Command Prompt, enter the following command to have the e*Gate Control Broker (**stccb.exe**) start the imported sample schema:

```
stccb -rh localhost -un Administrator -up STC -ln localhost_cb -rs
<schema name>
```

To use a Registry Host other than **localhost**, a username/password combination other than **Administrator**/**STC**, and/or a logical Control Broker name other than **localhost_cb**, make the appropriate substitutions.

Results

Once the schema is run, the expected output results should match the input. The output is specified by the ewEater configuration. Successful execution creates a file named **AccountOut0.dat**, **AccountOut1.dat**, **AccountOut2.dat**...or a similar file name in **C:\OUTDATA** specified by the e*Way Connection.

The contents of the file should be:

```
ACCOUNT_ID:1000
ACCOUNT_TYPE:saving
```

Chapter 10

Developing the Automatic e*Way Connection

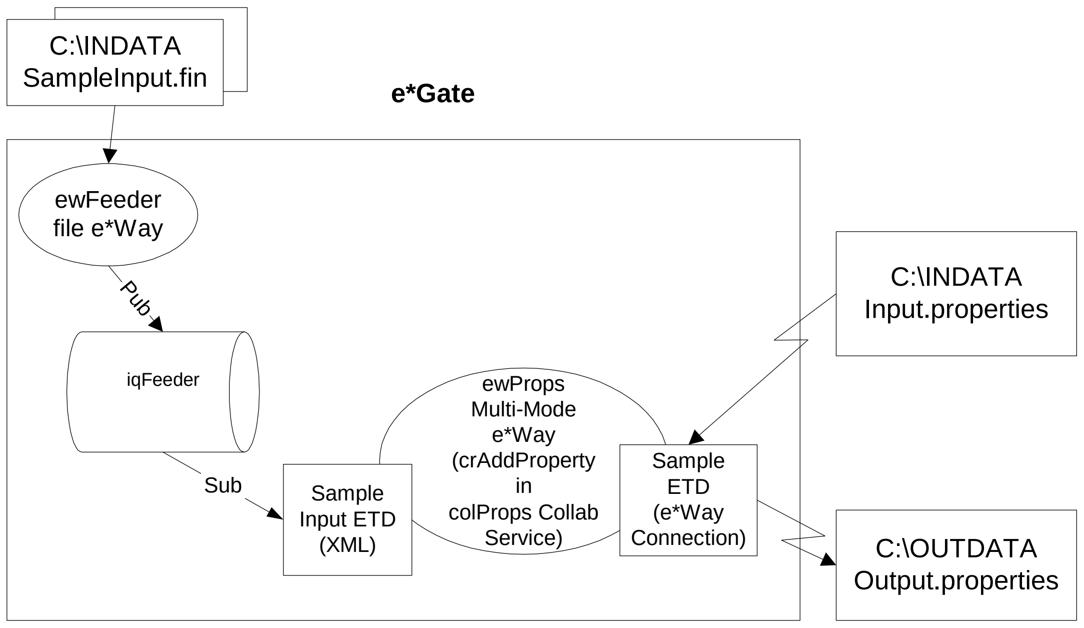
The *Generic Multi-Mode e*Way Extension Kit* provides a complete set of files for setting up a sample e*Way Connection that handles connection/disconnection automatically: an "Automatic Connection" e*Way Connection. This chapter takes you step by step through the process of modifying the files, compiling them, placing them in the correct locations, and validating them in the environment of an e*Gate schema.

The final section focuses on the sample schema, showing you how the user-created ETD e*Way Connection fits into e*Gate so you can match up your development efforts with features seen by end users.

10.1 Overview

Figure 45 shows the architecture of the Automatic Connection sample.

Figure 45 Architecture of the Automatic Connection Sample



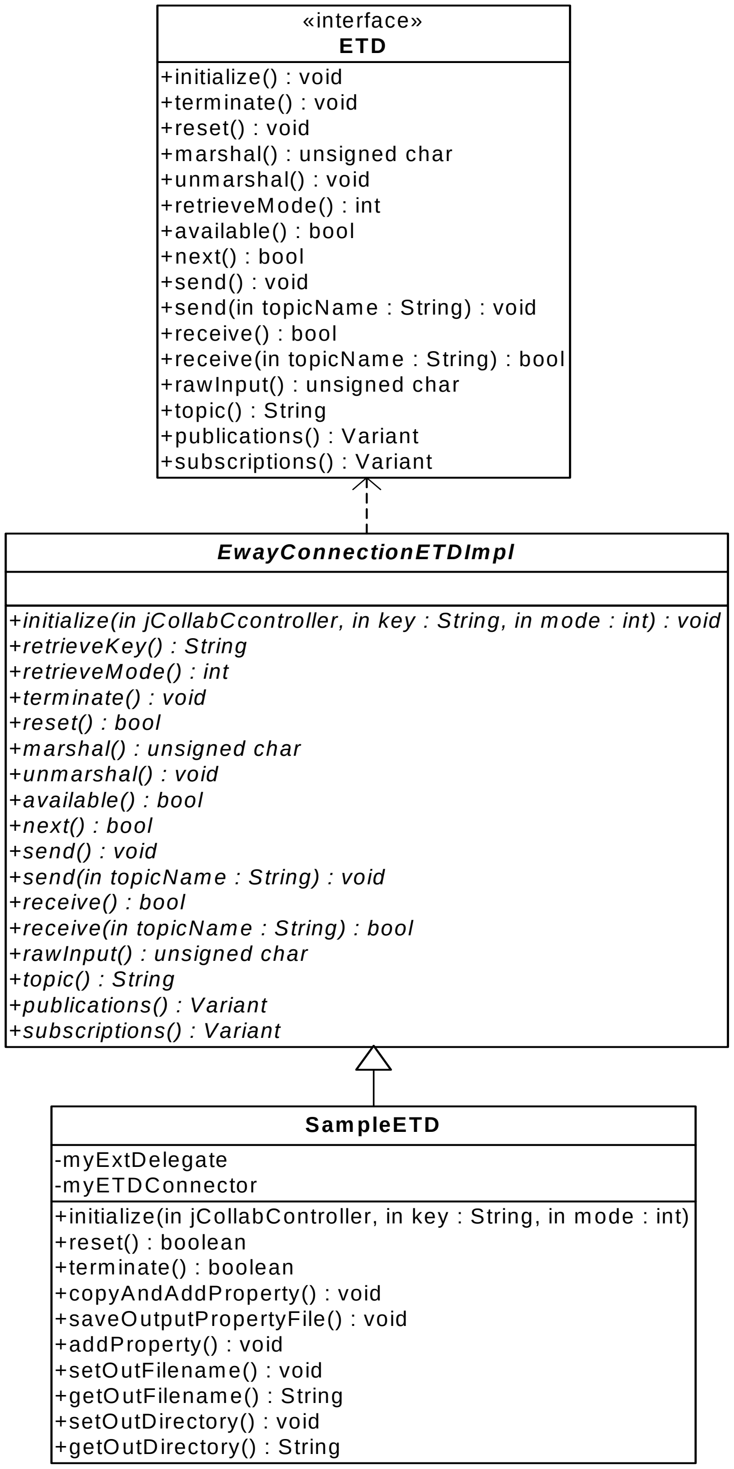
10.2 Classes and Interactions for the Automatic Connection Sample

The Automatic Connection sample uses an ETD class named **SampleETD** and a connector class named **SampleETDConnector**.

10.2.1 ETD Class

Figure 46 shows the class diagram for the ETD class named **SampleETD**. The *EwayConnectionETDImpl* interface extends the *ETD* interface to allow setting and getting the Connector object associated with an ETD. The configuration associated with the ETD can also be obtained as an object. These are needed mainly to allow the user to set connection parameters on the ETD. The corresponding changes in the **.xsc** file are discussed in **"Editing/Viewing the .xsc File" on page 176**.

Figure 46 ETD Class (SampleETD) for the Automatic Connection Sample

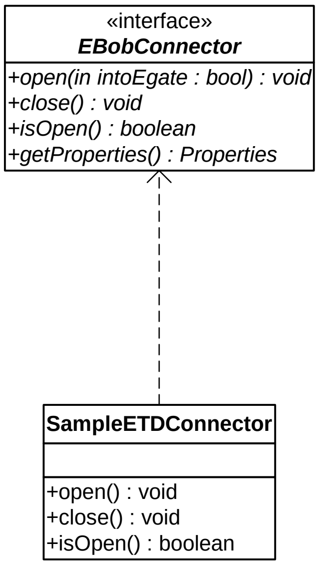

«interface»
ETD
+initialize() : void
+terminate() : void
+reset() : void
+marshal() : unsigned char
+unmarshal() : void
+retrieveMode() : int
+available() : bool
+next() : bool
+send() : void
+send(in topicName : String) : void
+receive() : bool
+receive(in topicName : String) : bool
+rawInput() : unsigned char
+topic() : String
+publications() : Variant
+subscriptions() : Variant
EwayConnectionETDImpl
+initialize(in jCollabCcontroller, in key : String, in mode : int) : void
+retrieveKey() : String
+retrieveMode() : int
+terminate() : void
+reset() : bool
+marshal() : unsigned char
+unmarshal() : void
+available() : bool
+next() : bool
+send() : void
+send(in topicName : String) : void
+receive() : bool
+receive(in topicName : String) : bool
+rawInput() : unsigned char
+topic() : String
+publications() : Variant
+subscriptions() : Variant
SampleETD
-myExtDelegate
-myETDConnector
+initialize(in jCollabController, in key : String, in mode : int)
+reset() : boolean
+terminate() : boolean
+copyAndAddProperty() : void
+saveOutputPropertyFile() : void
+addProperty() : void
+setOutFilename() : void
+getOutFilename() : String
+setOutDirectory() : void
+getOutDirectory() : String

10.2.2 Connector Class

The Automatic Connection sample uses a connector class named

SampleETDConnector. Its class diagram is shown in Figure 47.

Figure 47 Connector Class (SampleETDConnector)



10.3 Overview of the Automatic Connection Sample

Implementing and validating the Automatic Connection sample requires these steps:

- Ensure your environment meets the prerequisites, and then load and unzip the sample source/install files in **gmeek.taz**.
- Review or edit the **.java** files to understand the logic contained within the code.
- Edit the **compile.bat** script (or, on UNIX, **compile.sh**) to reflect your development environment.
- Run the compile script to compile the **.java** files and create **.jar** files.
- Edit the **.ctl** and **.def** files to reflect both your environment and the functionality required.
- Edit the **.xsc** file to understand the logic required for it to perform correctly.
- Optionally, edit the **test.prop** file and then run the **runTester.bat** file to test the sample class outside of e*Gate.
- Start the Schema Designer and create a schema into which to commit the sample.
- Run the **installETD** script to make the sample files available to e*Gate.
- Return to the Schema Designer and validate the results of the preceding steps.
- Import the sample schema into e*Gate, start the Schema Manager, and validate the behavior of the sample ETD and sample e*Way Connection.

10.4 Installing the Sample

The installation package for the Automatic Connection sample comprises the following files:

- **gmeek.taz**
- **MySchema.zip**
- **INDATA.zip**

To install the files

1. Copy the **.taz/.zip** files to a temporary directory.
2. Extract the **gmeek.taz** and **INDATA.zip** files to any convenient location. The path location of these files will be used in scripts later.

   Changes to the directory locations of the unzipped files will require changes to the supplied scripts. For this reason, it is recommended that you complete the sample before making any changes to directory locations or file names.

10.5 Setting Up the Automatic Connection Sample Files

In this section, the details for editing the samples scripts and creating the **.java** files are broken down into steps:

- **"Editing/Viewing the .java Files" on page 159**
- **"Customizing the Compile Script" on page 174**
- **"Compiling the .java Files and Creating the .jar File" on page 175**
- **"Editing/Viewing the .ctl Files" on page 175**
- **"Editing/Viewing the .def Files" on page 176**
- **"Editing/Viewing the .xsc File" on page 176**

The **gmeek.taz** file contains all the sample code required to implement and validate a sample e*Way Connection of the following varieties:

- Automatic Connection
- Connection Management
- Transactional

See **"Product Files Contained in File gmeek.taz" on page 26** for the contents of the **gmeek.taz** file.

10.5.1 Editing/Viewing the .java Files

To deploy the sample schema contained in **MySchema.zip,** the files that are provided require only minor edits. This section provides the information necessary to view the code, edit and compile it, and commit files to the e*Gate Registry. The code samples are provided to further your understanding. The sample code is described in sections that describe its purpose.

The source code files for the Automatic Connection sample are as follows:

- SampleETD.java
- SampleETDConnector.java
- SampleETDDefs.java
- SampleETDExternalClass.java
- SampleETDExternalException.java
- SampleETDTester.java

The following sections discuss each of these files in detail.

SampleETD.java

An e*Way Connection has an associated class that implements the *ETD* interface. We refer to this class as your ETD class. The class **EwayConnectionETDImpl** is a sample provided to implement the *ETD* interface. **EwayConnectionETDImpl** is an abstract class that your ETD class must extend, and it provides the default implementation used for e*Way Connection (non-messageable) ETDs. The **EwayConnectionETDImpl** class should not normally be modified. Any additional desired functionality should be included in your ETD class.

By extending **EwayConnectionETDImpl**, your ETD class inherits common behavior when interacting with e*Gate IQs and Oracle SeeBeyond JMS IQ Managers.

Note: *Since it is not intended for message parsing, **EwayConnectionETDImpl** contains empty implementations of the **marshal()** and **unmarshal()** methods.*

To modify SampleETD.java

The following steps show the possible changes that you may need to make to **SampleETD.java**. These steps only show the sections of code that you will modify. See page 163 for the entire contents of **SampleETD.java**.

1. The following source code, from **SampleETD.java**, shows a sample ETD class skeleton.

```
package sample;

import java.util.Properties;

import com.stc.common.collabService.EGate;
import com.stc.common.collabService.JCollabController;
import com.stc.common.collabService.CollabConnException;
import com.stc.common.collabService.CollabDataException;
import com.stc.common.collabService.CollabResendException;
import com.stc.common.utils.ScEncrypt;
import com.stc.jcsre.EBobConnectorFactory;


public class SampleETD extends EwayConnectionETDImpl
```

The **import** statements (1) through (8) are required. They import the e*Gate core classes that are used in a typical ETD class.

2 Create a delegate object in your ETD class.

The delegate object is an instance of the **Delegate** class, which wraps the calls to an external's Java API. Use of a **Delegate** class is not mandatory, but it is recommended as a best practice.

```
(15)  myExtDelegate = new SampleETDExternalClass();
(16)  }
(17)
(18)  /**
(19)  * Called by external (collab service) to initialize object.
(20)  * Reads configuration from .cfg file, obtains a Connector
(21)  * object used to establish connection with the external system
(22)  * through the connector factory, initializes myExtDelegate.
(23)  *
(24)  * @param     cntrCollab  The Java Collaboration Controller
(25)  * object.
(26)  * @param     key
(27)  * @param     mode
(28)  * @see       com.stc.jcsre.ETD
(29)  *
(30)  * @author    SeeBeyond
(31)  */
```

3 Override the **initialize()** method of your ETD class. Start by calling **super.initialize()** to ensure you perform the default implementation's initialization code.

```
(32)  public void initialize(JCollabController cntrCollab, String key, int
      mode)
(33)  (46) throws CollabConnException, CollabDataException
(34)  (47) {
(35)  (48) EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
(36)  (49) "Inside SampleETD.initialize()");
(37)  (50)
(38)  (51) super.initialize(cntrCollab, key, mode);
(39)  (52)
```

4 Instantiate your connector class. Use the **EBobConnectorFactory** class to do this. The **createConnector()** method of this factory loads the configuration specified by the user from the GUI (through the **.cfg** file). The configuration is saved as a Java Properties object which becomes associated with the Connector object.

```
(40)  EBobConnectorFactory connFactory = new EBobConnectorFactory();
(41)
(42)  myETDConnector = (SampleETDConnector)
(43)  connFactory.createConnector(cntrCollab, key, mode);
(44)
(45)  EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
(46)  "Created SampleETDConnector from EBobConnectorFactory.");
```

5 Initialize your ETD class using the Properties object from the configuration file. After the connector class is instantiated and the associated properties are set, call the **getProperties()** method and start obtaining configuration values from it.

```
(47)  //Extract properties from .cfg file
(48)  //
(49)  this.cfgProps = myETDConnector.getProperties();
(50)  ////////////////////////////////////////////////////////////////
(51)  // outFilename
```

```
////////////////////////////////////////////////////////////////
    String propsFilename =
cfgProps.getProperty(SampleETDDefs.ETD_DEF_PROP_NAME_FILENAME);
    try {
      if (propsFilename != null)
      {
        this.outFilename = propsFilename;
        EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
          "Default outFilename set to " + propsFilename);
      }
    }
    catch (Exception ex)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
        "Failed to set the default filename; Exception : " +
ex.toString());
     throw new CollabConnException("Failed to set the default filename;
Exception : " + ex.toString());
    }

```

6 Override the **reset()** method of your ETD class.

Note: *Returning **true** indicates that the ETD will not be re-instantiated on the next Event. Returning **false** indicates that the re-instantiation will take place for each Event.*

```
///////////////////////////////////////////////////////////////
//                                                           //
//    reset                                                  //
//                                                           //
///////////////////////////////////////////////////////////////

/**
* Clears settings for document type, recipient, sender,
* destination, xmlString, synchronous response string.
*
* @return true - don't re-instantiate this ETD on next event
*         false - causes re-instantiation
*/
public boolean reset()
{
EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
"reset() has been called!");

myExtDelegate.reset();

return true;
}

```

7 Override the **terminate()** method of your ETD class.

```
///////////////////////////////////////////////////////////////
//                                                           //
//    terminate                                              //
//                                                           //
///////////////////////////////////////////////////////////////

/**
* Closes external connection if NOT in a Subcollaboration Rule.
* If in Subcollaboration, you must release resources used in
* the Subcollaboration but don't close your external
* connection.
```

```
*/
public void terminate()
{
EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
"terminate() has been called!");

try {
if (myETDConnector.isOpen())
myETDConnector.close();
}
catch (com.stc.jcsre.EBobConnectionException ex)
{
}
}

```

SampleETD.java listing

The next few pages contain the source code for the ETD class (**SampleETD.java)** in its entirety.

Note: *This section of source code displays an unencrypted password. It is purposely coded this way to show you how to use the decrypt() method. However, Oracle recommends not to use unencrypted passwords in a live production environment.*

```
package sample;

import java.util.Properties;

import com.stc.common.collabService.EGate;
import com.stc.common.collabService.JCollabController;
import com.stc.common.collabService.CollabConnException;
import com.stc.common.collabService.CollabDataException;
import com.stc.common.collabService.CollabResendException;
import com.stc.common.utils.ScEncrypt;
import com.stc.jcsre.EBobConnectorFactory;


public class SampleETD extends EwayConnectionETDImpl
{
  private String outDirectory  = null;
  private String outFilename   = null;

  private Properties              cfgProps   = null;
  private SampleETDExternalClass  myExtDelegate = null;
  private SampleETDConnector      myETDConnector = null;

  /**
   * SampleETD constructor
   */
  public SampleETD()
  {
    myExtDelegate = new SampleETDExternalClass();
  }

  /**
   * Called by external (collab service) to initialize object.
   * Reads configuration from config file, obtains a Connector object
   * used to establish connection with the external system through the
   * connector factory, initializes myExtDelegate.
   *
   * @param     cntrCollab  The Java Collaboration Controller object.
   * @param     key
   * @param     mode
   * @see       com.stc.jcsre.ETD
   *
   * @author    SeeBeyond
   */
  public void initialize(JCollabController cntrCollab, String key, int
mode)
    throws CollabConnException, CollabDataException
  {
    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
      "Inside SampleETD.initialize()");

    super.initialize(cntrCollab, key, mode);

    EBobConnectorFactory connFactory = new EBobConnectorFactory();

myETDConnector = (SampleETDConnector)
connFactory.createConnector(cntrCollab, key, mode);

    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
      "Created SampleETDConnector from EBobConnectorFactory.");

    // Extract properties from .cfg file
    //
    this.cfgProps = myETDConnector.getProperties();

    ////////////////////////////////////////////////////////////////
    // outFilename
    ////////////////////////////////////////////////////////////////
    String propsFilename =
cfgProps.getProperty(SampleETDDefs.ETD_DEF_PROP_NAME_FILENAME);
    try {
      if (propsFilename != null)
      {
        this.outFilename = propsFilename;
        EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
          "Default outFilename set to " + propsFilename);
      }
    }
    catch (Exception ex)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
        "Failed to set the default filename; Exception : " +
ex.toString());
     throw new CollabConnException("Failed to set the default filename;
Exception : " + ex.toString());
    }

    ////////////////////////////////////////////////////////////////
    // outDirectory
    ////////////////////////////////////////////////////////////////
    String propsDirectory =
cfgProps.getProperty(SampleETDDefs.ETD_DEF_PROP_NAME_DIRECTORY);
    try {
      if (propsDirectory != null)
      {
        this.outDirectory = propsDirectory;
        EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
          "Default outDirectory set to " + propsDirectory);
      }
    }
    catch (Exception ex)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
        "Failed to set the default directory; Exception : " + ex.toString());
      throw new CollabConnException("Failed to set the default directory; Exception : " + ex.toString());
    }

    ////////////////////////////////////////////////////////////////
    // Username (NOT REQUIRED)
    ////////////////////////////////////////////////////////////////
```

Note: ***Username (NOT REQUIRED)*** *indicates that the username is not required in this sample.*

```
String propsUsername = cfgProps.getProperty(SampleETDDefs.ETD_DEF_PROP_NAME_USERNAME);
    if (propsUsername != null)
    {
      ////////////////////////////////////////////////////////////////
      // password (NOT REQUIRED)
      ////////////////////////////////////////////////////////////////
```

Note: ***Password (NOT REQUIRED)*** *indicates that the password is not required in this sample.*

```
      String propsPassword = cfgProps.getProperty(SampleETDDefs.ETD_DEF_PROP_NAME_PASSWORD);
      try
      {
        // Decrypt the encrypted password here.
        String passWordDecrypt = null;
        passWordDecrypt = ScEncrypt.decrypt(propsUsername, propsPassword);
        if (passWordDecrypt != null)
        {
         EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
            "Stored password is " + passWordDecrypt);
        }
        else
        {
          EGate.traceln (EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
            "Error decrypting Password.");

          throw new CollabConnException("Error decryping Password.");
        }
      }
      catch (Exception ex)
      {
        EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR, ex.toString());
        throw new CollabConnException (ex.toString());
      }
    }

    ////////////////////////////////////////////////////////////////
    // Do some Initialization
    ////////////////////////////////////////////////////////////////
    try {
      if (myExtDelegate == null)
        myExtDelegate = new SampleETDExternalClass();
        myETDConnector.setExternalClass(myExtDelegate);

      // You can call any initialization methods in myExtDelegate here.

    }
    catch (Exception e)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
                    "Exception caught initializing external.");
      e.printStackTrace();
      throw new CollabConnException("Exception caught initializing
external; Exception : " + e.toString());
    }
  }

  ///////////////////////////////////////////////////////////////
  //                                                           //
  //    Getter/setter methods for attributes exposed in ETD    //
  //                                                           //
  ///////////////////////////////////////////////////////////////

  /**
   * Call this in your Collaboration to set the outFilename attribute.
   *
   * @param     filename - filename
   */
  public void setOutFilename(String filename)
  {
    this.outFilename = filename;
  }

  /**
   * Call this in your Collaboration to get the outFilename attribute.
   *
   * @return    filename - filename
   */
  public String getOutFilename()
  {
    return(this.outFilename);
  }

  /**
   * Call this in your Collaboration to set the outDirectory attribute.
   *
   * @param     directory - directory name
   */
  public void setOutDirectory(String directory)
  {
    this.outDirectory = directory;
  }

  /**
   * Call this in your Collaboration to get the outDirectory attribute.
   *
   * @return    directory - directory name
   */
  public String getOutDirectory()
  {
    return(this.outDirectory);
  }

  ///////////////////////////////////////////////////////////////
  //                                                           //
  //    reset                                                  //
  //                                                           //
  ///////////////////////////////////////////////////////////////

  /**
   * Clears settings for document type, recipient, sender, destination,
   * xmlString, synchronous response string.
   *
   * @return true - don't re-instantiate this ETD on next event
   *         false - causes re-instantiation
   */
  public boolean reset()
  {
    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
      "reset() has been called!");

    myExtDelegate.reset();

    return true;
  }

  ///////////////////////////////////////////////////////////////
  //                                                           //
  //    terminate                                              //
  //                                                           //
  ///////////////////////////////////////////////////////////////

  /**
   * Closes external connection if NOT in a Subcollaboration Rule.
   * If in Subcollaboration, you must release resources used in
   * the Subcollaboration but don't close your external connection.
   *
   */
  public void terminate()
  {
    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
      "terminate() has been called!");

    try {
      if (myETDConnector.isOpen())
        myETDConnector.close();
    }
    catch (com.stc.jcsre.EBobConnectionException ex)
    {
    }
  }


  ///////////////////////////////////////////////////////////////
  //                                                           //
  //    Methods exposed in ETD                                 //
  //                                                           //
  ///////////////////////////////////////////////////////////////

  /**
   * Call this in your collaboration to copy the properties from the
   * input file and add a new property variable/value to it.
   *
   */
  public void copyAndAddProperty(String inputFilePath, String var,
String val)
    throws CollabConnException, CollabDataException,
CollabResendException
  {
    try {
      myExtDelegate.setPropFilePath(getOutDirectory() + "/" +
getOutFilename());
      myExtDelegate.copyAndAddProperty(inputFilePath, var, val);
    }
    catch (Exception e) {
      throw new CollabDataException(e.toString());
    }
  }

  public void addProperty(String var, String val)
    throws CollabConnException, CollabDataException, CollabResendException
  {
    try {
      myExtDelegate.getOutProperties().setProperty(var, val);
    }
    catch (Exception e) {
      throw new CollabDataException(e.toString());
    }
  }

  /**
   * Call this in your collaboration to save the file associated with the
   * ETD object attributes outDirectory and outFilename.
   *
   */
  public void saveOutPropertyFile()
    throws CollabConnException, CollabDataException, CollabResendException
  {
    try {
      myExtDelegate.open();
      if (myExtDelegate.isOpen())
        myExtDelegate.save();
      myExtDelegate.close();
    }
    catch (Exception e) {
      throw new CollabDataException(e.toString());
    }
  }

}
```

SampleETDConnector.java

The implementation of your e*Way Connection's configuration and connection management functions must be provided in a class which implements **EBobConnector**. This class is referred to as your ETD's **Connector** class.

The source code file for **SampleETDConnector.java** creates your ETD's **Connector** class.

```
package sample;

import java.util.Properties;

import com.stc.common.collabService.*;
import com.stc.jcsre.*;

public class SampleETDConnector implements EBobConnector
{
protected Properties props;
private   SampleETDExternalClass extClass = null;

/**
```

```
* Constructs an SampleETDConnector
*
* @param  props  A Properties object.
*/
public SampleETDConnector(Properties props)
{
this.props = props;
}

/**
* Opens the connector for accessing the external system.
*
* @param  intoEgate <code>true</code> if connector is to
* subscribe for events initially from an external and inbound to
* e*Gate;
* <code>false</code> if connector is to publish events
* outbound from e*Gate and to an external
*
* @see    com.stc.jcsre.EbobConnector
*
* @throws com.stc.jcsre.EBobConnectionException when connection
* problems occur.
*/
public void open(boolean intoEgate)
throws com.stc.jcsre.EBobConnectionException
{
if (extClass == null)
{
throw new EBobConnectionException("External class is null");
}

// Implement opening connection to external system
//
String filename =
props.getProperty(SampleETDDefs.ETD_DEF_PROP_NAME_FILENAME);

String directory =
props.getProperty(SampleETDDefs.ETD_DEF_PROP_NAME_DIRECTORY);

String filepath = directory + "/" + filename;

extClass.setPropFilePath(filepath);
extClass.open();
}

/**
* Closes the connector to the external system and releases
* resources.
* @see      com.stc.jcsre.EbobConnector
*
* @throws    com.stc.jcsre.EBobConnectionException When
* connection problems occur.
*/
public void close() throws com.stc.jcsre.EBobConnectionException
{
// Implement closing connection to external system
//
if (extClass != null)
extClass.close();
}

/**
* Verifies that the connector to the external system is still
* available.
* @return  <code>true</code> if the connector is still open and
* available;
* <code>false</code> otherwise.
*
* @see       com.stc.jcsre.EbobConnector
*
* @exception com.stc.jcsre.EBobConnectionException
* When connection problems occur.
*/
public boolean isOpen() throws
com.stc.jcsre.EBobConnectionException
{
// Implement returning if connection to external system is open
//
if (extClass != null)
return extClass.isOpen();
else
return false;
}

/**
* Retrieves the connection properties (stored by constructor)
* used by the
* connector to access the external.
*
* @return  Connection properties of the external system.
*/
public java.util.Properties getProperties()
{
return props;
}

/**
* Set to the delegate external class instance by the
*/
public void setExternalClass(SampleETDExternalClass extClassInstance)
{
this.extClass = extClassInstance;
}
}
```

SampleETDDefs.java

The source code file **SampleETDDefs.java** defines the string constants and property names to be pulled in from the default configuration-file template (**.def** file).

```
package sample;

public class SampleETDDefs {

// Property names from eway connection config file
//
public static final String ETD_DEF_PROP_NAME_DIRECTORY =
"External_Configuration.Directory";

public static final String ETD_DEF_PROP_NAME_FILENAME =
"External_Configuration.Filename";

public static final String ETD_DEF_PROP_NAME_USERNAME =
"External_Configuration.Username";

public static final String ETD_DEF_PROP_NAME_PASSWORD =
"External_Configuration.Password";

}

```

SampleETDExternalClass.java

The source code file **SampleETDExternalClass.java** creates a class for interfacing with the external system.

```
package sample;

import java.io.*;
import java.util.Properties;

import com.stc.common.collabService.*;

/**
* SampleETDExternalClass
* Sample to illustrate interface to an external.
*/
public class SampleETDExternalClass {

private Properties       myProp = null;
private FileOutputStream myOut = null;
private String           propFilePath = null;

/**
* SampleETDExternalClass constructor
*/
public SampleETDExternalClass()
{
}

public void reset()
{
}

public void copyAndAddProperty(String inputPath, String var, String
value)
throws SampleETDExternalException {

try {

EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
"Input Properties file is " + inputPath);

FileInputStream fis = new FileInputStream(inputPath);

// load file as Properties object
//
Properties tranProps = new Properties();
tranProps.load(fis);
fis.close();
fis = null;
tranProps.setProperty(var, value);
this.setOutProperties(tranProps);

// save to output prop file specified in this.propFilePath
if (propFilePath == null)
throw new SampleETDExternalException("Output prop file path not set.");

}
catch(Exception e) {
```

```
throw new SampleETDExternalException("Error copying " +
inputPath);
}
}
public void setPropFilePath(String filePath)
{
this.propFilePath = filePath;
}

/* Save properties in file
*/
public boolean save()
{
try {
if (myOut == null)
this.open();

this.myProp.store(myOut, null);
}
catch (Exception e) {
EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
e.toString());
return false;
}
return true;
}

/* Set output properties
*/
public void setOutProperties(Properties prop)
{
myProp = prop;
}

/* Get output properties
*/
public Properties getOutProperties()
{
return myProp;
}

/* output file open
*/
public boolean open()
{
try {
if (myOut == null)
myOut = new FileOutputStream(propFilePath);

return true;
}
catch (Exception e) {
EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
e.toString());
return false;
}
}

/* output file close
*/
public boolean close()
{
if (myOut != null)
{
```

```
try {
myOut.close();
myOut = null;
}
catch (Exception e) {
EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
e.toString());
return false;
}
}
return true;
}

/* output file isOpen
*/
public boolean isOpen()
{
if (myOut != null)
return true;
else
return false;
}

}

```

SampleETDExternalException.java

The source code file **SampleETDExternalException.java** defines the exception class.

```
package sample;

import com.stc.eways.exception.*;

public class SampleETDExternalException extends STCDataException
{
public SampleETDExternalException()
{
super();
}

public SampleETDExternalException(String ex)
{
super(ex);
}

public SampleETDExternalException(String ex, Exception e)
{
super(ex, e);
}
}

```

SampleETDTester.java

The source code file **SampleETDTester.java** creates an optional stand-alone test to be used to test the **sampleETDExternalClass** class.

```
package sample;

import java.io.*;

```

```
public class SampleETDTester {
private SampleETDExternalClass extCall = null;

public SampleETDTester() {
extCall = new SampleETDExternalClass();
}

public void test(String inputPropFile,
String var,
String val,
String outputPropFile) {
try {

// Set the output file
//
extCall.setPropFilePath(outputPropFile);

// Copy and add a property to the Properties object from
// the input file into the new output Properties.
//
extCall.copyAndAddProperty(inputPropFile, var, val);

// Save the new Properties copy with the new property into
// the output Properties file specified using setPropFilePath.
//
// This illustrates the typical interface to an external
// system involving procedure that does open external, do work
// with external, close external.
//
extCall.open();
if (extCall.isOpen())
extCall.save();
extCall.close();
}
catch (Exception e) {
}
}

public static void main(String args[]) {
SampleETDTester tester = new SampleETDTester();

System.out.println("");
System.out.println("---------------------------------------");
System.out.println("Input Properties file : " + args[0]);
System.out.println("Property variable     : " + args[1]);
System.out.println("Property value        : " + args[2]);
System.out.println("Output Properties file: " + args[3]);
System.out.println("---------------------------------------");
System.out.println("");
tester.test(args[0], args[1], args[2], args[3]);
}
}

```

10.5.2 Customizing the Compile Script

The kit provides a **compile** script (**compile.bat** on Windows; **compile.sh** on UNIX) to set CLASSPATH information and to create a **.jar** file for the compiled **.java** files upon completion.

```
set GMEEK_EXTRACTDIR=C:\gmeekjars
set JAVA_PATH=C:\jdk1.3.1_02\bin
```

```
set
MYCLASSPATH="%GMEEK_EXTRACTDIR%\classes\stcjcs.jar;%GMEEK_EXTRACTDIR%\c
lasses\stcexception.jar;%GMEEK_EXTRACTDIR%\classes\stcutil.jar;"

%JAVA_PATH%\javac -classpath %MYCLASSPATH% -d . *.java

@REM
@REM jar up the classes
@REM
%JAVA_PATH%\jar cvf ..\installETD\SampleETD\SampleETD.jar
sample\*.class

@REM
@REM jar up the source files to allow for debugger to use;
@REM please remove the following for a release version
@REM
copy *.java sample
%JAVA_PATH%\jar uvf ..\installETD\SampleETD\SampleETD.jar sample\*.java
del sample\*.java
```

You may need to make one or more of the following edits:

1. `set GMEEK_EXTRACTDIR=C:\gmeekjars`

   If your e*Gate installation resides anywhere other than the root **\eGate** directory on your **C** drive, specify the correct location.

2. `JAVA_PATH=c:\jdk1.3.1_02\bin`

   Specify the correct path location for your JDK.

3. When creating e*Way Connections from scratch, modify the directory locations in **compile.bat** (or, on UNIX, **compile.sh**) as needed.

10.5.3 Compiling the .java Files and Creating the .jar File

Do one of the following:

- On Windows, open a Command Prompt, change to the correct directory and then run the **compile.bat** file by entering the following:

  ```
  .\compile.bat
  ```

- On UNIX, open a shell, change to the correct directory, and then run the **compile.sh** file by entering the following:

  ```
  ./compile.sh
  ```

The script is designed to set the CLASSPATH and create the **.jar** file. The **.jar** file is then saved to the **gmeek\installETD\SampleETD** directory.

10.5.4 Editing/Viewing the .ctl Files

The **SampleETD.ctl** file contains information required by the GUI to be able to successfully load the ETD. Any **.jar** files that are required by the ETD must be included here.

```
##------------------------------------------------------------------
#
```

```
#  SampleETD.ctl  (The ETD CTL file)
#
#  This CTL file is used by the GUIs.  It specifies the JAR
#  files that are needed by your ETD classes for compilation
#  (in a Collaboration).  It also specifies the JAR files needed
#  at run time.
#
##-----------------------------------------------------------------

##-----------------------------------------------------------------
#   JAR files containing the classes associated with your ETD
#
##-----------------------------------------------------------------
SampleETD.jar,etd/SampleETD,FILETYPE_BINTEXT
stcexception.jar,classes,FILETYPE_BINTEXT

##-----------------------------------------------------------------
#   Third-party JAR files used (listed as an example only)
#
##-----------------------------------------------------------------
jcert.jar,ThirdParty/jsse/jsse1.0.2/classes,FILETYPE_BINTEXT
jnet.jar,ThirdParty/jsse/jsse1.0.2/classes,FILETYPE_BINTEXT
jsse.jar,ThirdParty/jsse/jsse1.0.2/classes,FILETYPE_BINTEXT
```

10.5.5 Editing/Viewing the .def Files

End users configure e*Ways using the e*Way Configuration Editor, a graphical user interface (GUI) that enables one to change configuration parameters quickly and easily. The e*Way Configuration Editor uses the default configuration-file template (**.def** file) to classify each parameter by its type and name, and can specify other information as well, such as the range of permissible options for a given parameter.

The Configuration Editor stores the values that you assign to those parameters within two configuration files. Each configuration file contains similar information, but the two are formatted differently:

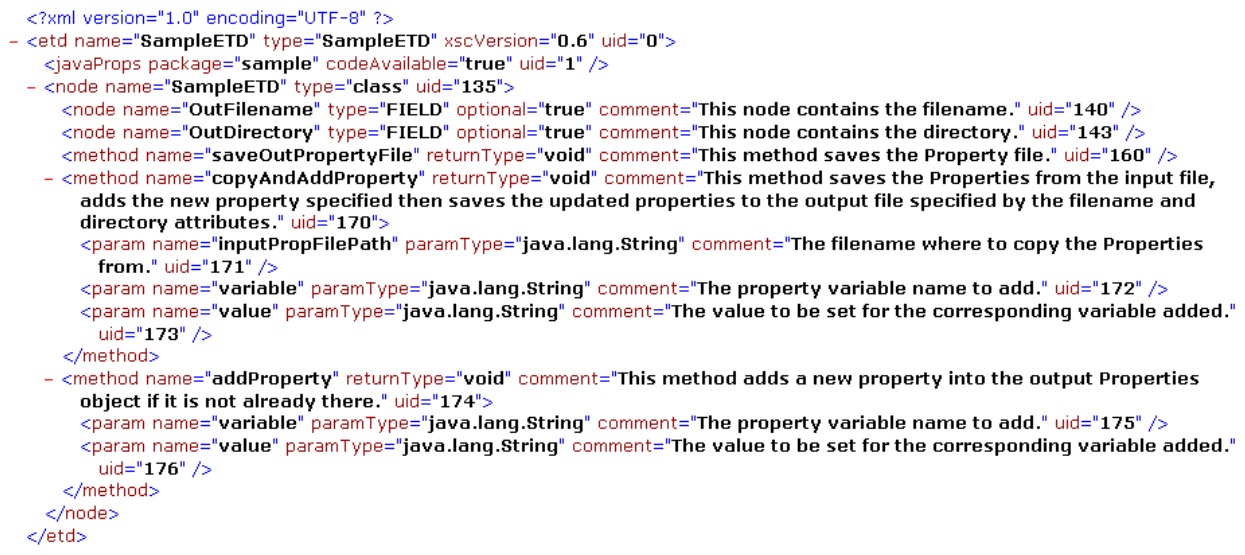
- The **.cfg** file contains the parameter values in delimited records and is parsed by the e*Way at run time.
- The **.sc** file contains the parameter values and additional information needed by the GUI.

The e*Way Editor loads the **.sc** file—not the **.cfg** file—when you edit the configuration settings for an e*Way. Both configuration files are generated automatically by the e*Way Configuration Editor whenever the configuration settings are saved.

For more information on creating a custom **.def** file, see **Appendix A “Extending the .def File” on page 256**.

10.5.6 Editing/Viewing the .xsc File

The **sampleETD.xsc** file contains the information provided by the **.java** files and required by the GUI.



```
  <?xml version="1.0" encoding="UTF-8" ?>
- <etd name="SampleETD" type="SampleETD" xscVersion="0.6" uid="0">
    <javaProps package="sample" codeAvailable="true" uid="1" />
  - <node name="SampleETD" type="class" uid="135">
      <node name="OutFilename" type="FIELD" optional="true" comment="This node contains the filename." uid="140" />
      <node name="OutDirectory" type="FIELD" optional="true" comment="This node contains the directory." uid="143" />
      <method name="saveOutPropertyFile" returnType="void" comment="This method saves the Property file." uid="160" />
    - <method name="copyAndAddProperty" returnType="void" comment="This method saves the Properties from the input file,
        adds the new property specified then saves the updated properties to the output file specified by the filename and
        directory attributes." uid="170">
        <param name="inputPropFilePath" paramType="java.lang.String" comment="The filename where to copy the Properties
          from." uid="171" />
        <param name="variable" paramType="java.lang.String" comment="The property variable name to add." uid="172" />
        <param name="value" paramType="java.lang.String" comment="The value to be set for the corresponding variable added."
          uid="173" />
      </method>
    - <method name="addProperty" returnType="void" comment="This method adds a new property into the output Properties
        object if it is not already there." uid="174">
        <param name="variable" paramType="java.lang.String" comment="The property variable name to add." uid="175" />
        <param name="value" paramType="java.lang.String" comment="The value to be set for the corresponding variable added."
          uid="176" />
      </method>
    </node>
  </etd>
```

ETD entity

The <etd> entity is the top-level entity in an **.xsc** file. It represents a complete ETD. Every **.xsc** file must contain exactly one of these entities. In this example, the file's ETD entity is has the name "SampleETD"; it is a single, complete ETD with no references to any other ETD.

```
(2)   <etd name="SampleETD" type="SampleETD" xscVersion="0.6" uid="0">
```

The **name** attribute defines the name of the ETD. The name of this ETD is **SampleETD**.

The **type** attribute is used to define the Connection type. The value of the **type** attribute is used in the selection list when creating an e*Way Connection within the e*Gate Schema Designer. The e*Way Connection is "SampleETD". The value of the **type** attribute must also match that of the Connection's ETD **.ctl** filename. The **.ctl** file is stored in the **gmeek\installETD** directory until it is transferred to the Registry.

Finally, the **uid** attribute uniquely identifies the <etd> entity within the **.xsc** file. All entities in the **.xsc** file are uniquely marked with the **uid** attribute. As a developer writing the file, ensure that the **uid** attributes are unique for all entities in the file.

A Connection XSC must contain, at a minimum, the **name**, **type**, and **uid** attributes for the <etd> entity.

Class node entities

The ETD structure begins with a <node> entity that represents the parent entity for the ETD.

```
(4)   <node name="SampleETD" type="class" uid="135">
```

The **name** attribute of the <node> entity must match the Java class name implementing the ETD. In this case, the name is **SampleETD**. There must be a corresponding class, named **sampleETD.class**, in the package named **sample** (as defined in the **packageName="sample"** attribute of the <etd> entity).

Because the implementation of the ETD is a customized Java class, the value for the **type** attribute of the <node> entity must be set to "class".

The attributes **minOccurs** and **maxOccurs** define the minimum and maximum occurrences allowed for this ETD. The example shows “1” and “1” respectively. The **minOccurs** and **maxOccurs** attributes define the lowest and highest possible number of occurrences of the given <node> below its parent. The values must both be non-negative integers (or “unbounded” for **maxOccurs**, meaning no upper limit), and the value for **minOccurs** must not exceed the value for **maxOccurs**. If unspecified, both attributes default to “1”.

The **optional** attribute, if unspecified, defaults to “false”. If set to “true”, then the node’s occurrence is optional. For example, if **minOccurs=“6”**, **maxOccurs=“8”**, and **optional=“true”**, then the node can occur 6, 7 or 8 times, or not at all.

The **uid** attribute uniquely identifies each entity within the **.xsc** file. As a developer writing the XSC, ensure that all **uid** attributes are unique for all entities in the **.xsc** file.

Class attribute node entities

To expose any Java class attributes of the class implementing the e*Way Connection, node entities under the ETD node must be defined in the **.xsc** file. For example, when the Automatic Connection was designed and implemented, it was given a class attribute **outFilename**, and corresponding access class methods **getFilename()** and **setFilename()** were also implemented; these methods allow the user to set or get the filename of the properties file within the Collaboration. The nodes named **directory** and **property** were also exposed to the user with the subsequent <node> entities.

For the second <node> entity, the value of the **name** attribute is set to “OutFilename”:

```
(5)   <node name="OutFilename" type="FIELD" optional="true" comment="This
      node contains the filename." uid="140" />
```

The <node> attributes **minOccurs** and **maxOccurs** define the minimum and maximum occurrences allowed for this ETD. The example shows “1” and “1” respectively.

The the **type** attribute is set to “FIELD”.

The **optional** attribute, if unspecified, defaults to “false”. If set to “true”, then the node’s occurrence is optional. In this example, the node can occur 1 time, or not at all.

The value of the **comment** attribute explains the purpose of this node.

Methods without parameters

To expose a class method of the Java class implementing the e*Way Connection, a <method> entity must be defined for the method:

```
(7)   <method name="saveOutPropertyFile" returnType="void" comment="This
      method saves the Property file." uid="160" />
```

The attributes are a subset of the attributes explained in **“Methods with parameters”**.

Methods with parameters

The Automatic Connection sample was designed and implemented to use a method called **copyAndAddProperty()**. This method takes the following parameters: a filename, a variable, and a value. The method copies the property or properties set by the user to a Java properties file (provided by the **filename** parameter), in a directory (provided by the **directory** parameter), and adds a property-value pair.

The **name** attribute defines the method name; the value **copyAndAddProperty** provides the name of the sample method as exposed to the end user.

The **returnType** attribute is set to the data type of the method's return value.

The value of the **comment** attribute explains the purpose of this method.

```
(8)   <method name="copyAndAddProperty" returnType="void" comment="This
      method saves the Properties from the input file, adds the new property
      specified then saves the updated properties to the output file specified
      by the filename and directory attributes." uid="170">
(9)   <param name="inputPropFilePath" paramType="java.lang.String"
      comment="The filename where to copy the Properties from." uid="171" />
(10)  <param name="variable" paramType="java.lang.String" comment="The
      property variable name to add." uid="172" />
(11)  <param name="value" paramType="java.lang.String" comment="The value to
      be set for the corresponding variable added." uid="173" />
(12)  </method>
```

10.6 Installing the Sample Files to e*Gate

After compiling the **.java** files and creating the **.xsc** file, you must commit all changes and additions to the e*Gate Registry. The instructions for committing the changes are contained in the **installETD.*** and **install.ctl** files provided.

10.6.1 Customizing the install.ctl File

The **install.ctl** file specifies the following information that must be modified when necessary:

- The **.ctl** file used by your ETD during run time.
- The **.jar** files containing the classes associated with your ETD.
- Any third-party **.jar** files used.
- The **.xsc** file associated with your ETD.
- The **.def** file (if used) associated with your e*Way Connection.

10.6.2 Testing Outside of e*Gate

This section is optional. It shows you how to validate the sample ETD outside of the e*Gate environment. The **SampleETDTester.java** file contains the source code for testing the ETD, and the **runTester.bat** file contains a script for running the tester.

SampleETDTester.java

The **SampleETDTester.class** file tests the **SampleETDExternalClass**.

```
package sample;

import java.io.*;

public class SampleETDTester {
  private SampleETDExternalClass extCall = null;

  public SampleETDTester() {
    extCall = new SampleETDExternalClass();
  }

  public void test(String inputPropFile,
                   String var,
                   String val,
                   String outputPropFile) {
    try {

      // Set the output file
      //
      extCall.setPropFilePath(outputPropFile);

      // Copy and add a property to the Properties object from
      // the input file into the new output Properties.
      //
      extCall.copyAndAddProperty(inputPropFile, var, val);

      // Save the new Properties copy with the new property into
      // the output Properties file specified using  setPropFilePath.
      //
      // This illustrates the typical interface to an external
      // system involving procedure that does open external, do work
      // with external, close external.
      //
      extCall.open();
      if (extCall.isOpen())
        extCall.save();
      extCall.close();
    }
    catch (Exception e) {
    }
  }

  public static void main(String args[]) {
    SampleETDTester tester = new SampleETDTester();

System.out.println("");
System.out.println("------------------------------------------------------------");
System.out.println("Input Properties file : " + args[0]);
System.out.println("Property variable     : " + args[1]);
System.out.println("Property value        : " + args[2]);
System.out.println("Output Properties file: " + args[3]);
System.out.println("------------------------------------------------------------");
System.out.println("");
tester.test(args[0], args[1], args[2], args[3]);
  }
}
```

Running the runTester File

Once the **.java** files have been compiled, run the **runTester.bat** file by opening a Command Prompt and entering the command **runTester**.

Do one of the following:

- On Windows, open a Command Prompt, change to the correct directory and then run the **runTester.bat** file by entering the following:

  ```
  .\runTester
  ```

- On UNIX, open a shell, change to the correct directory, and then run the **runTester.sh** file by entering the following:

  ```
  ./runTester.sh
  ```

The output should resemble the following:

```
#Thu May 16 14:08:33 PST 2002
newProperty=testValue
hello=world
```

10.6.3 Running the installETD Script

Do one of the following:

- On Windows, open a Command Prompt, change directories to **gmeek\installETD**, and enter the following command:

  ```
  .\installETD -e sampleETD -s MySchema -h localhost -g c:\eGate
  ```

  If you are installing to a different schema, host, or directory, make the appropriate substitutions. For syntax details, see **Windows: installETD.bat** on page 76.

- On UNIX, open a shell, change directories to **gmeek/installETD**, and enter the following command:

  ```
  ./installETD.sh -e sampleETD -s MySchema -h localhost -g /eGate
  ```

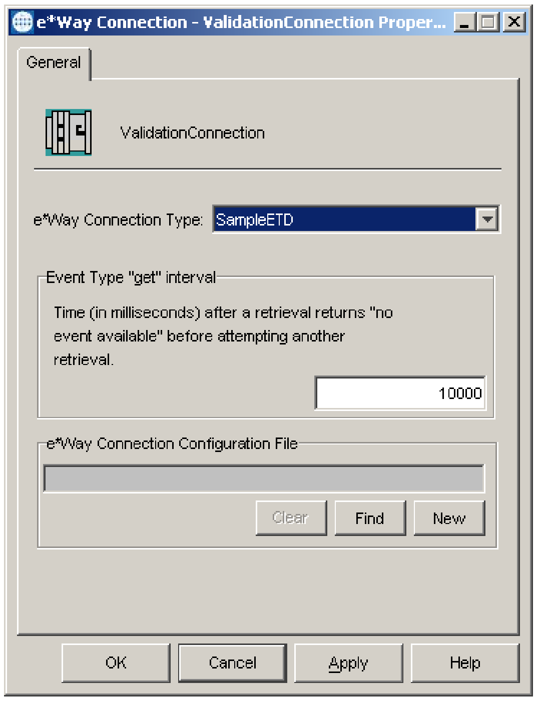
  If you are installing to a different schema, host, or directory, make the appropriate substitutions. For syntax details, see **UNIX: installETD.sh** on page 76.

10.6.4 Validating the Sample Files Within e*Gate

Start the Schema Designer and open the schema into which the **installETD** command was run.

Create a new e*Way Connection. If all of the files committed successfully, the e*Way Connection Editor contains the SampleETD entry as a possible e*Way Connection Type, as shown in Figure 48.

Figure 48 e*Way Connection Type SampleETD


e*Way Connection - ValidationConnection Proper...

General

ValidationConnection

e*Way Connection Type: SampleETD

Event Type "get" interval

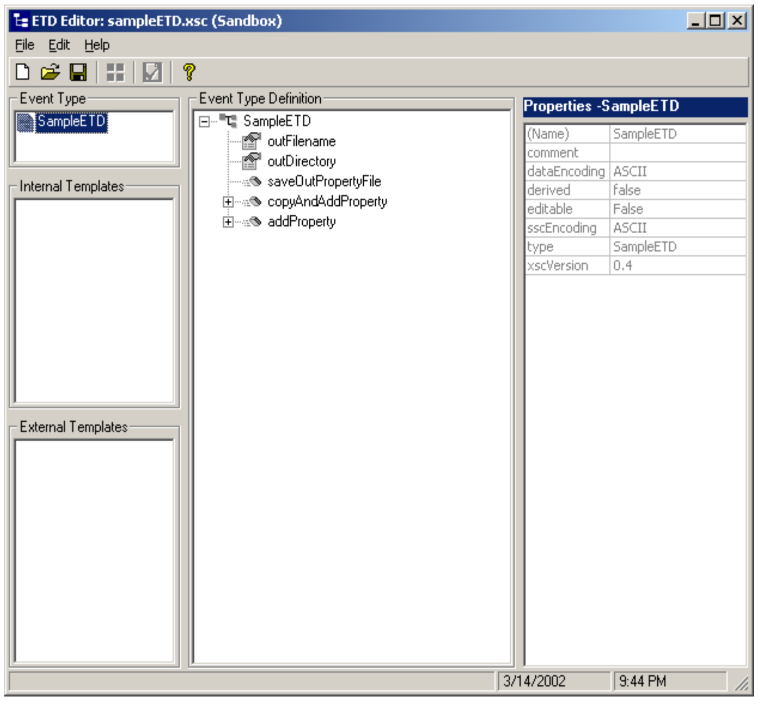
Time (in milliseconds) after a retrieval returns "no event available" before attempting another retrieval.

10000

e*Way Connection Configuration File

From the ETD Editor, open the **SampleETD.xsc**. If all the files committed successfully it should appear as shown in Figure 49.

Figure 49 SampleETD.xsc



(Name)	SampleETD
comment	
dataEncoding	ASCII
derived	false
editable	False
sscEncoding	ASCII
type	SampleETD
xscVersion	0.4

You are ready to import the sample schema.

10.7 Understanding the SampleETD Implemented in a Schema

This section:

- Tells you how to load the sample schema included in **MySchema.zip**.
- Shows you how the sample components appear to the end user.
- Shows you how the sample components behave in the e*Gate environment.

10.7.1 Importing the MySchema.zip Schema

To import the sample schema into e*Gate

1. Start the e*Gate Schema Designer GUI.
2. When the Schema Designer prompts you to log in, select the host that you specified during installation, and enter your password.

3 You are then prompted to select a schema. Click **New**. The New Schema dialog box opens. (Schemas can also be imported or opened from the e*Gate File menu by selecting **New Schema** or **Open Schema**.)

4 Enter a name for the new Schema, for example, **My_Sample_Schema**, or any name as desired.

5 To import the sample schema select **Create from Export**, and use **Find** to locate and select the **MySchema.zip** file on the e*Gate Integrator Installation CD-ROM, or from the location it was copied in earlier.

   MySchema.zip is the file supplied in the sample directory.

The e*Gate Schema Designer opens to the new schema. You are now ready to make any configuration changes that may be necessary for this sample schema to run on your specific system.

10.7.2 Sample Data INDATA

1 Extract the **INDATA.zip** file into a temporary directory.

2 Copy the extracted files into the **C:\INDATA** directory.

Note: *If you want or need to use a location other than **C:\INDATA** (for example, on UNIX), you must change the string "\INDATA" to the correct location in both your e*Gate schema's ewFeeder configuration and in the crAddProperty Collaboration.*

3 Rename the **SampleInput.~in** file to **SampleInput.fin**.

   The presence of a file with extension **.fin** triggers the file e*Way to read its contents.

   The **INDATA.zip** file contains two files: The **SampleInput.~in** file contains a new XML value pair to be added to the **input.properties** structure, shown as follows.

- **SampleInput.~in**:

```
<SampleInput>
  <PropsName>Color</PropsName>
  <PropsValue>Blue</PropsValue>
</SampleInput>
```

- **input.properties**:

```
#Thu May 16 14:08:33 PST 2002
newproperty=testvalue
hello=world
```

4 From a Command Prompt, enter the following command to have the e*Gate Control Broker (**stccb.exe**) start the imported sample schema:

```
stccb -rh localhost -un Administrator -up STC -ln localhost_cb -rs
<schema name>
```

   To use a Registry Host other than **localhost**, a username/password combination other than **Administrator**/**STC**, and/or a logical Control Broker name other than **localhost_cb**, make the appropriate substitutions.

Results

Successful execution creates a file named **output.properties** in c:\OutData specified by the e*Way Connection.

```
#Thu May 16 14:08:33 PST 2002
newproperty=testvalue
date=Thu May 16 14\:08\:33 PST 2002
hello=world
Color=Blue
```

Chapter 11

Developing an e*Way Connection With Connection Management

This *Generic Multi-Mode e*Way Extension Kit* also provides a complete set of files for setting up an e*Way Connection that shows you how to implement the Connection Management features.

Note: *The Connection Management feature is available only in e*Gate version 4.5.2 or later.*

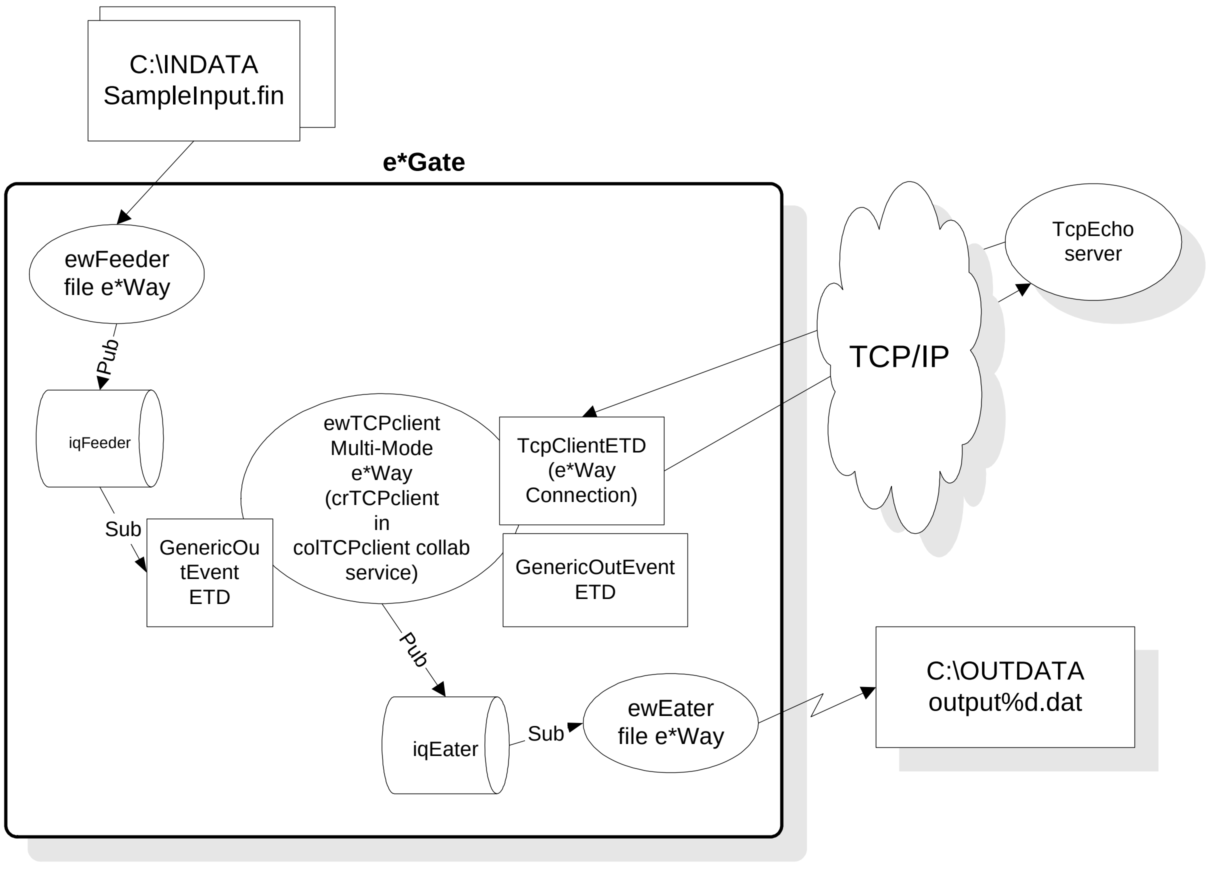
This chapter describes the necessary steps to add Connection Management into the ETD and connector classes of your e*Way Connection. It describes each of the files and takes you step by step through the process of modifying the files, compiling them, placing them in the correct locations, and validating them within the environment of an e*Gate schema.

The final section of this chapter takes you through the sample schema, showing you how the user-created ETD and e*Way Connection fit into e*Gate so you can match up your development efforts with the features seen by end users.

11.1 Overview

Figure 50 shows the architecture of the Connection Management sample.

Figure 50 Architecture of Connection Management Sample



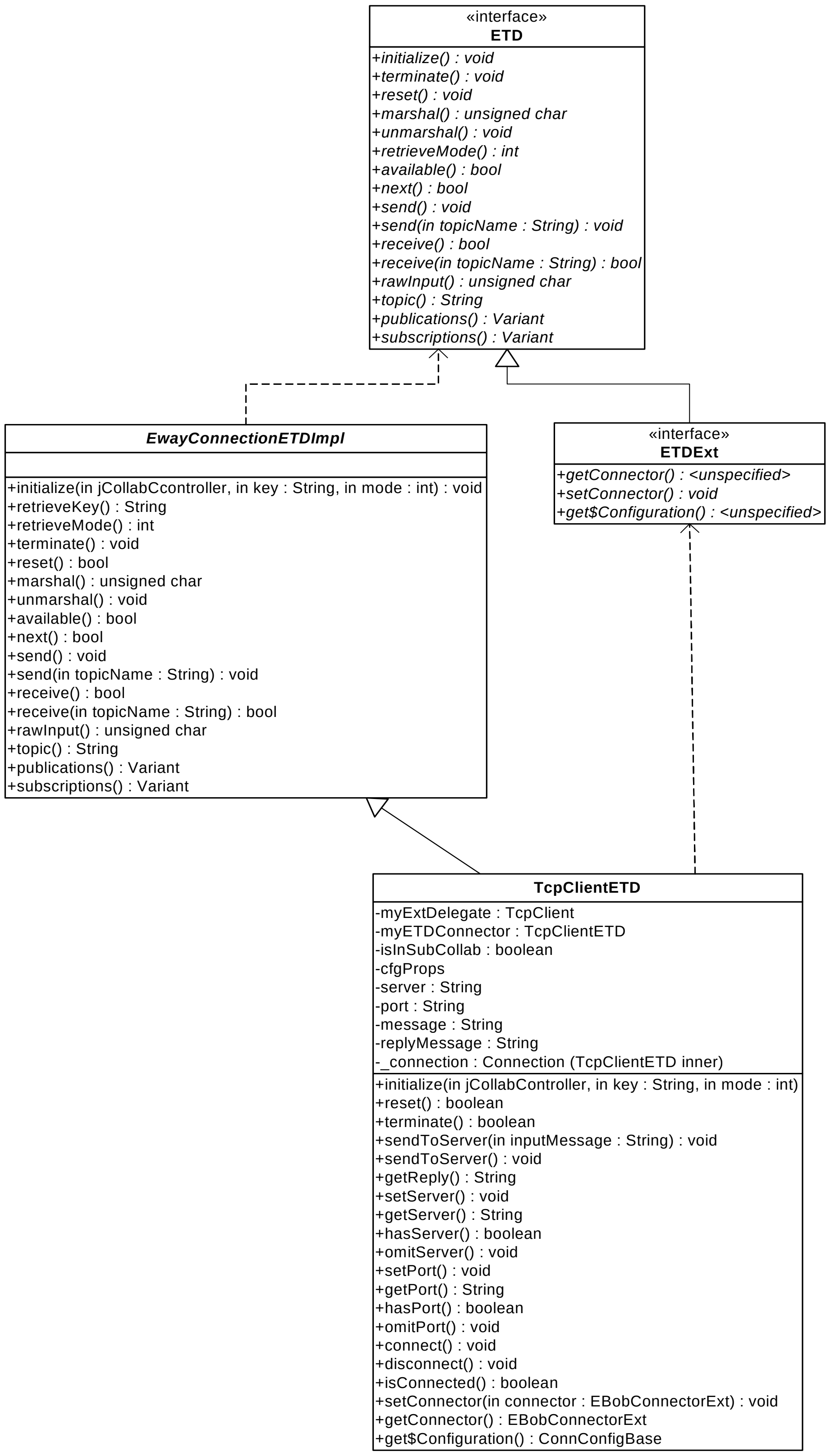
11.2 Classes and Interactions for the Connection Management Sample

The ETD class and the connector class for a Connection Management e*Way are slightly different from the ETD and connector classes for an Automatic Connection e*Way, and have different interactions. These differences are discussed in detail in the following sections.

11.2.1 ETD Class

For the Connection Management sample, the class diagram for the ETD class for the e*Way Connection is shown in **Figure 51 on page 187**.

Figure 51 Extending ETDImpl: The ETD Class for the Connection Management Sample

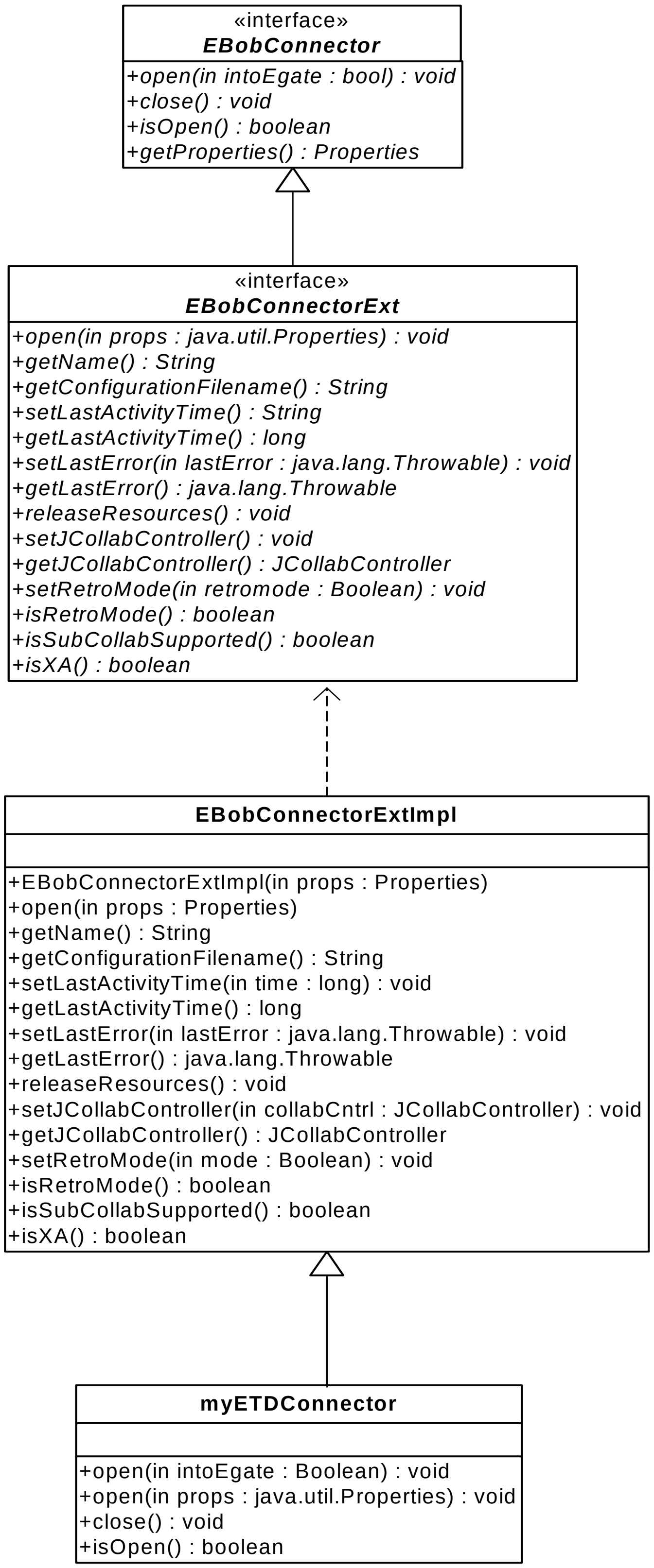


As shown in Figure 51, the *ETDExt* interface extends the *ETD* interface to allow setting and getting the Connector object associated with an ETD. The configuration associated with the ETD can also be obtained as an object. These are needed mainly to allow the user to set connection parameters on the ETD. The corresponding changes in the **.xsc** file are discussed in the next section.

11.2.2 Connector Class

For the Connection Management sample, the class diagram for the connector class for the e*Way Connection is shown in **Figure 52 on page 189**.

Figure 52 Connector Class for the Connection Management Sample


«interface»
EBobConnector
+open(in intoEgate : bool) : void
+close() : void
+isOpen() : boolean
+getProperties() : Properties
«interface»
EBobConnectorExt
+open(in props : java.util.Properties) : void
+getName() : String
+getConfigurationFilename() : String
+setLastActivityTime() : String
+getLastActivityTime() : long
+setLastError(in lastError : java.lang.Throwable) : void
+getLastError() : java.lang.Throwable
+releaseResources() : void
+setJCollabController() : void
+getJCollabController() : JCollabController
+setRetroMode(in retromode : Boolean) : void
+isRetroMode() : boolean
+isSubCollabSupported() : boolean
+isXA() : boolean
EBobConnectorExtImpl
+EBobConnectorExtImpl(in props : Properties)
+open(in props : Properties)
+getName() : String
+getConfigurationFilename() : String
+setLastActivityTime(in time : long) : void
+getLastActivityTime() : long
+setLastError(in lastError : java.lang.Throwable) : void
+getLastError() : java.lang.Throwable
+releaseResources() : void
+setJCollabController(in collabCntrl : JCollabController) : void
+getJCollabController() : JCollabController
+setRetroMode(in mode : Boolean) : void
+isRetroMode() : boolean
+isSubCollabSupported() : boolean
+isXA() : boolean
myETDConnector
+open(in intoEgate : Boolean) : void
+open(in props : java.util.Properties) : void
+close() : void
+isOpen() : boolean

As shown in Figure 52, the **EBobConnector** class subclass **EBobConnectorExt** adds the methods that interact with the Collaboration Controller and the Manager. The default implementation **EBobConnectorExtImpl** has been supplied; you can just extend this class to implement the connector class for your e*Way Connection.

11.3 Overview of the Connection Management Sample

The steps for implementing the Connection Management e*Way Connection begin with the same steps required by the Automatic Connection sample:

- Ensure your environment meets the prerequisites, and then load and unzip the sample source/install files in **gmeek.taz**. (You have already done this if you completed one of the previous samples.)
- Review or edit the **.java** files to understand the logic contained within the code.
- Edit the **compile.bat** script (or, on UNIX, **compile.sh**) to reflect your development environment.
- Run the **compile** script to compile the **.java** files and create **.jar** files.
- Edit the **.ctl** and **.def** files to reflect both your environment and the functionality required.
- Edit the **.xsc** file to understand the logic required for it to perform correctly.
- Start the Schema Designer and create a schema into which to commit the sample.
- Run the **installETD** script to make the sample files available to e*Gate.
- Return to the Schema Designer and validate the results of the preceding steps.
- Import the sample schema into e*Gate, start the Schema Manager, and validate the behavior of the TCP sample ETD and TCP sample e*Way Connection.

11.4 Installing the Sample

The installation package for the Connection Management sample comprises the following files:

- **gmeek.taz**
- **TcpEcho.zip**
- **INDATA.zip**

To install the files

1. Copy the **.taz**/**.zip** files to a temporary directory.

   If you have already completed the Automatic Connection sample, you only need to copy **TcpEcho.zip**, and you can skip step 2.

2 If you have not already done so, extract the **gmeek.taz** and **INDATA.zip** files to a convenient location. The path location of these files will be used in scripts later.

Changes to the directory locations of the unzipped files will require changes to the supplied scripts. For this reason, it is recommended that you complete the sample before making any changes to directory locations or file names.

11.5 Setting Up the Connection Management Sample Files

In this section, the details for editing the sample scripts and creating the **.java** files are broken down into steps:

"Editing/Viewing the .java Files" on page 192

"Customizing the Compile Script" on page 214

"Compiling the .java Files and Creating the .jar File" on page 215

"Editing/Viewing the .ctl Files" on page 215

"Editing/Viewing the .def Files" on page 216

"Editing/Viewing the .xsc File" on page 216

The files that apply specifically to the Connection Management sample are shown in Table 14.

Table 14 Connection Management Sample Files

Directory	Files
gmeek\TcpClientETD\	compile.bat compile.sh TcpClient.java TcpClientETD.java TcpClientETDConnector.java TcpClientETDDefs.java TcpClientException.java
gmeek\TcpClientETD\server\	runServer.bat TCPEchoServer.class TCPEchoServer.java
gmeek\installETD	installETD.bat *(on Windows systems)* installETD.sh *(on UNIX systems)*
gmeek\installETD\TcpClientETD	connectionpoint.ini install.ctl runTester.bat runTester.sh TcpClientETD.ctl TcpClientETD.def TcpClientETD.xsc

11.5.1 Editing/Viewing the .java Files

The files that have been provided require only minor edits to enable the sample schema contained in **TcpEcho.zip** to be implemented. This section provides the information necessary to view the existing code, edit, compile, and commit to the e*Gate Registry. The code samples are provided to further your understanding. The sample code is described in sections that describe its purpose.

The source code files for the Connection Management sample are as follows:

- TCPClientETD.java
- TCPClientETDConnector.java
- TCPClientDefs.java
- TCPClient.java
- TCPClientException.java
- TCPClientServer.java

The following sections discuss each of these files in detail.

11.5.2 TCPClient

TCPClientETD.java

1 Create your ETD class.

An e*Way Connection has an associated class that implements the **ETD** interface. This class is called your ETD class. The class **EwayConnectionETDImpl** is a sample provided to implement the ETD interface. **EwayConnectionETDImpl** is an abstract class that your ETD class must extend. **EwayConnectionETDImpl** provides the default implementation used for e*Way Connection for non-messageable ETDs. The **EwayConnectionETDImpl** class should not normally be modified. Any additional desired functionality should be included in your ETD class.

By extending **EwayConnectionETDImpl**, your ETD class inherits common behavior when interacting with e*Gate IQs and Oracle SeeBeyond JMS IQ Managers.

Note: *Since it is not intended for message parsing, **EwayConnectionETDImpl** contains empty implementations of the **marshal()** and **unmarshal()** methods.*

To modify TCPClientETD.java

The following steps show the changes that you will make to **TCPClientETD.java**. These steps only show the sections of code that you will modify. See **“TCPClientETD.java listing” on page 200** for the entire contents of **TCPClientETD.java**.

The following source code shows a sample ETD class skeleton (**TcpClientETD**).

```
(1)   package tcpsample;
(2)
```

```
(3)   import java.util.Properties;
(4)
(5)   import com.stc.common.collabService.EGate;
(6)   import com.stc.common.collabService.JCollabController;
(7)   import com.stc.common.collabService.JConnectionManager;
(8)   import com.stc.common.collabService.CollabConnException;
(9)   import com.stc.common.collabService.CollabDataException;
(10)  import com.stc.common.collabService.CollabResendException;
(11)  import com.stc.jcsre.ETDExt;
(12)  import com.stc.jcsre.EBobConnectorExtFactory;
(13)  import com.stc.jcsre.EBobConnectorExt;
(14)  import com.stc.jcsre.cfg.ConnConfigBase;
(15)
(16)
(17)  public class TcpClientETD extends TcpClientETDImpl implements ETDExt
```

The **import** statements (3) through (14) are required. They import the e*Gate core classes that are used in an ETD class utilizing a Connection Manager.

2 Create a delegate object in your ETD class.

```
(18)  /**
(19)     * Called by external (Collab Service) to initialize object.
(20)     * Reads configuration from config file, obtains a connector object
(21)     * used to establish connection with the external system through the
(22)     * connector factory, initializes myExtDelegate.
(23)     *
(24)     * @param     cntrCollab  The Java Collaboration Controller object.
(25)     * @param     key
(26)     * @param     mode
(27)     * @see       com.stc.jcsre.ETD
(28)     */
```

3 Override the **initialize()** method of your ETD class.

```
(29)  public void initialize(JCollabController cntrCollab, String key, int
      mode)
(30)      throws CollabConnException, CollabDataException
(31)    {
(32)      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
(33)        "Inside TcpClientETD.initialize()");
(34)
(35)      super.initialize(cntrCollab, key, mode);
```

4 Instantiate your connector class.

```
(36)  EBobConnectorExtFactory connFactory = new EBobConnectorExtFactory();
(37)      myETDConnector = (TcpClientETDConnector)
      connFactory.createConnectorExt(cntrCollab, key, mode);
(38)
(39)      if (myETDConnector != null)
(40)      {
(41)        EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
(42)          "Created TcpClientETDConnector via EBobConnectorExtFactory.");
(43)      }
(44)      else
(45)      {
(46)        EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
(47)          "Failed to create TcpClientETDConnector via
      EBobConnectorExtFactory.");
(48)        throw new CollabConnException("Unable to create
      TcpClientETDConnector");
(49)      }
(50)
```

5 Initialize your ETD class using the Properties from the configuration file.

```
(51)  // Extract properties from .cfg file
```

```
    //
    this.cfgProps = myETDConnector.getProperties();

    if (this.cfgProps != null)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
        "Got TcpClientETDConnector Properties.");
    }
    else
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
        "Failed to get TcpClientETDConnector Properties.");
     throw new CollabConnException("Failed to get TcpClientETDConnector
Properties.");
    }
```

6 Obtain the server name property from the configuration Properties object.

```
///////////////////////////////////////////////////////////////
// server
///////////////////////////////////////////////////////////////
    String propsServer =
cfgProps.getProperty(TcpClientETDDefs.ETD_DEF_PROP_NAME_SERVER);
    try {
      if (propsServer != null)
      {
        this.server = propsServer;
        EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
          "Default server set to " + propsServer);
      }
    }
    catch (Exception ex)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
        "Failed to set the default server; Exception : " +
ex.toString());
      throw new CollabConnException("Failed to set the default server;
Exception : " + ex.toString());
    }

```

7 Obtain the port number as a string. This must be converted from the configuration Properties object to an integer.

```
///////////////////////////////////////////////////////////////
// port
///////////////////////////////////////////////////////////////
    String propsPort =
cfgProps.getProperty(TcpClientETDDefs.ETD_DEF_PROP_NAME_PORT);
    try {
      if (propsPort != null)
      {
        this.port = propsPort;
        EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
          "Default port set to " + propsPort);
      }
    }
    catch (Exception ex)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
        "Failed to set the default port; Exception : " + ex.toString());
      throw new CollabConnException("Failed to set the default port;
Exception : " + ex.toString());
    }
```

8 Instantiate the delegate object. Set the external class in the connector object.

```
///////////////////////////////////////////////////////////////
// Do some Initialization
///////////////////////////////////////////////////////////////
    try {
      myExtDelegate = new TcpClient();

      myETDConnector.setExternalClass(myExtDelegate);

      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
                    "Set TcpClient class delegate.");
    }
    catch (Exception e)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
                    "Exception caught initializing external.");
      e.printStackTrace();
      throw new CollabConnException("Exception caught initializing 
external; Exception : " + e.toString());
    }

```

9 If this ETD is used in a Subcollaboration Rule, the same Connector object returned by the **createConnectorExt()** method is the same instance used by the parent Collaboration. The **isInSubcollab** Boolean flag is set here, to be used later by the **terminate()** method to determine whether or not to close the connection. If the connection is in a Subcollaboration Rule, it should not be closed at this time; it will be closed in the parent Collaboration.

```
if (myETDConnector.isSubCollabSupported())
    {
      isInSubCollab = cntrCollab.isSubCollaboration();
    }
    if (!isInSubCollab)
    {
      myETDConnector.setJCollabController(cntrCollab);
      if (myETDConnector.isRetroMode())
      {
        myETDConnector.open(false);
      }
```

10 Register the connector object with the Connection Manager. This allows the Connection Manager to call the Connector object's **open()**, **close()**, and **isOpen()** methods based on the Connection Management mode settings.

```
else  // register with Connection Manager
      {
        JConnectionManager conMgr = cntrCollab.getConnectionManager();
        conMgr.registerConnector(myETDConnector);
      }
    }

  }
```

11 Create the **get___()** and **set____()** methods to be used with attributes that are exposed in the ETD.

```
///////////////////////////////////////////////////////////////
//                                                           //
//    Getter/setter methods for attributes exposed in ETD    //
//                                                           //
///////////////////////////////////////////////////////////////

```

```
/**
* Call this in your Collaboration to set the server attribute.
*
* @param     server - server host
*/
  public void setServer(String server)
  {
    this.server = server;
  }

/**
* Call this in your Collaboration to get the server attribute.
*
* @return    server host
*/
  public String getServer()
  {
    return(this.server);
  }

/**
* Call this in your Collaboration to set the port attribute.
*
* @param     server port
*/
  public void setPort(String port)
  {
    this.port = port;
  }

/**
* Call this in your Collaboration to get the port attribute.
*
* @return    server port
*/
  public String getPort()
  {
    return(this.port);
  }

/**
* Call this in your Collaboration to set the message attribute.
*
* @param     message
*/
  public void setMessage(String msg)
  {
    this.message = msg;
  }

/**
* Call this in your Collaboration to get the message attribute.
*
* @return    message
*/
  public String getMessage()
  {
    return(this.message);
  }
```

12 Override the **reset()** method of your ETD class.

```
////////////////////////////////////////////////////////////////
//                                                            //
//    reset                                                   //
//                                                              //
//////////////////////////////////////////////////////////////////

  public boolean reset()
  {
    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
      "reset() has been called!");

    replyMessage = null;

    return true;
  }
```

13 Override the **terminate()** method of your ETD class. The connection should not be closed if it is in a Subcollaboration Rule. It will be closed in the parent Collaboration instead.

```
//////////////////////////////////////////////////////////////////
//                                                              //
//    terminate                                                 //
//                                                              //
//////////////////////////////////////////////////////////////////

/**
* Closes external connection if NOT in a Subcollaboration Rule.
* If in Subcollaboration, you must release resources used in
* the Subcollaboration but don't close your external connection.
*
*/
  public void terminate() throws CollabConnException
  {
    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
      "terminate() has been called!");

    try {
      if (!isInSubCollab)
        if (myETDConnector.isOpen())
          myETDConnector.close();
    }
    catch (com.stc.jcsre.EBobConnectionException ex)
    {
    }
  }
```

14 If you are supporting Manual mode, you must add the appropriate attributes and methods that allows the user to set the connection parameters and perform the following method calls from the ETD:

- **connect()**
- **disconnect()**
- **isConnected()**

```
//////////////////////////////////////////////////////////////////
//                                                              //
//    Methods to support Manual Connection Management mode      //
//                                                              //
//////////////////////////////////////////////////////////////////

  private Connection _connection = null;

  public Connection getConnection()
  {
    if (_connection == null)
      _connection = new Connection(myETDConnector.getProperties());
    return _connection;
  }

  public void setConnection(Connection conn)
  {
    _connection = conn;
  }

  public class Connection extends ConnConfigBase
  {
    public Connection()
    {
    }

    public Connection(Properties props)
    {
      setProperties(new Properties(props));
    }

    public java.lang.String getServer()
    {
      return getProperties().getProperty("Connection.Server");
    }

    public void setServer(java.lang.String val)
    {
      getProperties().setProperty("Connection.Server", val);
    }

    public boolean hasServer()
    {
      return (getServer() == null) ? false : true;
    }

    public void omitServer()
    {
      getProperties().remove("Connection.Server");
    }

    public java.lang.String getPort()
    {
      return getProperties().getProperty("Connection.Port");
    }

    public void setPort(java.lang.String val)
    {
      getProperties().setProperty("Connection.Port", val);
    }

    public boolean hasPort()
    {
      return (getPort() == null) ? false : true;
    }

    public void omitPort()
    {
      getProperties().remove("Connection.Port");
    }
  }

  public void connect() throws CollabConnException
  {
```

```
  if (myETDConnector != null)
  {
   boolean isManual = 
("Manual".equals(myETDConnector.getProperties().getProperty("connector.
Connection_Establishment_Mode"))) ? true : false;

   if (!isManual && !myETDConnector.isRetroMode())
    throw new CollabConnException ("Connector was not configured to 
be Manual. Can not call connect.");

   if (myETDConnector.isOpen())
    myETDConnector.close();
   if (_connection.getProperties() != null)
    myETDConnector.open(_connection.getProperties());
   else
    myETDConnector.open(true); // use default props from connector
  }
  else
   throw new CollabConnException("No TcpClientETDConnector 
instance.");
 }

 public void disconnect() throws CollabConnException
 {
  if (myETDConnector != null)
   myETDConnector.close();
  else
   throw new CollabConnException ("No TcpClientETDConnector 
instance.");
 }

 public boolean isConnected() throws CollabConnException
 {
  if (myETDConnector != null)
   return myETDConnector.isOpen();
  else
   throw new CollabConnException ("No TcpClientETDConnector 
instance.");
 }

 public void setConnector(EBobConnectorExt connector)
 {
  myETDConnector = (TcpClientETDConnector) connector;
 }


 public EBobConnectorExt getConnector()
 {
  return myETDConnector;
 }

 public ConnConfigBase get$Configuration()
 {
  return _connection;
 }
```

15 Define the methods that are exposed in the ETD.

```
//////////////////////////////////////////////////////////////
//                                                          //
//    Methods exposed in ETD                                //
//                                                          //
//////////////////////////////////////////////////////////////

/*
```

```
* Call this in your Collaboration to send a string to the server.
*/
  public void sendToServer(String inputMessage)
    throws CollabConnException, CollabDataException,
CollabResendException
  {
    try {
      byte[] reply = myExtDelegate.send(getMessage().getBytes());
      replyMessage = new String(reply);

      if (!myETDConnector.isRetroMode())
        myETDConnector.setLastActivityTime(System.currentTimeMillis());

    }
    catch (Exception e) {
      throw new CollabDataException(e.toString());
    }
  }
  public void sendToServer()
    throws CollabConnException, CollabDataException,
CollabResendException
  {
    sendToServer(this.message);
  }

/**
* Call this in your Collaboration to get the reply from the server.
*/
  public String getReply()
    throws CollabConnException, CollabDataException,
CollabResendException
  {
    try {
      if (replyMessage != null)
        return replyMessage;
    }
    catch (Exception e) {
      throw new CollabDataException(e.toString());
    }
    return null;
  }

}

```

TCPClientETD.java listing

The next few pages contain the source code for the ETD class (**TCPClientETD.java)** in its entirety.

```
package tcpsample;

import java.util.Properties;

import com.stc.common.collabService.EGate;
import com.stc.common.collabService.JCollabController;
import com.stc.common.collabService.JConnectionManager;
import com.stc.common.collabService.CollabConnException;
import com.stc.common.collabService.CollabDataException;
import com.stc.common.collabService.CollabResendException;
import com.stc.jcsre.ETDExt;
import com.stc.jcsre.EBobConnectorExtFactory;
import com.stc.jcsre.EBobConnectorExt;
import com.stc.jcsre.cfg.ConnConfigBase;

```

```

public class TcpClientETD extends TcpClientETDImpl implements ETDExt
{
  private String server  = null;
  private String port    = null;
  private String message = null;
  private String replyMessage = null;

  private Properties            cfgProps  = null;
  private TcpClient             myExtDelegate = null;
  private TcpClientETDConnector myETDConnector = null;
  private boolean               isInSubCollab = false;

  public TcpClientETD()
  {
  }

/**
* Called by external (Collab Service) to initialize object.
* Reads configuration from config file, obtains a connector object
* used to establish connection with the external system through the
* connector factory, initializes myExtDelegate.
*
* @param     cntrCollab  The Java Collaboration Controller object.
* @param     key
* @param     mode
* @see       com.stc.jcsre.ETD
*/
  public void initialize(JCollabController cntrCollab, String key, int mode)
    throws CollabConnException, CollabDataException
  {
    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
      "Inside TcpClientETD.initialize()");

    super.initialize(cntrCollab, key, mode);

    EBobConnectorExtFactory connFactory = new EBobConnectorExtFactory();
    myETDConnector = (TcpClientETDConnector) connFactory.createConnectorExt(cntrCollab, key, mode);

    if (myETDConnector != null)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
        "Created TcpClientETDConnector via EBobConnectorExtFactory.");
    }
    else
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
        "Failed to create TcpClientETDConnector via EBobConnectorExtFactory.");
      throw new CollabConnException("Unable to create TcpClientETDConnector");
    }

// Extract properties from .cfg file
//
    this.cfgProps = myETDConnector.getProperties();

    if (this.cfgProps != null)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
        "Got TcpClientETDConnector Properties.");
    }
    else
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
        "Failed to get TcpClientETDConnector Properties.");
     throw new CollabConnException("Failed to get TcpClientETDConnector Properties.");
    }


////////////////////////////////////////////////////////////////
// server
////////////////////////////////////////////////////////////////
    String propsServer = cfgProps.getProperty(TcpClientETDDefs.ETD_DEF_PROP_NAME_SERVER);
    try {
      if (propsServer != null)
      {
        this.server = propsServer;
        EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
          "Default server set to " + propsServer);
      }
    }
    catch (Exception ex)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
        "Failed to set the default server; Exception : " + ex.toString());
      throw new CollabConnException("Failed to set the default server; Exception : " + ex.toString());
    }

////////////////////////////////////////////////////////////////
// port
////////////////////////////////////////////////////////////////
    String propsPort = cfgProps.getProperty(TcpClientETDDefs.ETD_DEF_PROP_NAME_PORT);
    try {
      if (propsPort != null)
      {
        this.port = propsPort;
        EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_DEBUG,
          "Default port set to " + propsPort);
      }
    }
    catch (Exception ex)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
        "Failed to set the default port; Exception : " + ex.toString());
      throw new CollabConnException("Failed to set the default port; Exception : " + ex.toString());
    }

////////////////////////////////////////////////////////////////
// Do some Initialization
////////////////////////////////////////////////////////////////
    try {
      myExtDelegate = new TcpClient();

      myETDConnector.setExternalClass(myExtDelegate);

      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
                    "Set TcpClient class delegate.");
    }
    catch (Exception e)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
                    "Exception caught initializing external.");
      e.printStackTrace();
      throw new CollabConnException("Exception caught initializing 
external; Exception : " + e.toString());
    }

    if (myETDConnector.isSubCollabSupported())
    {
      isInSubCollab = cntrCollab.isSubCollaboration();
    }
    if (!isInSubCollab)
    {
      myETDConnector.setJCollabController(cntrCollab);
      if (myETDConnector.isRetroMode())
      {
        myETDConnector.open(false);
      }
      else  // register with Connection Manager
      {
        JConnectionManager conMgr = cntrCollab.getConnectionManager();
        conMgr.registerConnector(myETDConnector);
      }
    }

  }

///////////////////////////////////////////////////////////////
//                                                           //
//    Getter/setter methods for attributes exposed in ETD    //
//                                                           //
///////////////////////////////////////////////////////////////

/**
* Call this in your Collaboration to set the server attribute.
*
* @param     server - server host
*/
  public void setServer(String server)
  {
    this.server = server;
  }

/**
* Call this in your Collaboration to get the server attribute.
*
* @return    server host
*/
  public String getServer()
  {
    return(this.server);
  }

/**
* Call this in your Collaboration to set the port attribute.
*
* @param     server port
*/
  public void setPort(String port)
  {
    this.port = port;
  }
```

```

/**
* Call this in your Collaboration to get the port attribute.
*
* @return    server port
*/
  public String getPort()
  {
    return(this.port);
  }

/**
* Call this in your Collaboration to set the message attribute.
*
* @param     message
*/
  public void setMessage(String msg)
  {
    this.message = msg;
  }

/**
* Call this in your Collaboration to get the message attribute.
*
* @return    message
*/
  public String getMessage()
  {
    return(this.message);
  }

////////////////////////////////////////////////////////////////
//                                                            //
//    reset                                                   //
//                                                            //
////////////////////////////////////////////////////////////////

  public boolean reset()
  {
    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
      "reset() has been called!");

    replyMessage = null;

    return true;
  }

////////////////////////////////////////////////////////////////
//                                                            //
//    terminate                                               //
//                                                            //
////////////////////////////////////////////////////////////////

/**
* Closes external connection if NOT in a Subcollaboration Rule.
* If in Subcollaboration, you must release resources used in
* the Subcollaboration but don't close your external connection.
*
*/
  public void terminate() throws CollabConnException
  {
    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
      "terminate() has been called!");

    try {
      if (!isInSubCollab)
        if (myETDConnector.isOpen())
          myETDConnector.close();
    }
    catch (com.stc.jcsre.EBobConnectionException ex)
    {
    }
  }

////////////////////////////////////////////////////////////////
//                                                            //
//    Methods to support Manual Connection Management mode    //
//                                                            //
////////////////////////////////////////////////////////////////

  private Connection _connection = null;

  public Connection getConnection()
  {
    if (_connection == null)
      _connection = new Connection(myETDConnector.getProperties());
    return _connection;
  }

  public void setConnection(Connection conn)
  {
    _connection = conn;
  }

  public class Connection extends ConnConfigBase
  {
    public Connection()
    {
    }

    public Connection(Properties props)
    {
      setProperties(new Properties(props));
    }

    public java.lang.String getServer()
    {
      return getProperties().getProperty("Connection.Server");
    }

    public void setServer(java.lang.String val)
    {
      getProperties().setProperty("Connection.Server", val);
    }

    public boolean hasServer()
    {
      return (getServer() == null) ? false : true;
    }

    public void omitServer()
    {
      getProperties().remove("Connection.Server");
    }

    public java.lang.String getPort()
    {
      return getProperties().getProperty("Connection.Port");
    }

    public void setPort(java.lang.String val)
    {
      getProperties().setProperty("Connection.Port", val);
    }

    public boolean hasPort()
    {
      return (getPort() == null) ? false : true;
    }

    public void omitPort()
    {
      getProperties().remove("Connection.Port");
    }
  }

  public void connect() throws CollabConnException
  {
    if (myETDConnector != null)
    {
      boolean isManual = ("Manual".equals(myETDConnector.getProperties().getProperty("connector.Connection_Establishment_Mode"))) ? true : false;

      if (!isManual && !myETDConnector.isRetroMode())
        throw new CollabConnException ("Connector was not configured to be Manual. Cannot call connect.");

      if (myETDConnector.isOpen())
        myETDConnector.close();
      if (_connection.getProperties() != null)
        myETDConnector.open(_connection.getProperties());
      else
        myETDConnector.open(true); // use default props from connector
    }
    else
      throw new CollabConnException("No TcpClientETDConnector instance.");
  }

  public void disconnect() throws CollabConnException
  {
    if (myETDConnector != null)
      myETDConnector.close();
    else
      throw new CollabConnException ("No TcpClientETDConnector instance.");
  }

  public boolean isConnected() throws CollabConnException
  {
    if (myETDConnector != null)
      return myETDConnector.isOpen();
    else
      throw new CollabConnException ("No TcpClientETDConnector instance.");
  }

  public void setConnector(EBobConnectorExt connector)
  {
    myETDConnector = (TcpClientETDConnector) connector;
  }


  public EBobConnectorExt getConnector()
  {
    return myETDConnector;
  }

  public ConnConfigBase get$Configuration()
  {
    return _connection;
  }

///////////////////////////////////////////////////////////////
//                                                           //
//    Methods exposed in ETD                                 //
//                                                           //
///////////////////////////////////////////////////////////////

/*
* Call this in your Collaboration to send a string to the server.
*/
  public void sendToServer(String inputMessage)
    throws CollabConnException, CollabDataException,
CollabResendException
  {
    try {
      byte[] reply = myExtDelegate.send(getMessage().getBytes());
      replyMessage = new String(reply);

      if (!myETDConnector.isRetroMode())
        myETDConnector.setLastActivityTime(System.currentTimeMillis());

    }
    catch (Exception e) {
      throw new CollabDataException(e.toString());
    }
  }
  public void sendToServer()
    throws CollabConnException, CollabDataException,
CollabResendException
  {
    sendToServer(this.message);
  }

  /**
   * Call this in your Collaboration to get the reply from the server.
   */
  public String getReply()
    throws CollabConnException, CollabDataException,
CollabResendException
  {
    try {
      if (replyMessage != null)
        return replyMessage;
    }
    catch (Exception e) {
      throw new CollabDataException(e.toString());
    }
    return null;
  }

}

```

TCPClientETDConnector.java

Creates your ETD's connector class.

The implementation of your e*Way Connection's configuration and connection management functions must be provided in a class which implements **EBobConnector**. This class is referred to as the *connector class* of your ETD.

The following source code provided contains the information required for this connector class.

```
package tcpsample;

import java.util.Properties;

import com.stc.common.collabService.EGate;
import com.stc.jcsre.EBobConnectorExtFactory;
import com.stc.jcsre.EBobConnectorExt;
import com.stc.jcsre.EBobConnectorExtImpl;
import com.stc.jcsre.EBobConnectionException;

public class TcpClientETDConnector extends EBobConnectorExtImpl
{
  private   TcpClient  extClass;

  public TcpClientETDConnector(Properties props)
  {
    super(props);
  }

  /**
   * Opens the connector for accessing the external system.
   *
   * @param  intoEgate <code>true</code> if connector is to subscribe
   *                   to Events initially from an external and inbound
   *                   to e*Gate;
   *                   <code>false</code> if connector is to publish
   *                   Events outbound from e*Gate and to an external.
   *
   * @see    com.stc.jcsre.EbobConnector
   *
   * @throws com.stc.jcsre.EBobConnectionException when connection problems occur.
   */
   public void open(boolean intoEgate)
     throws com.stc.jcsre.EBobConnectionException
   {

     // Implement opening connection to external system
     //
     if (props == null)
     {
      lastError = new EBobConnectionException("Connector properties not set.");
       throw (EBobConnectionException)lastError;
     }

     String server = props.getProperty(TcpClientETDDefs.ETD_DEF_PROP_NAME_SERVER);
     if (server == null)
     {
      lastError = new EBobConnectionException("Server is not specified in the config file.");
```

```
        throw (EBobConnectionException)lastError;
      }

      String port = props.getProperty(TcpClientETDDefs.ETD_DEF_PROP_NAME_PORT);
      if (port == null)
      {
       lastError = new EBobConnectionException("Port is not specified in the config file.");
        throw (EBobConnectionException)lastError;
      }


      if (extClass != null)
        extClass.open(server, Integer.parseInt(port));
      else
      {
        lastError = new EBobConnectionException("TcpClient instance is null.");
        throw (EBobConnectionException)lastError;
      }
    }

    public void open(Properties connectProps)
      throws com.stc.jcsre.EBobConnectionException
    {
      if (connectProps == null)
      {
        lastError = new EBobConnectionException("Passed connector properties is null.");
        throw (EBobConnectionException)lastError;
      }

      String server = connectProps.getProperty(TcpClientETDDefs.ETD_DEF_PROP_NAME_SERVER);
      if (server == null)
      {
       lastError = new EBobConnectionException("Server is not specified in the config file.");
        throw (EBobConnectionException)lastError;
      }

      String port = connectProps.getProperty(TcpClientETDDefs.ETD_DEF_PROP_NAME_PORT);
      if (port == null)
      {
       lastError = new EBobConnectionException("Port is not specified in the config file.");
        throw (EBobConnectionException)lastError;
      }

      if (extClass != null)
        extClass.open(server, Integer.parseInt(port));
      else
      {
        lastError = new EBobConnectionException("No TcpClient class instance!");
        throw (EBobConnectionException)lastError;
      }
    }

    /**
     * Closes the connector to the external system and releases resources.
    *
    * @see      com.stc.jcsre.EbobConnector
    *
    * @throws    com.stc.jcsre.EBobConnectionException When connection problems occur.
    */
    public void close() throws com.stc.jcsre.EBobConnectionException
    {
      // Implement closing connection to external system
      //
      if (extClass != null)
        extClass.close();
      else
      {
        lastError = new EBobConnectionException("No TcpClient class instance!");
        throw (EBobConnectionException)lastError;
      }
    }

   /**
    * Verifies that the connector to the external system is still available.
    *
    * @return  <code>true</code> if the connector is still open and available;
    *          <code>false</code> otherwise.
    *
    * @see       com.stc.jcsre.EbobConnector
    *
    * @exception com.stc.jcsre.EBobConnectionException
    *              When connection problems occur.
    */
    public boolean isOpen() throws com.stc.jcsre.EBobConnectionException
    {
      // Implement returning if connection to external system is open
      //
      if (extClass != null)
        return extClass.isOpen();
      else
        return false;
    }

   /**
    * Set to the delegate external class instance by the
    */
    public void setExternalClass(TcpClient extClassInstance)
    {
      this.extClass = extClassInstance;
    }

   /**
    * Get to the delegate external class instance by the
    */
    public TcpClient getExternalClass()
    {
      return this.extClass;
    }
}


```

TCPClientDefs.java

Defines the string constants and property names to be pulled in from the default configuration-file template (**.def** file) in the source file **TCPClientETDDefs.java**.

```
package tcpsample;

public class TcpClientETDDefs {

  // Property names from e*Way Connection config file
  //
  public static final String ETD_DEF_PROP_NAME_SERVER =
    "TCPIP_Configuration.Server";

  public static final String ETD_DEF_PROP_NAME_PORT =
    "TCPIP_Configuration.Port";

}

```

TCPClient.java

Creates your class for interfacing with the external system.

```
package tcpsample;

import java.io.*;
import java.net.*;
import java.util.Properties;

public class TcpClient
{
  private InputStream  in = null;
  private OutputStream out = null;
  private Socket       socket = null;

  public TcpClient()
  {
  }

  public void open(String server, int serverPort)
  {
    try {
      socket = new Socket(server, serverPort);
      socket.setSoTimeout(10000);
      socket.setSoLinger(true, 10000);
      socket.setTcpNoDelay(true);

      System.err.println("Connected to server... sending echo string");

      in = socket.getInputStream();
      out = socket.getOutputStream();
    }
    catch (Exception e) {
    }
  }

  public void close()
  {
    try {
      socket.close();
    }
    catch (Exception e) {
      }
    }

    public boolean isOpen()
    {
      if (in == null || out == null || socket == null)
        return false;

      try {
        byte[] reply = send("ping".getBytes());

        if ("ping".equals(new String(reply)))
          return true;
      }
      catch (TcpClientException te) {
        return false;
      }
      return false;
    }

    public byte[] send(byte[] inBuffer)
      throws TcpClientException
    {
      byte[] outBuffer = new byte[inBuffer.length];

      try {
        System.err.println("Sending msg to server...");

        // send msg to server
        out.write(inBuffer);

        // receive back same msg from server
        int totalBytesRcvd = 0;
        int bytesRcvd;
        while (totalBytesRcvd < inBuffer.length)
        {
          if ((bytesRcvd = in.read(outBuffer, totalBytesRcvd,
                                  outBuffer.length - totalBytesRcvd)) ==
-1)
            throw new SocketException("Connection closed prematurely");

          totalBytesRcvd += bytesRcvd;
        }
      }
      catch (IOException ie) {
        throw new TcpClientException("Got IO exception.");
      }
      catch (Exception e) {
        throw new TcpClientException("Got exception sending message.");
      }

      return outBuffer;
    }

    public static void main(String[] args)
    {
      if (args.length < 1)
        System.out.println("Usage: java tcpsample.TcpClient <msg>");

      try {
        TcpClient myClient = new TcpClient();

        myClient.open("localhost", 9999);

```

```
   if (myClient.isOpen())
    System.out.println("--- connection is open ---");
   else
    System.out.println("--- connection is closed ---");

   System.out.println("Sending " + args[0]);

   byte[] fromServer = myClient.send(args[0].getBytes());

   System.out.println("Received: " + new String(fromServer));

   if (myClient.isOpen())
    System.out.println("--- connection is open ---");
   else
    System.out.println("--- connection is closed ---");

   myClient.close();
  }
  catch (Exception e) {
   System.out.println("Got exception running test.");
  }
 }
}


```

TCPClientException.java

Defines the exception class in source file **TCPClientException.java**.

```
package tcpsample;

import com.stc.eways.exception.STCDataException;

public class TcpClientException extends STCDataException
{
  public TcpClientException()
  {
    super();
  }

  public TcpClientException(String ex)
  {
    super(ex);
  }

  public TcpClientException(String ex, Exception e)
  {
    super(ex, e);
  }
}


```

11.5.3 TCPServer

TCPServer.java is a simple server sample which echoes back messages sent to it by the associated client. You can use it to test the TCPClient ETD. The script **RunServer.bat** is used to run the server on port 9999. The client and the server must be configured to run on the same port.

TCPServer.java

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

public class TCPEchoServer {

  private static final int BUFSIZE = 32;

  public static void main(String[] args) throws IOException {

    if (args.length != 1)
      throw new IllegalArgumentException("Parameter(s): <Port>");

    int servPort = Integer.parseInt(args[0]);

    ServerSocket servSock = new ServerSocket(servPort);

    int recvMsgSize;
    byte[] byteBuffer = new byte[BUFSIZE];

    for (;;) {
      Socket clntSock = servSock.accept();

      System.out.println("Handling client at " +
        clntSock.getInetAddress().getHostAddress() + " on port " +
          clntSock.getPort());

      InputStream in = clntSock.getInputStream();
      OutputStream out = clntSock.getOutputStream();

      while ((recvMsgSize = in.read(byteBuffer)) != -1)
        out.write(byteBuffer, 0, recvMsgSize);

      clntSock.close();
    }
  }
}

```

RunServer.bat

```
java -classpath . TCPEchoServer 9999
```

11.5.4 Customizing the Compile Script

The kit provides a **compile** script (**compile.bat** on Windows; **compile.sh** on UNIX) to set CLASSPATH information and to create a **.jar** file for the compiled **.java** files upon completion.

```
set GMEEK_EXTRACTDIR=C:\gmeekjars
set JAVA_PATH=C:\jdk1.3.1_02\bin
set
MYCLASSPATH="%GMEEK_EXTRACTDIR%\classes\stcjcs.jar;%GMEEK_EXTRACTDIR%\c
lasses\stcexception.jar;%GMEEK_EXTRACTDIR%\classes\stcutil.jar;"

%JAVA_PATH%\javac -classpath %MYCLASSPATH% -d . *.java

@REM
@REM jar up the classes
@REM

```

```

%JAVA_PATH%\jar cvf ..\installETD\TcpClientETD\TcpClientETD.jar
tcpsample\*.class

@REM
@REM jar up the source files to allow for debugger to use;
@REM please remove the following for a release version
@REM
copy *.java tcpsample
%JAVA_PATH%\jar uvf ..\installETD\TcpClientETD\TcpClientETD.jar
tcpsample\*.java
del tcpsample\*.java
```

As needed, make the following changes to reflect your environment:

1. set GMEEK_EXTRACTDIR=C:\gmeekjars

   Specify the correct location, if your e*Gate installation resides anywhere other than the root **\eGate** directory on your **C** drive.

2. `JAVA_PATH=c:\jdk1.3.1_02\bin`

   Specify the correct path location for your JDK.

3. When creating e*Way Connections from scratch, modify the directory locations in the **compile** script as needed.

11.5.5 Compiling the .java Files and Creating the .jar File

Open a Command Prompt, change to the correct directory and run the **compile** script:

On Windows

```
.\compile.bat
```

On UNIX

```
./compile.sh
```

The script is designed to set the CLASSPATH and create the **.jar** file. The **.jar** file is then saved to the **gmeek\installETD\TCPClientETD** directory.

11.5.6 Editing/Viewing the .ctl Files

The **TCPClientETD.ctl** contains the information required by the GUI to be able to successfully load the ETD. Any **.jar** files required by the ETD must be included here.

```
##-------------------------------------------------------------------
#
#  TcpClientETD.ctl  (The ETD CTL file)
#
#  This CTL file is used by the GUIs.  It specifies the JAR
#  files that are needed by your ETD classes for compilation
#  (in a Collaboration).  It also specifies the JAR files needed
#  at run time.
##-------------------------------------------------------------------

##-------------------------------------------------------------------
#   JAR files containing the classes associated with your ETD
#
##-------------------------------------------------------------------
TcpClientETD.jar,etd/TcpClientETD,FILETYPE_BINTEXT
```

```
stcexception.jar,classes,FILETYPE_BINTEXT
```

11.5.7 Editing/Viewing the .def Files

End users configure e*Ways using the e*Way Configuration Editor, a graphical user interface (GUI) that allows one to change configuration parameters quickly and easily. The e*Way Configuration Editor uses the default configuration-file template (**.def** file) to classify each parameter by its type and name, and can specify other information as well, such as the range of permissible options for a given parameter.

The Configuration Editor stores the values that you assign to those parameters within two configuration files. Each configuration file contains similar information, but the two are formatted differently:

- The **.cfg** file contains the parameter values in delimited records and is parsed by the e*Way at run time.
- The **.sc** file contains the parameter values and additional information needed by the GUI.

The e*Way Editor loads the **.sc** file—not the **.cfg** file—when you edit the configuration settings for an e*Way. Both configuration files are generated automatically by the e*Way Configuration Editor whenever the configuration settings are saved.

For more information on creating a custom **.def** file, see **Appendix A “Extending the .def File” on page 256**.

11.5.8 Editing/Viewing the .xsc File

The **.xsc** file provided with the sample contains the information provided by the **.java** files and required by the GUI.

TcpClientETD.xsc listing

```
<?xml version="1.0" encoding="UTF-8"?>
<etd name="TcpClientETD" type="TcpClientETD" xscVersion="0.6" uid="0" >
<javaProps package="tcpsample" codeAvailable="true" uid="1" />
<node name="Connection" type="CLASS" uid="cfg1">
    <node name="Server" optional="true" comment="Server" uid="cfg9" type="FIELD" />
<method name="hasServer" signature="hasServer()Z" returnType="boolean" comment="test
if element is present" uid="cfg10" />
   <method name="omitServer" signature="omitServer()V" returnType="void" comment="omit
element" uid="cfg11" />
    <node name="Port" optional="true" comment="Port" uid="cfg12" javaType="long"
type="FIELD" />
   <method name="hasPort" signature="hasPort()Z" returnType="boolean" comment="test if
element is present" uid="cfg13" />
    <method name="omitPort" signature="omitPort()V" returnType="void" comment="omit
element" uid="cfg14" />
  </node>
<node name="TcpClientETD" type="CLASS" uid="135">
<node name="Connection" type="FIELD" javaType="Connection" uid="cfg33" />
<node name="Server" type="FIELD" optional="true" comment="This node contains the
filename." uid="140" />
    <node name="Port" type="FIELD" optional="true" comment="This node contains the
directory." uid="143" />
    <node name="Message" type="FIELD" optional="true" comment="This node contains the
message sent if the no arg sendToServer method is used." uid="144" />
<method name="sendToServer" returnType="void" comment="This method sends the passed
string to the server." uid="170">
      <param name="message" paramType="java.lang.String" comment="The message to send
to the server." uid="171" />
    </method>
    <method name="sendToServer" returnType="void" comment="This method sends the
current value of getMessage to the server." uid="174"/>
```

```
<method name="getReply" returnType="java.lang.String" comment="This method returns the
reply message from the server." uid="160" />
<method name="connect" returnType="void" signature="connect()V" uid="31" />
    <method name="disconnect" returnType="void" signature="disconnect()V" uid="32" />
   <method name="isConnected" returnType="boolean" signature="isConnected()Z" uid="33"
/>
</node>
</etd>
```

11.6 Installing the Sample Files to e*Gate

After compiling the **.java** files and creating the **.xsc** file, you must commit all changes and additions to the e*Gate Registry. The instructions for committing the changes are contained in the **installETD.*** and **install.ctl** files provided.

11.6.1 Customizing the install.ctl File

The **install.ctl** file specifies the following information that must be modified when necessary:

- The **.ctl** file used by your ETD during run time.
- The **.jar** files containing the classes associated with your ETD.
- Any third-party **.jar** files used.
- The **.xsc** file associated with your ETD.
- The **.def** file (if used) associated with your e*Way Connection.

11.6.2 Testing Outside of e*Gate

This section is optional. It shows you how to validate the APIs used by TcpClientETD outside of the e*Gate environment. The **TcpClient.java** and **TCPEchoServer.java** files contain the source code for testing the APIs, and the **runServer** scripts are provided to run the server.

Running the runServer Script for TcpEchoServer

To compile the TCP Server

1. Do one of the following:
   - Invoke the javac:

     ```
     javac TCPEchoServer.java
     ```
   - Alternatively, you can use the **compile.bat** (or, on UNIX, **compile.sh**) file supplied in **gmeek\TcpClientETD\server\**.
2. After the **.java** files have been compiled, run the script by opening a Command Prompt and entering the command **runServer**.

Do one of the following:

- On Windows, open a Command Prompt, change to the correct directory and then run the **runServer.bat** file by entering the following:

  ```
  .\runServer
  ```

- On UNIX, open a shell, change to the correct directory, and then run the **runServer.sh** file by entering the following:

  ```
  ./runServer.sh
  ```

Note: *Initially, there should not be any output.*

TcpClient.java

The most important portion of the TcpClient.java file is listed as follows.

```
...
public static void main(String[] args)
{
  if (args.length < 1)
    System.out.println("Usage: java tcpsample.TcpClient <msg>");

  try {
    TcpClient myClient = new TcpClient();

    myClient.open("localhost", 9999);

    if (myClient.isOpen())
      System.out.println("--- connection is open ---");
    else
      System.out.println("--- connection is closed ---");

    System.out.println("Sending " + args[0]);

    byte[] fromServer = myClient.send(args[0].getBytes());

    System.out.println("Received: " + new String(fromServer));

    if (myClient.isOpen())
      System.out.println("--- connection is open ---");
    else
      System.out.println("--- connection is closed ---");

    myClient.close();
  }
  catch (Exception e) {
    System.out.println("Got exception running test.");
  }
}
...
```

Running the runTester Script for TcpEchoServer

Once the **.java** files have been compiled, run the **runTester.bat** file by opening a Command Prompt and entering the command **runTester**.

Do one of the following:

- On Windows, open a Command Prompt, change to the correct directory and then run the **runTester.bat** file by entering the following:

  ```
  .\runTester
  ```

- On UNIX, open a shell, change to the correct directory, and then run the **runTester.sh** file by entering the following:

  ```
  ./runTester.sh
  ```

The output for TcpClient should resemble the following:

```
(1)   Connected to server... sending echo string
(2)   Sending msg to server...
(3)   --- connection is open ---
(4)   Sending hello
(5)   Sending msg to server...
(6)   Received: hello
(7)   ...
```

At the same time, the output for TcpEchoServer should resemble the following:

```
(1)   Handling client at 127.0.0.1 on port 2588
(2)   Handling client at 127.0.0.1 on port 2589
(3)   ...
```

11.6.3 Running the installETD Script

If you do not already have a schema into which to commit your files, create one (such as **TcpEcho**). Then, do one of the following:

- On Windows, open a Command Prompt, change directories to **gmeek\installETD**, and enter the following command:

  ```
  .\installETD -e TCPClientETD -s TcpEcho -h localhost -g c:\eGate
  ```

  If you are installing to a different schema, host, or directory, make the appropriate substitutions. For syntax details, see **Windows: installETD.bat** on page 76.

- On UNIX, open a shell, change directories to **gmeek/installETD**, and enter the following command:

  ```
  ./installETD.sh -e TCPClientETD -s TcpEcho -h localhost -g /eGate
  ```

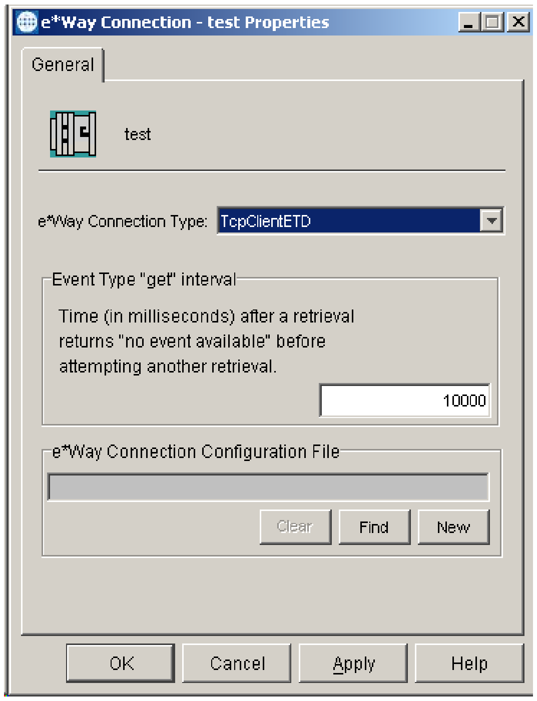
  If you are installing to a different schema, host, or directory, make the appropriate substitutions. For syntax details, see **UNIX: installETD.sh** on page 76.

11.6.4 Validating the Sample Files Within e*Gate

Start the Schema Designer and open the schema into which the installETD command was run.

Create a new e*Way Connection. If all of the files committed successfully, the e*Way Connection Editor contains theTcpClientETD entry as a possible e*Way Connection Type, as shown in Figure 53.

Figure 53 e*Way Connection Type TcpClientETD



test

e*Way Connection Type: TcpClientETD

Event Type "get" interval

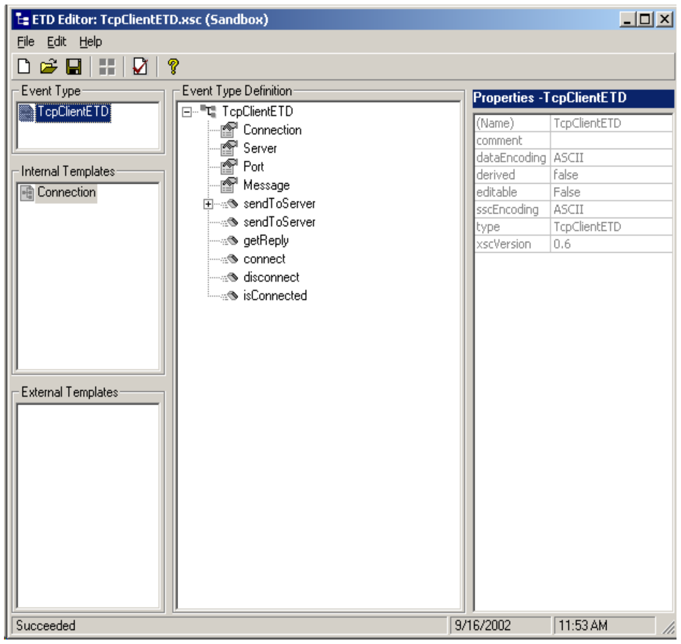
Time (in milliseconds) after a retrieval returns "no event available" before attempting another retrieval.

10000

e*Way Connection Configuration File

From the ETD Editor, open the **TcpClientETD.xsc**. If all the files committed successfully it should appear as shown in Figure 54.

Figure 54 TcpClientETD.xsc



```
TcpClientETD
    Connection
    Server
    Port
    Message
  + sendToServer
    sendToServer
    getReply
    connect
    disconnect
    isConnected
```

Properties -TcpClientETD

(Name)	TcpClientETD
comment	
dataEncoding	ASCII
derived	false
editable	False
sscEncoding	ASCII
type	TcpClientETD
xscVersion	0.6

Succeeded 9/16/2002 11:53 AM

11.7 Understanding the TcpClientETD Implemented in a Schema

This section:

- Tells you how to load the sample schema included in **TcpEcho.zip**.
- Shows you how the sample components appear to the end user.
- Shows you how the sample components behave in the e*Gate environment.

11.7.1 Importing the TcpEcho.zip Schema

To import the sample schema into e*Gate version 4.5.2 and later

1. Start the e*Gate Schema Designer GUI.
2. When the Schema Designer prompts you to log in, select the host that you specified during installation, and enter your password.
3. You are then prompted to select a schema. Click **New**. The New Schema dialog box opens. (Schemas can also be imported or opened from the e*Gate File menu by selecting **New Schema** or **Open Schema**.)
4. Enter a name for the new Schema, for example, **Tcp_Echo_Sample_Schema**, or any name as desired.
5. To import the sample schema select **Create from Export**, and use **Find** to locate and select the **TcpEcho.zip** file on the e*Gate Integrator Installation CD-ROM or from the location it was copied in earlier.

   TcpEcho.zip is the file supplied in the sample directory.

The e*Gate Schema Designer opens to the new schema. You are now ready to make any configuration changes that may be necessary for this sample schema to run on your specific system.

11.7.2 Sample Data INDATA

If you have not already extracted the **INDATA.zip** file into a temporary directory, follow steps 1 and 2. If you have already done this step, skip steps 1 and 2 and begin with step 3.

1. Extract the **INDATA.zip** file into a temporary directory.
2. Copy the extracted files into the **C:\INDATA** directory.

Note: *If you want or need to use a location other than* ***C:\INDATA*** *(for example, on UNIX), you must change the string "\INDATA" to the correct location in your e*Gate schema's ewFeeder configuration.*

3. Rename the **SampleInput.~in** file to **SampleInput.fin**.

The presence of a file with extension **.fin** triggers the file e*Way to read its contents while the e*Way is running.

The **INDATA.zip** file contains the **SampleInput.~in** file, as follows.

- **SampleInput.~in**:

```
<SampleInput>
  <PropsName>Color</PropsName>
  <PropsValue>Blue</PropsValue>
</SampleInput>
```

Note: *If the server is not already running, run the runServer script. For details on running the runServer script, see* **"Running the runServer Script for TcpEchoServer" on page 217**.

4. From a Command Prompt, enter the following command to have the e*Gate Control Broker (**stccb.exe**) start the imported sample schema:

   ```
   stccb -rh localhost -un Administrator -up STC -ln localhost_cb -rs
   <schema name>
   ```

   To use a Registry Host other than **localhost**, a username/password combination other than **Administrator**/**STC**, and/or a logical Control Broker name other than **localhost_cb**, make the appropriate substitutions.

Results

Once the schema is run, the output results should match the input. The output is specified by the ewEater configuration. Successful execution creates a file named **output0.dat**, or **output1.dat**, ... or a similar file name in C:\OUTDATA specified by the e*Way Connection.

Chapter 12

Developing a Transactional e*Way Connection

The *Generic Multi-Mode e*Way Extension Kit* provides a complete set of files for setting up a sample e*Way Connection that employs an XA-compliant Resource Manager and Transaction Manager —in other words, an "XA-enabled" e*Way Connection.

This chapter describes each of the files and takes you step by step through the process of modifying the files, compiling them, placing them in the correct locations, and validating them within the environment of an e*Gate schema.

The final section of this chapter takes you through the sample schema, showing you how the user-created ETD and e*Way Connection fit into e*Gate so you can match up your development efforts with the features seen by end users.

Note: *The **e*Gate User's Guide** provides an overview of XA terms and concepts and an architectural review of XA as it applies to e*Gate Collaborations. The overview in the following section expands on the material in the **e*Gate User's Guide**, taking an e*Way-centric view of transactional processing rather than a Collaboration-centric view.*

12.1 Overview

As discussed in **"e*Way Connections with Transaction Processing and XA" on page 68**, e*Gate supports one-phase as well as two-phase commit transactions through its implementation in user Collaborations of the Java Transaction API (JTA) specification. The relevant JTA components are:

- the Application Program (AP), represented in e*Gate by the Collaboration.
- the Application Server (AS), represented in e*Gate by the Multi-Mode e*Way.
- Resource Managers (RMs), represented in e*Gate by the ETDs that allow access to external resources.

12.1.1 Transactional Interfaces for e*Way Connection ETDs

One-phase transactional processing (XA-noncompliant)

To enable e*Way Connection ETDs to participate in one-phase transactions, the ETD class—in other words, the class that implements the **ETD** interface—must register itself with the Transaction Manager using the method **registerTransactionAdapter()**.

In addition, the ETD class must implement the **JTransactionAdapter** interface. The **JTransactionAdapter** interface contains the following methods:

- public void **commit()** throws CollabConnException;
- public void **rollback()** throws CollabConnException;
- public boolean **isPublisher()**;
- public java.lang.String **getID()**;

For information on the methods themselves, see the Javadoc files supplied on the e*Gate Integrator Installation CD-ROM.

Two-phase transactional processing (XA-compliant)

To allow your XA Resource Manager (managed by the underlying e*Way Connection ETD) to be recognized by the e*Gate Transaction components, the e*Way ETD must register itself with the Transaction Manager using the **register()** method, and the e*Way ETD must also implement the **JXAResourceAdapter** interface.

The **JXAResourceAdapter** interface contains the following methods:

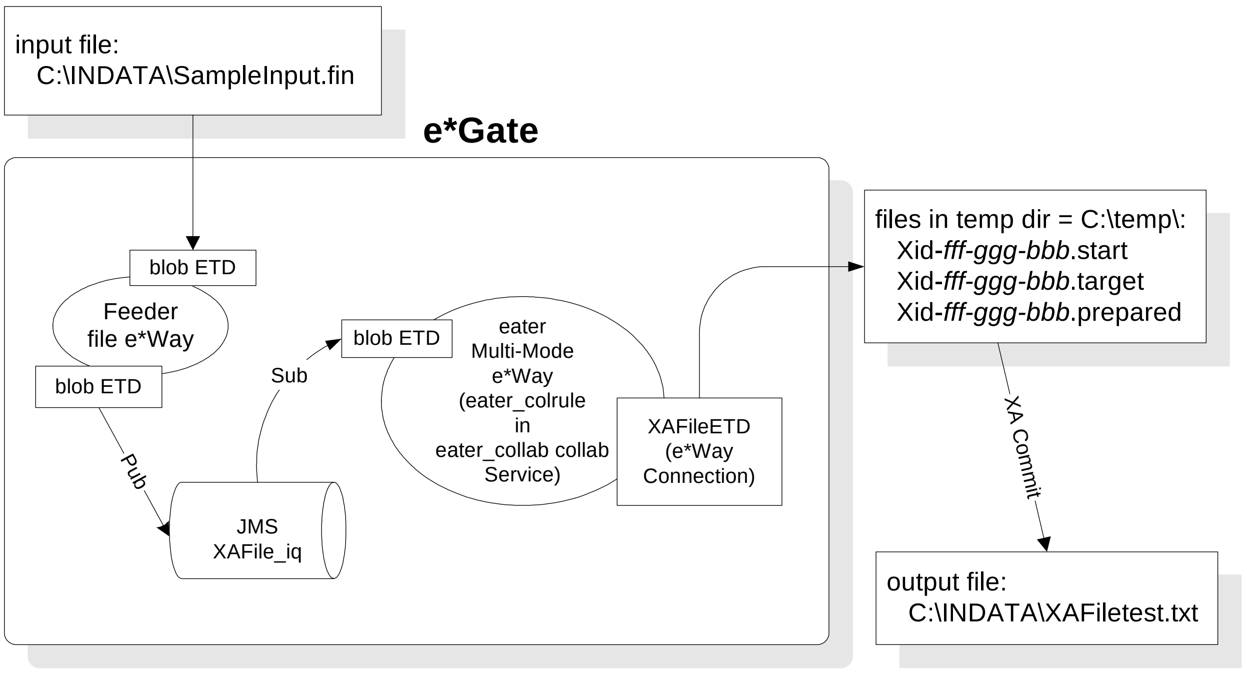
- public void **xaOpen(**java.lang.String *aKey***)** throws CollabConnException;
- public XAResource **getXAResource()** throws CollabConnException;
- public void **xaClose(**java.lang.String *aKey***)** throws CollabConnException;

For a discussion of the concepts and overall approach, refer to **“e*Way Connections with Transaction Processing and XA” on page 68**; for information on the methods themselves, see the Javadoc files supplied on the e*Gate Integrator Installation CD-ROM.

12.1.2 Architecture of the Sample Transactional e*Way Connection

The architecture of the Transactional sample is diagrammed in **Figure 55 on page 225**.

Figure 55 Architecture of e*Way Connection: Transactional Sample



In this sample schema, the XAFileETD e*Way Connection contains simple logic to process the file content of an input file feeding through the file e*Way to a BLOB ETD. The file's content is published to a JMS–enabled queue (an XA-compliant resource) and eventually arrives at the Collaboration business rule of the Multi-Mode e*Way. This is where the logic of an XA-compliant resource is used to create temporary journal files (**temp\id-yyy.***) corresponding to the different phases of the transaction.

When the XA-compliant resource is completely ready, a **commit()** command is issued and the designated output file is created or overwritten. However, if any external error conditions occur before the **commit()** logic is fully executed, the file is not created or overwritten. Instead, the file is re-created by the **rollback()** recovery logic of the XA-compliant resource the next time the e*Gate environment is properly set up again. There is no need to read the input file again.

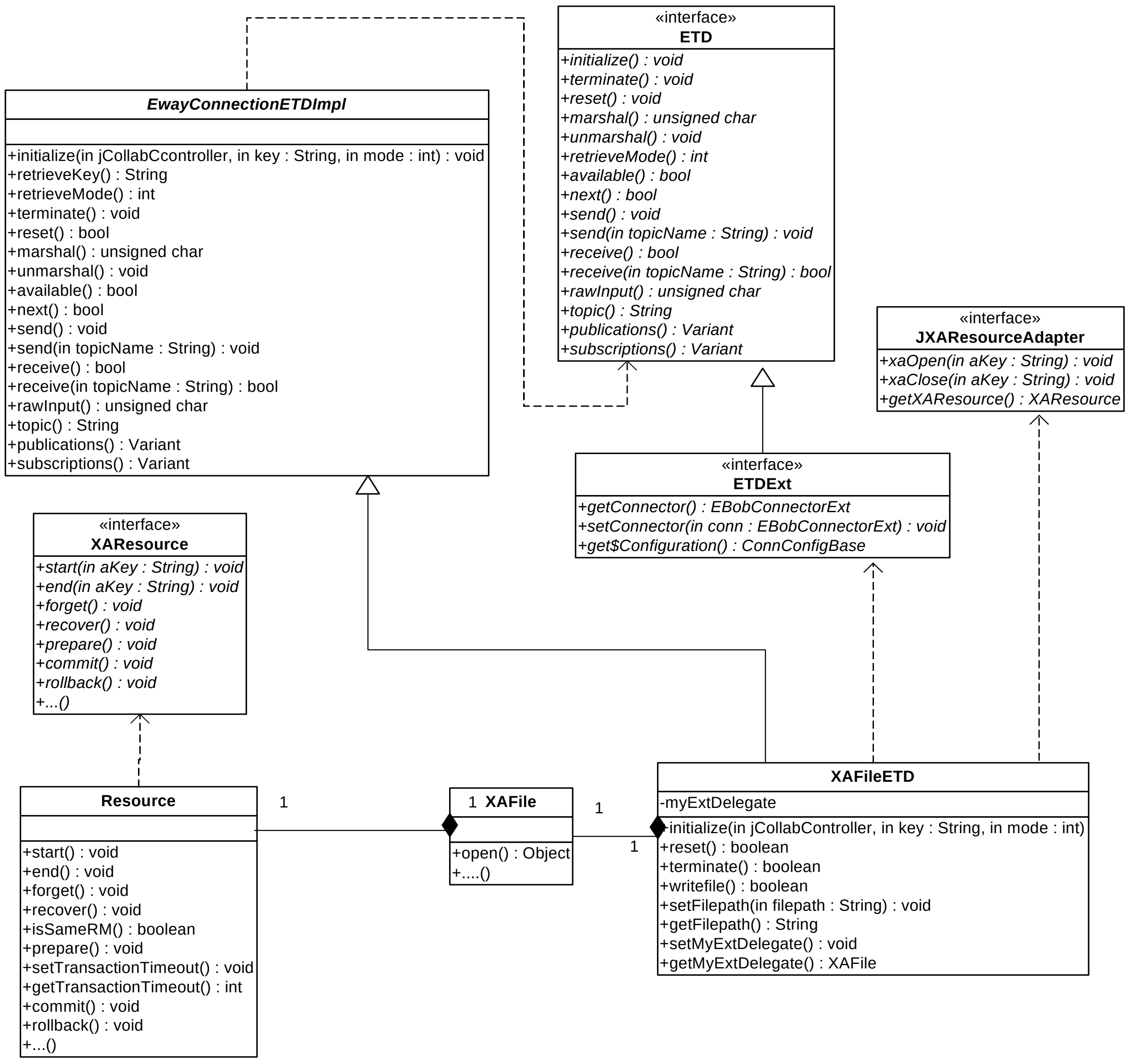
12.2 Classes and Interactions for the Transactional Sample

For the Transactional sample, the flow sequence is discussed in Figure 21 (“Sequence of Class Interactions in XA: Initialization Phase” on page 71) and Figure 22 (“Sequence of Class Interactions in XA: Translation Phase” on page 72). The connector class is the same as for the Automatic Connection sample (see “TCPClientETDConnector.java” on page 208). However, the ETD class is slightly different and has different interactions. These differences are discussed in detail in the following section.

12.2.1 ETD Class

For the Transactional sample, the class diagram for the ETD class for the e*Way Connection is shown in Figure 56.

Figure 56 ETD Class for the Sample XA-enabled e*Way Connection


EwayConnectionETDImpl
+initialize(in jCollabCcontroller, in key : String, in mode : int) : void
+retrieveKey() : String
+retrieveMode() : int
+terminate() : void
+reset() : bool
+marshal() : unsigned char
+unmarshal() : void
+available() : bool
+next() : bool
+send() : void
+send(in topicName : String) : void
+receive() : bool
+receive(in topicName : String) : bool
+rawInput() : unsigned char
+topic() : String
+publications() : Variant
+subscriptions() : Variant
«interface»
ETD
+initialize() : void
+terminate() : void
+reset() : void
+marshal() : unsigned char
+unmarshal() : void
+retrieveMode() : int
+available() : bool
+next() : bool
+send() : void
+send(in topicName : String) : void
+receive() : bool
+receive(in topicName : String) : bool
+rawInput() : unsigned char
+topic() : String
+publications() : Variant
+subscriptions() : Variant
«interface»
JXAResourceAdapter
+xaOpen(in aKey : String) : void
+xaClose(in aKey : String) : void
+getXAResource() : XAResource
«interface»
ETDExt
+getConnector() : EBobConnectorExt
+setConnector(in conn : EBobConnectorExt) : void
+get$Configuration() : ConnConfigBase
«interface»
XAResource
+start(in aKey : String) : void
+end(in aKey : String) : void
+forget() : void
+recover() : void
+prepare() : void
+commit() : void
+rollback() : void
+...()
Resource
+start() : void
+end() : void
+forget() : void
+recover() : void
+isSameRM() : boolean
+prepare() : void
+setTransactionTimeout() : void
+getTransactionTimeout() : int
+commit() : void
+rollback() : void
+...()
1
1 XAFile
+open() : Object
+....()
1
1
XAFileETD
-myExtDelegate
+initialize(in jCollabController, in key : String, in mode : int)
+reset() : boolean
+terminate() : boolean
+writefile() : boolean
+setFilepath(in filepath : String) : void
+getFilepath() : String
+setMyExtDelegate() : void
+getMyExtDelegate() : XAFile

The relationships between class **XAFileETD** and interfaces *EwayConnectionETDImpl*, *ETD*, and *ETDExt* are similar to that for TcpClientETD shown in **Figure 51 on page 187**. Figure 56 shows the delegate object (XAFile) along with the Resource object it contains. The Resource object implements the *XAResource* interface, which defines the contract between a Resource Manager and the e*Gate Transaction Manager, and the **XAFileETD** class implements the *JXAResourceAdapter* interface. To keep the kit as generic as possible, the sample does not involve a sophisticated Resource Manager such as a DBMS, so the Resource object takes on the role of XA Resource Manager.

12.3 Overview of the Transactional Sample

Implementing and validating the Transactional sample requires the following steps:

- Ensure your environment meets the prerequisites, and then load and unzip the sample source/install files in **gmeek.taz**. (You have already done this if you completed one of the previous samples.)
- Review or edit the **.java** files to understand the logic contained within the code.
- Edit the **compile.bat** script (or, on UNIX, **compile.sh**) to reflect your development environment.
- Run the **compile** script to compile the **.java** files and create a **.jar** file.
- Edit the **.ctl** and **.def** files to reflect both your environment and the functionality required.
- Review or edit the **.xsc** file to understand the logic required for it to perform correctly.
- Start the Schema Designer and create a schema into which to commit the sample.
- Run the **installETD** script to make the sample files available to e*Gate.
- Return to the Schema Designer and validate the results of the preceding steps.
- Import the sample schema into e*Gate, start the Schema Manager, and validate the behavior of the sample ETD and sample e*Way Connection.

12.4 Installing the Sample

The installation package for the Transactional sample comprises the following files:

- **XAFile.zip** (e*Gate schema for the Transactional sample)
- **gmeek.taz**
- **INDATA.zip**

To install the files

1. Copy the files to a temporary directory.

   If you have already completed one of the previous samples, you only need to copy **XAFile.zip**.
2. Extract the **gmeek.taz** file (and, if necessary, the **INDATA.zip** file) to a convenient location. The path location of these files will be used in scripts later.

Note: *Changes to the directory locations of the unzipped files will require changes to the supplied scripts. For this reason, it is recommended that you complete the sample before making any changes to directory locations or file names.*

12.5 Setting Up the Transactional Sample Files

In this section, the details for editing the sample scripts and creating the **.java** files are broken down into steps:

- **"Editing/Viewing the .java Files" on page 228**
- **"Customizing the Compile Script" on page 232**
- **"Compiling the .java Files and Creating the .jar File" on page 233**
- **"Editing/Viewing the XAFileETD.ctl File" on page 233**
- **"Editing/Viewing the XAFile.def File" on page 234**
- **"Editing/Viewing the XAFileETD.xsc File" on page 234**
- **"Customizing the install.ctl File" on page 235**
- **"Creating a Schema for the New ETD" on page 235**

The files that apply specifically to the Transactional sample are shown Table 15. Unless otherwise specified, when this chapter refers to <*filename*>**.java**, **.ctl**, **.def**, and so forth, the corresponding directory names are implied.

Table 15 Transactional Sample Files

Directory	Files
gmeek\XAFileETD\	compile.bat compile.sh EwayConnectionETDImpl.java Resource.java XAFile.java XAFileETD.java XAFileETDConnector.java XAFileETDDefs.java XidValue.java
gmeek\installETD\	installETD.bat *(on Windows systems)* installETD.sh *(on UNIX systems)*
gmeek\installETD\XAFileETD\	connectionpoint.ini install.ctl runTester.bat runTester.sh XAFileETD.ctl XAFileETD.def XAFileETD.xsc

12.5.1 Editing/Viewing the .java Files

The files that have been provided require only minor edits to allow you to deploy the sample schema contained in **XAFile.zip**. This section provides the information necessary to view the code, edit and compile it, and commit files to the e*Gate Registry.

The code samples are provided to further your understanding. The sample code is described in sections that describe its purpose.

The source code files for the Transactional sample are as follows:

- XAFileETD.java
- XAFileETDDefs.java
- Resource.java
- XidValue.java
- XAFile.java
- XAFileETDConnector.java
- EwayConnectionETDImpl.java

The following sections discuss each of these files in detail.

XAFileETD.java

XAFileETD is the main e*Way Connection ETD class which may be instantiated in a Collaboration as an e*Way Connection ETD. The class relationships discussed in **"TCPClient" on page 192** apply here as well.

The following source code shows a portion of the sample ETD class skeleton for **XAFileETD**:

```
(1)   package xasample;
(2)   import java.util.Properties;
(3)   import com.stc.common.collabService.EGate;
(4)   import com.stc.common.collabService.JCollabController;
(5)   import com.stc.common.collabService.JConnectionManager;
(6)   import com.stc.common.collabService.CollabConnException;
(7)   import com.stc.common.collabService.CollabDataException;
(8)   import com.stc.common.collabService.CollabResendException;
(9)   import com.stc.jcsre.ETDExt;
(10)  import com.stc.jcsre.EBobConnectorExtFactory;
(11)  import com.stc.jcsre.EBobConnectorExt;
(12)  import com.stc.jcsre.cfg.ConnConfigBase;

(13)  import java.io.*;
(14)  import java.util.Date;
(15)  import java.lang.Thread;
(16)  import javax.transaction.xa.*;
(17)  import com.stc.common.collabService.JXAResourceAdapter;

(18)  public class XAFileETD extends EwayConnectionETDImpl implements ETDExt,
      JXAResourceAdapter
(19)
(20)  import xasample.*;
```

The **import** statements (2) through (17) are required.

In addition to the e*Gate core classes that are imported as discussed in**"TCPClient" on page 192**, there are a few more new classes to import:

- **javax.transaction.xa.***—needed for XA-related classes such as **XAResource** and **Xid**.

- **com.stc.common.collabService.***JXAResourceAdapter*—as mentioned earlier, this XAFile ETD must implement the *JXAResourceAdapter* interface so that with proper registration, the Transaction Manager can call back various methods of the external XA resource, such as **commit()**, **rollback()**, **start()**, **prepare()**, and so forth, using the **getXAResource()** method to obtain the reference to the Resource object for this XAFileETD sample.
- **java.lang.Thread**—needed for an optional delay parameter. (When you validate the sample, this delay allows you to manually introduce an external error so you can test XA recovery.)
- **java.util.Date**—needed for prepending a timestamp to the output file.

For typical actions, strategies, and rationales regarding the following steps, see **"Creating .java Files" on page 56**:

- Create a Delegate object in your ETD class: See step 2 on page 58, and remember that your ETD class in this sample is **XAFile**.
- Override the **initialize()** method of your ETD class: See step 3 on page 58.
- Instantiate your connector class: See step 4 on page 58.
- Override your ETD class's **reset()** method: See step 5 on page 59.
- Override your ETD class's **terminate()** method: See step 6 on page 59.
- Create your ETD's connector class: See step 7 on page 59.

Additionally, for this XA sample, you use the **registerConnector()** method to register the connector object myETDConnector with the Connection Manager (obtained through the **getConnectionManager()** method of the **JCollabController** input parameter of the **initialize()** method to override) and also to register the ETD class itself (which is an implementation of the *JXAResourceAdapter* interface) with the Transaction Manager using a simple **register()** method of the **JCollabController** input parameter.

The **XAFileETD** class only needs to implement the **getXAResource()** method of the *JXAResourceAdapter* interface. The method **getXAResource()** is invoked to obtain a reference to the Resource object associated with the Resource Manager (**XAResource**) actually held by the delegate object XAFile; the implementation of the methods **xaOpen()** and **xaClose()** are placeholders for this ETD.

XAFileETDDefs.java

In source code file **XAFileETDDefs.java**, certain constants are used by the other classes in the sample package. These consist of string constants and property names; the latter are associated with the default configuration-file template (**.def** file).

Resource.java

Create your class for interfacing with the external system—in this case, an XA-compliant Resource Manager. You may opt to include the definition of **Resource** as an inner class for the **XAFile** class for this sample.

Resource is a sample class containing the methods that interact with an external entity—in this case, the native file system, but normally the external entity is a complete XA-compliant Resource Manager (perhaps provided as a third-party component) that may or may not support Connection Pooling. Notice that the **XAFile** class contains an instance of this class serving as a delegate that performs the method calls made by the e*Gate Collaboration user through an XAFileETD object.

For more details on each XAResource interface method, see the Javadocs for **javax.transaction.xa.XAResource** in **"e*Way Classes and Methods" on page 254**.

- The **start()** method creates a file with the Transaction Manager–generated XID as the basis for the file name and a **.start** file extension in the temporary directory. At first, the file does not contain any detail.
- The **end()** method performs a flush on the buffer associated with the output target file (which is at this point a temporary file in the temporary directory) specified in the **.def** file.
- The **prepare()** method takes the temporary file in the temporary directory containing the flushed data and renames to a **.prepared** file, signifying that the **prepare()** operation is finished.
- The **commit()** method renames the **.prepared** file to the name currently contained in **.target** file in the temporary directory (after the output target file is first removed, if already exists) and performs cleanup to remove the remaining **.target** file.
- The **recover()** method obtains the list of transaction branches—in other words, a list of XIDs—that are currently in prepared states, by extracting the XIDs from the file names of all the **.prepared** files left in the temporary directory.
- The **rollback()** method cleans up all the temporary files in the temporary directory. These consist of **.target**, **.prepared**, and **.start** files, if any.
- The **setTransactionTimeout()** method is not implemented, since there is no underlying Resource Manager for this sample. As supplied, it always returns false, dooming any attempt to use this method. If a Resource Manager is available, provide the appropriate logic—for example, you could invoke the corresponding transaction timeout setup method for the resource.

  The **getTransactionTimeout()** and **forget()** methods are also not fully implemented, for similar reasons.
- The **isSameRM()** method normally compares the current Resource Manager with another Resource Manager. However, there is no other Resource Manager involved in this case, and so this method is not implemented fully.

XidValue.java

XidValue is an implementation of the **Xid** interface, which is a Java mapping of the *X/Open transaction identifier* (XID) structure. The **Xid** interface is used by the e*Gate Transaction Manager and the Resource Managers.

The three parts of an XID must meet he following criteria:

- The *format identifier* is unique to the implementation;

- The *global identifier* uniquely identifies the transaction across a collection of server.
- The *branch qualifier* (also called the *branch identifier*) identifies one of several branches inside the transaction.

In the XA sample provided, the Xid does not make use of the generation logic that the XidValue class implements. Instead, it is used to obtain the field values originated by the e*Gate Transaction Manager.

XAFile.java

XAFile is the same delegate object discussed in chapters 10 and 11. For additional information on the rationale and usage of the delegate object, refer to **"Creating .java Files" on page 56** and to **"Using a Delegate Class" on page 243**.

XAFileETDConnector.java

The **XAFileETDConnector** class is the counterpart of the TcpclientETD sample TcpClientETDConnector. For more information, refer **"Editing/Viewing the .java Files" on page 159**.

Notice that an additional helper method is included: **isXA()**. This returns a Boolean **true** to indicate that the ETD is XA-enabled. This is useful for implementing logic to prohibit XA operations from running at the Subcollaboration Rule level. See **"Caveats" on page 55**.

EwayConnectionETDImpl.java

EwayConnectionETDImpl implements the ETD class so as to provide all the default implementations used for e*Way Connection (non-messageable) ETDs. For more information, refer to **"Editing/Viewing the .java Files" on page 159**.

Note: *Since it is not intended for message parsing, **EwayConnectionETDImpl** contains empty implementations of the **marshal()** and **unmarshal()** methods. For more information on parsing messageable ETDs, see* **"Handling Messageable ETDs" on page 243**.

12.5.2 Customizing the Compile Script

The kit provides a **compile** script (**compile.bat** on Windows; **compile.sh** on UNIX) to set CLASSPATH information and to create a **.jar** file for the compiled **.java** files upon completion.

```
set GMEEK_EXTRACTDIR=C:\gmeekjars
set JAVA_PATH=C:\jdk1.3.1_02\bin
set 
MYCLASSPATH="%GMEEK_EXTRACTDIR%\classes\stcjcs.jar;%GMEEK_EXTRACTDIR%\T
hirdParty\sun\jta.jar;%GMEEK_EXTRACTDIR%\classes\stcexception.jar;"

%JAVA_PATH%\javac -classpath %MYCLASSPATH% -d . *.java

@REM
@REM jar up the classes
```

```
@REM


%JAVA_PATH%\jar cvf ..\installETD\XAFileETD\XAFileETD.jar xasample\*.class

@REM
@REM jar up the source files to allow for debugger to use;
@REM please remove the following for a release version
@REM
copy ..\XAFileETD\*.java xasample
%JAVA_PATH%\jar uvf ..\installETD\XAFileETD\XAFileETD.jar xasample\*.java
del xasample\*.java
```

Note: *The **jta.jar** file contains the classes for **javax.transaction.xa** and is installed by default for e*Gate version 4.5.2 and later.*

As needed, make the following changes to reflect your environment:

1. `set GMEEK_EXTRACTDIR=C:\gmeekjars`

   If your e*Gate installation resides anywhere other than the root **\eGate** directory on your **C** drive, specify the correct location.

2. `JAVA_PATH=c:\jdk1.3.1_02\bin`

   Specify the correct path location for your JDK.

3. When creating e*Way Connections from scratch, modify the directory locations in **compile.bat** (or, on UNIX, in **compile.sh**) as needed.

12.5.3 Compiling the .java Files and Creating the .jar File

Open a Command Prompt, change to the correct directory and run the **compile** script:

On Windows

```
.\compile.bat
```

On UNIX

```
./compile.sh
```

The script is designed to set the CLASSPATH and create the **.jar** file. The **.jar** file is then saved to the **gmeek\installETD\XAFileETD** directory.

12.5.4 Editing/Viewing the XAFileETD.ctl File

The **XAFileETD.ctl** file contains information required by the GUI to be able to successfully load the ETD. Any **.jar** files that are required by the ETD, such as third-party **.jar** files, must be included here.

XAFileETD.ctl

```
##-----------------------------------------------------------------
#
#  XAFileETD.ctl  (The ETD CTL file)
#
```

```
#  This CTL file is used by the GUIs.  It specifies the JAR files
#  that are needed by your ETD classes for compilation
#  (in a Collaboration).  It also specifies the JAR files
#  needed during run time.
##-------------------------------------------------------------------

##-------------------------------------------------------------------
#   JAR files containing the classes associated with your ETD
#
##-------------------------------------------------------------------
XAFileETD.jar,etd/XAFileETD,FILETYPE_BINTEXT
```

12.5.5 Editing/Viewing the XAFile.def File

For a general description of the purpose and operation of default configuration-file templates (**.def** files), see **"Editing/Viewing the .def Files" on page 176**. For this sample, you can add or modify additional parameters entries as needed if you plan to use this sample as a working code template. For more information on creating a custom **.def** file, see **Appendix A "Extending the .def File" on page 256**.

12.5.6 Editing/Viewing the XAFileETD.xsc File

The **.xsc** file provided with the sample contains the information provided by the **.java** files and required by the GUI. If you want the end user to see additional methods or properties in the GUI (ETD Editor, Collaboration Rules Editor), update this **.xsc** file appropriately. Complete details on the XSC formats are provided in **Appendix B "The XSC Format" on page 287**.

XAFileETD.xsc

```
<?xml version="1.0" encoding="UTF-8"?>
<etd name="XAFileETD" type="XAFileETD" xscVersion="0.6" uid="0" >
<javaProps package="xasample" codeAvailable="true" uid="1" />

<node name="XAFileETD" type="CLASS" uid="2">
    <method name="writefile" returnType="boolean" comment="This method
sends the passed string to the specified file (if specified) under XA
transactional environment." uid="3">
      <param name="XAFileETDObj" paramType="xasample.XAFileETD"
comment="The XAFileETD object itself or null for default." uid="4" />
      <param name="msg" paramType="java.lang.String" comment="The
message to post to the file." uid="5" />
      <param name="filenamepath" paramType="java.lang.String"
comment="The output file name with path or null for default." uid="6" />
      <param name="testdelay" paramType="java.lang.String" comment="A
test milliseconds delay string to allow manual error to be introduced or
null for default of none." uid="7" />
    </method>
  </node>
</etd>
```

12.6 Installing the Sample Files to e*Gate

After you have compiled the **.java** files to create one or more **.jar** file(s), customized the **.def** and **.ctl** files so they correspond to your environment, and created or customized the **.xsc** file for your ETD, you are ready to customize the **install.ctl** file and use the **installETD** script to commit all your files to the e*Gate Registry.

After compiling the **.java** files and creating the **.xsc** file, you must commit all changes and additions to the e*Gate Registry. The instructions for committing the changes are contained in the **installETD.*** and **install.ctl** files provided.

12.6.1 Customizing the install.ctl File

If you have made additional changes to the sample, modify the **install.ctl** file so that it correctly reflects all the following:

- The **.ctl** file used by your ETD at run time.
- The **.jar** files containing the classes associated with your ETD.
- Any third-party **.jar** files used.
- The **.xsc** file associated with your ETD.
- The **.def** file associated with your e*Way Connection.

12.6.2 Creating a Schema for the New ETD

If you do not have a pre-existing schema into which you want to commit the ETD files for the Transactional sample, start the Schema Designer and create a schema named, for example, **XAFile**.

12.6.3 Running the installETD Script

Do one of the following:

- On Windows, open a Command Prompt, change directories to **gmeek\installETD**, and enter the following command:

  ```
  .\installETD -e XAFileETD -s XAFile -h localhost -g c:\eGate
  ```

  If you are installing to a different schema, host, or directory, make the appropriate substitutions. For syntax details, see **Windows: installETD.bat** on page 76.

- On UNIX, open a shell, change directories to **gmeek/installETD**, and enter the following command:

  ```
  ./installETD.sh -e XAFileETD -s XAFile -h localhost -g /eGate
  ```

  If you are installing to a different schema, host, or directory, make the appropriate substitutions. For syntax details, see **UNIX: installETD.sh** on page 76.

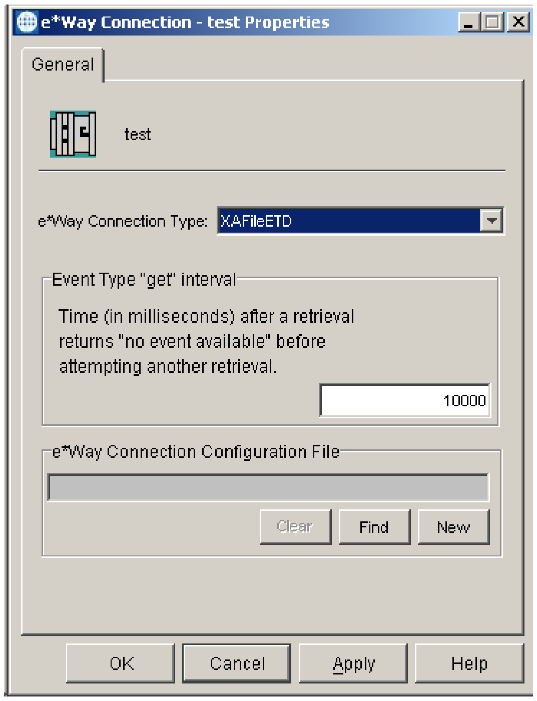
12.6.4 Validating the Sample Files Within e*Gate

You can double-check your **connectionpoint.ini** file and **XAFileETD.xsc** file within e*Gate by following these steps:

1 Start Schema Designer and open the schema into which the e*Way Connection was committed.

2 Create a new e*Way Connection. If all of the files committed successfully, the e*Way Connection Editor contains the XAFileETD entry as a possible e*Way Connection Type, as shown in Figure 57.

Figure 57 e*Way Connection Properties for XAFileETD


e*Way Connection - test Properties

General

test

e*Way Connection Type: XAFileETD

Event Type "get" interval

Time (in milliseconds) after a retrieval returns "no event available" before attempting another retrieval.

10000

e*Way Connection Configuration File

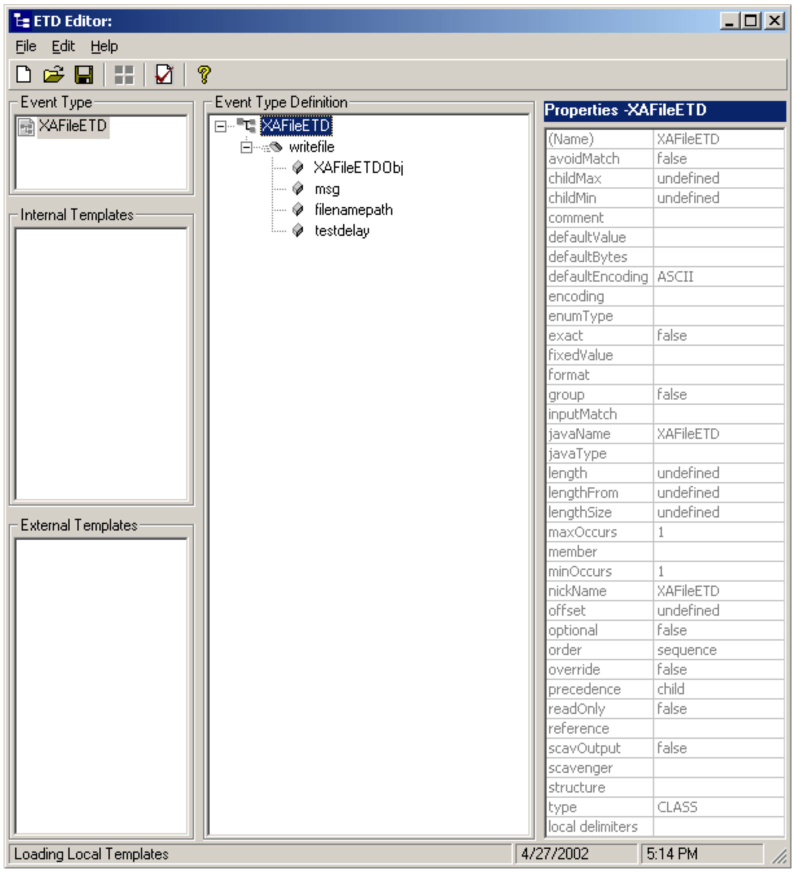
Note: *You do not need to create a new e*Way Connection Configuration file at this point.*

The contents of the e*Way Connection Type field correspond to the **connectionpoint.ini** file. If this field does not have an entry for XAFileETD, close the e*Way Connection Editor, make the necessary changes to the **connectionpoint.ini** file, and revalidate.

Note: *To edit the **connectionpoint.ini** from the sandbox, you must edit it in both the runtime and sandbox of the schema in the server repository. Also, you must remove the old existing file from the client.*

Start the ETD Editor and open the file **XAFileETD.xsc**, as shown in Figure 58.

Figure 58 ETD Editor Display of XAFileETD



Event Type: XAFileETD

```
XAFileETD
  writefile
    XAFileETDObj
    msg
    filenamepath
    testdelay
```

Properties -XAFileETD

(Name)	XAFileETD
avoidMatch	false
childMax	undefined
childMin	undefined
comment	
defaultValue	
defaultBytes	
defaultEncoding	ASCII
encoding	
enumType	
exact	false
fixedValue	
format	
group	false
inputMatch	
javaName	XAFileETD
javaType	
length	undefined
lengthFrom	undefined
lengthSize	undefined
maxOccurs	1
member	
minOccurs	1
nickName	XAFileETD
offset	undefined
optional	false
order	sequence
override	false
precedence	child
readOnly	false
reference	
scavOutput	false
scavenger	
structure	
type	CLASS
local delimiters	

Loading Local Templates | 4/27/2002 | 5:14 PM

If your sample does not look like this, analyze the problem, close the Editor, make the necessary changes, and validate it again. When your sample looks like Figure 58, you are ready to import the sample schema into e*Gate.

12.7 Understanding the ETD Implemented in a Schema

This section:

- Tells you how to load the sample version of the **XAFile** schema.
- Shows you how the sample components appear to the end user.
- Shows you how the sample components behave in the e*Gate environment.

12.7.1 Importing XAFile.zip Into e*Gate

To import the sample schema into e*Gate version 4.5.2 and later

1. Start the e*Gate Schema Designer GUI.
2. When the Schema Designer prompts you to log in, select the host that you specified during installation, and enter your password.

3 You are then prompted to select a schema. Click **New**. The New Schema dialog box opens. (Schemas can also be imported or opened from the e*Gate File menu by selecting **New Schema** or **Open Schema**.)

4 Enter a name for the new Schema, for example, **XAFile_Sample_Schema**, or any name as desired.

5 To import the sample schema select **Create from Export**, and use **Find** to locate and select the **XAFile.zip** file on the e*Gate Integrator Installation CD-ROM.

XAFile.zip is the file copied earlier in "Installing the Sample" on page 227.

The e*Gate Schema Designer opens to the new schema. You are now ready to make any configuration changes that may be necessary for this sample schema to run on your specific system.

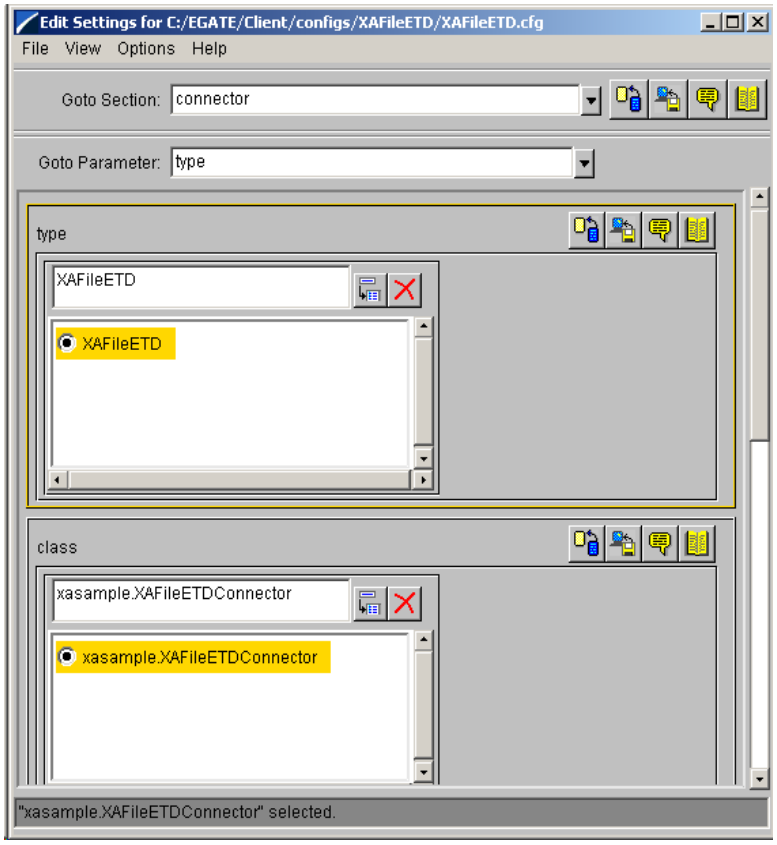
To validate the results

1 In the **XAFile** schema, open the **e*Way Connections** folder and verify that it contains an e*Way Connection named **XAFileETD**.

2 Edit the configuration settings of the **XAFileETD** e*Way Connection.

The **connector** section of the GUI should resemble Figure 59.

Figure 59 Configuration Editor Display of XAFileETD.cfg: Connector Section


Edit Settings for C:/EGATE/Client/configs/XAFileETD/XAFileETD.cfg

Goto Section: connector

Goto Parameter: type

type

XAFileETD

(●) XAFileETD

class

xasample.XAFileETDConnector

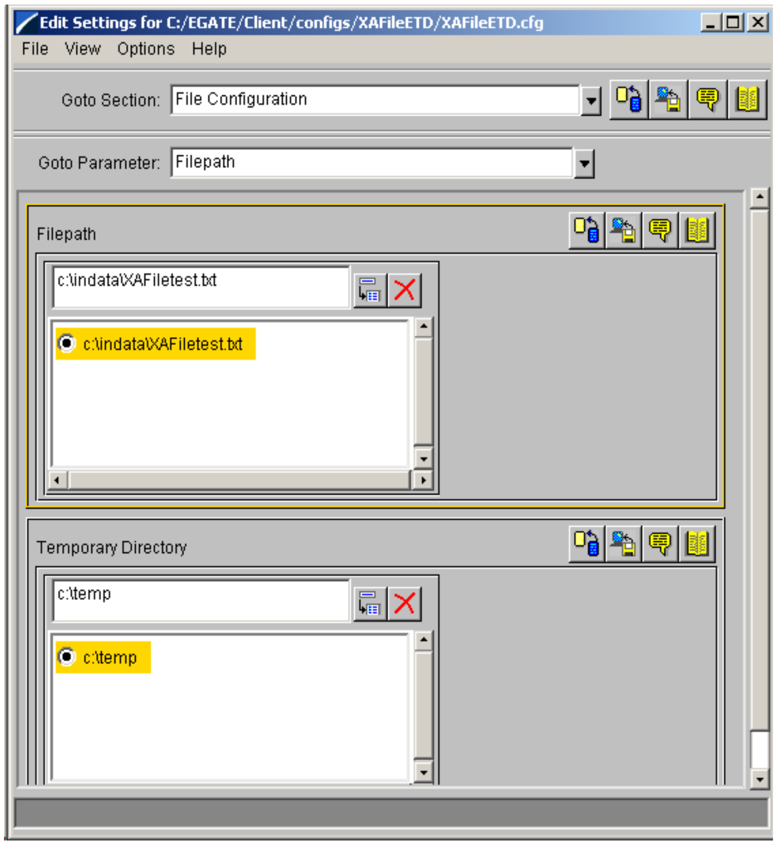
(●) xasample.XAFileETDConnector

"xasample.XAFileETDConnector" selected.

If you changed the name of the ETD and/or the name of the ETD Connector, update the **type** and **class** parameter entries accordingly.

The **File Configuration** section of the GUI should resemble Figure 60.

Figure 60 Configuration Editor Display of XAFileETD.cfg


Edit Settings for C:/EGATE/Client/configs/XAFileETD/XAFileETD.cfg

Goto Section: File Configuration

Goto Parameter: Filepath

Filepath: c:\indata\XAFiletest.txt
● c:\indata\XAFiletest.txt

Temporary Directory: c:\temp
● c:\temp

If one or more alternative directory locations are needed, update the **Filepath** and/or **Temporary Directory** parameter entries. Also, if the **XAFile.def** file was updated to contain different or additional parameters for the ETD and connector classes, create a new e*Way Connection configuration.

12.7.2 Sample INDATA

To validate the transactional sample using the data provided

1. Extract the **INDATA.zip** file into the **C:\INDATA\** directory.

   This is the location expected by the configuration of file e*Way being used as the feeder. The contents of the file are very simple:

   ```
   <SampleInput><PropsName>Color</PropsName><PropsValue>Blue</
   PropsValue></SampleInput>
   ```

2. Rename the sample input file from **SampleInput.~in** to **SampleInput.fin**

   The presence of a file with extension **.fin** triggers the file e*Way to read its contents.

3 From a Command Prompt, enter the following command to have the e*Gate Control Broker (**stccb.exe**) start the imported sample schema:

```
stccb -rh localhost -un Administrator -up STC -ln localhost_cb -rs
<schema name>
```

To use a Registry Host other than **localhost**, a username/password combination other than **Administrator**/**STC**, and/or a logical Control Broker name other than **localhost_cb**, make the appropriate substitutions.

Results

After a delay of approximately five seconds, a successful run generates the file **c:\indata\XAFiletest.txt** with the following content:

```
<Sun Jun 16 13:59:59 PDT 2002>
<SampleInput><PropsName>Color</PropsName><PropsValue>Blue</
PropsValue></SampleInput>
```

The timestamp prepended at the top of the file reflects the date and time of the run.

As the **SampleInput.fin** is renamed by the feeder e*Way back to **SampleInput.~in**, if you are looking at the temporary directory (**c:\temp** by default), you will see two temporary files flicker into and out of existence, with names like:

```
Xid-<fid>-<gid>-<bid>.start
Xid-<fid>-<gid>-<bid>.target
```

where

- *<fid>* is associated with the format ID of the Xid from the Transaction Manager.
- *<gid>* is associated with the global ID of the Xid from the Transaction Manager.
- *<bid>* is associated with the branch ID of the Xid from the Transaction Manager.

To re-run the test, rename the **SampleInput.~in** file back to **SampleInput.fin** (thus re-triggering the feeder e*Way).

Forcing an error condition and validating recovery

To test the recovery capability of this **XAFile** e*Way, re-run the test with one change: Immediately after you see the **Xid-*fff-ggg-bbb*.*** files appear in the temporary directory, manually stop the Control Broker process. (The default five-second delay was built in to the sample so as to provide enough time to do this.)

Stopping the Control Broker this way creates an external error condition, and thus no **XAFiletest.txt** output file will be created. However, the next time the Control Broker is manually started up and run normally, the rollback recovery logic will be automatically activated. The e*Way will use the temporary **Xid-*fff-ggg-bbb*.*** files to put the correct data into the **XAFiletest.txt** output file, without any need to re-read the input file **SampleInput.fin**. You can test this by deleting or moving the file out of the **C:\INDATA** directory.

Chapter 13

Best Practices

This chapter provides recommendations for developing and deploying custom e*Way Connections and e*Gate integrations. It consists of the following topics:

- Designing the classes for your e*Way Connection ETD on page 242.
  - Using the abstract class **EwayConnectionETDImpl**
  - Using the abstract class **EwayConnETDConnectorExtImpl**
  - The intended use of the Connector class and tips on where it should *not* be used.
  - Using a delegate class
  - Using inner classes
- Handling messageable ETDs on page 243.
- Handling exceptions on page 244.
- Troubleshooting and debugging on page 244.
  - Log files
  - Adding trace logs to your code
  - Sending Alerts
  - Using the e*Gate Java Debugger
- Cross-platform issues on page 251.
  - Wrapping third-party APIs using JNI
- Using JMS on page 252.
- Known Limitations on page 252.

Note: *In order to implement practices in this chapter, you should be thoroughly familiar with the design and development considerations discussed in* **Chapter 5**. *These same considerations apply to defining a viable overall integration architecture.*

13.1 Designing e*Way ETD Classes

13.1.1 General

The following are general tips and notes to consider when designing your e*Way.

- When creating classes for interfacing with an external system, the usual practice is to build your e*Way ETDs so that a class that encapsulates the calls to the external system's Java API is provided.
  - For more information, see **"Using a Delegate Class" on page 243**.
  - For an example, see **"SampleETDExternalClass.java" on page 171**.
- When defining the string constants and property names to be parsed from the **.def** configuration-file template, it is good practice to put all string constant definitions in a separate class.
  - For an example, see **"SampleETDDefs.java" on page 170**.
- An exception class that extends the **STCException** is used in your external API wrapper class.
  - See **"Handling Exceptions" on page 244**.
  - For an example, see **"SampleETDExternalException.java" on page 173**.
- The **STCException** class, provided in the **stcexception.jar** file, is an implementation of Chained Exceptions as described on the following web site:
  http://developer.java.sun.com/developer/technicalArticles/Programming/exceptions2

  Note: *For details on all classes and interfaces, see the Javadocs on the e*Gate Integrator Installation CD-ROM.*

The following sections provide specific tips on designing the classes for your e*Way Connection ETD.

13.1.2 Using Abstract Class EwayConnectionETDImpl

The ETD interface that implements the abstract class **EwayConnectionETDImpl** is used in all three samples provided with the kit. Each sample contains the same boilerplate code. The abstract class provides a default implementation that allows you to use the abstract class and extend it for your own ETD. Most of the methods have empty implementations. The methods that you must override to generate an e*Way Connection ETD are described in **"Creating .java Files" on page 56**.

13.1.3 Using Abstract Class EwayConnETDConnectorExtImpl

Two of the e*Way Connection samples provided—the Connection Management sample and the Transactional sample—use the **EBobConnectorExt** interface that implements the abstract class **EwayConnETDConnectorExtImpl**. Both samples contain the same boilerplate code. The abstract class provides a default implementation which again allows you to use the abstract class and extend it for your own connector. The methods that you must override to generate your e*Way connector class are described in **"Creating .java Files" on page 56**.

13.1.4 Do's and Don'ts for the Connector Class

The connector class must only contain the functionality that obtains the connection properties and opens and closes the connection to the external system. This may involve using calls to the external system's API. The connector class should not contain methods that perform external operations other than those that involve configuring or establishing a connection. Instead, it is good practice to use a delegate class to wrap calls that use the external system's API to perform these operations.

13.1.5 Using a Delegate Class

The samples provided show the use of a class that contains the API calls to interface directly with the external system. The ETD class serving as a Delegate object which performs the actual work behind the external methods exposed by the ETD contains a reference to this class. This practice is optional, as some developers may not want the extra layer of coding involved. Use of delegation works very well for encapsulating a well-defined set of calls in third-party APIs. Other design patterns, where appropriate, can also be used in designing the classes around the methods exposed by the ETD class.

13.1.6 Using Inner Classes

Use of inner classes in ETDs is recommended. These inner classes can encapsulate entities associated with the main entity represented by the ETD. For example, an ETD that represents an Account object in SAP might contain an inner class that holds account details. Inner classes are used for objects passed as input parameters to Account methods, and can be used for the output parameters are used as well.

13.2 Handling Messageable ETDs

Although this document focuses on developing nonmessageable ETD classes only, a Collaboration is configured with both messageable and non-messageable ETDs on either end (inbound or outbound).

- A messageable ETD on the inbound side must be unmarshaled (parsed) for its data to be available to e*Gate. This is done when the Multi-Mode e*Way calls **unmarshal()** on the ETD class.
- When a messageable ETD is on the outbound side of the Collaboration, it must be marshaled from Java objects to byte streams, and sent to a persistent destination such as a Oracle SeeBeyond JMS queue or topic, or IQ.

When non-messageable or e*Way ETDs are specified on the inbound side of the Collaboration, the **"get" interval** configuration for the corresponding e*Way

Connection is used as the interval between business rules execution. (For further information on the "get" interval, see the *e*Gate Integrator User's Guide*.) This allows users to write Collaborations to poll the external system periodically for data.

If non-messageable ETDs are on the outbound side of the Collaboration, they normally perform an operation based on rules that depend in some way on the data coming from the inbound side of the Collaboration. Note that when messageable data needs to be parsed, data is extracted from messageable ETDs and passed to non-messageable ETDs (usually as parameters to methods). The parsing functionality is not normally part of non-messageable ETDs, so it is common practice to use multiple ETDs, bringing messageable ETDs into a Collaboration if parsing data is needed. Also, if data obtained from the external system has to be saved as a message for placement into queues, it normally needs to be processed by messageable ETDs on the outbound side of the Collaboration.

13.3 Handling Exceptions

Your implementation of e*Way Connection ETDs will need to handle exceptions at the appropriate places. It is good practice to define specific exception classes by subclassing the **com.stc.eways.exception.STCDataException** class. This is an implementation of *chained exceptions*, based on Sun's recommendations published on the following web site:

http://developer.java.sun.com/developer/technicalArticles/Programming/exceptions2

When a Collaboration throws an exception, it rolls back the Event received by the Collaboration. As a developer, you can throw Collab[...]Exception exceptions from your ETD class—such as CollabConnException, CollabDataException, or CollabResendException—and let the user handle them in the Collaboration. You can also throw more specific exceptions for the user to handle in the Collaboration.

Exception Handling Within ETD Entities

Use the class **com.stc.jcs.JCSProperties** (see **"JCS Properties" on page 305**) to hold the **throws** attribute data.

13.4 Troubleshooting and Debugging

The e*Gate system provides the following basic troubleshooting tools:

- Log files, trace logging, and levels/flags
- Schema Monitoring and Alert notifications
- e*Gate Java Debugger

This section provides an introduction to these e*Gate features.

13.4.1 Log Files

An important source of information about your e*Gate configuration comes from log files. The e*Gate system's logging facility allows you to trace and store detailed operations information.

Each of the following components can generate log files:

- e*Way Intelligent Adapters
- Business Object Brokers (BOBs)
- Intelligent Queue (IQ) Managers
- Control Brokers
- e*Insight Business Process Manager modules

Each log file is clearly labeled as belonging to the component that generated it. You can control the type and amount of debugging information that appears in the log file for each component.

13.4.2 Adding Trace Logging

When you write the Java code for an e*Way Connection or a user Collaboration, it is standard practice to add calls to display trace logs. The log generated is stored in a file called <*eWayComponent*>**.log,** where <*eWayComponent*> is the logical name for the Multi-Mode e*Way configured in the Schema Designer GUI. The log file is stored in <*eGateRootDirectory*>**\client\logs** where <*eGateRootDirectory*> is the root directory in which e*Gate is installed.

To add program trace logging, use the **traceln()** method of the class **com.stc.common.collabService.EGate**, using one of the following two signatures.

```
public static void traceln(long tid, long event,
                           java.lang.String message)

public static void traceln(long tid, long event,
                           byte[] blob, java.lang.String tracestr)
```

The value to use for the **event** flag (the second parameter) for ETD debugging purposes is defined as follows:

```
public final static long TRACE_EVENT_DEBUG        = 0x00000001;  // (D)
public final static long TRACE_EVENT_TRACE        = 0x00000002;  // (T)
public final static long TRACE_EVENT_INFORMATION  = 0x00000004;  // (I)
public final static long TRACE_EVENT_WARNING      = 0x00000010;  // (W)
public final static long TRACE_EVENT_APIERROR     = 0x00000020;  // (A)
public final static long TRACE_EVENT_LOGERROR     = 0x00000040;  // (E)
public final static long TRACE_EVENT_FATAL        = 0x00000080;  // (F)
public final static long TRACE_EVENT_ERR          = TRACE_EVENT_APIERROR;
```

The **traceln()** output is written to the log file according the Event mask. As shown in Table 16, the trace entry is shown only if the Trace ID (debugging flag) is active and the Trace Event Logging Level) is in effect at run time.

Table 16 Trace Events (Logging Levels)

Logging Level	TRACE_EVENT_...					
	...TRACE	...DEBUG	...INFORMATION	...WARNING	...LOGERROR	...FATAL
TRACE	**shown**	**shown**	**shown**	**shown**	**shown**	**shown**
DEBUG	not shown	**shown**	**shown**	**shown**	**shown**	**shown**
INFO	not shown	not shown	**shown**	**shown**	**shown**	**shown**
WARNING	not shown	not shown	**shown**	**shown**	**shown**	**shown**
ERROR	not shown	not shown	**shown**	**shown**	**shown**	**shown**
FATAL	not shown	not shown	**shown**	**shown**	**shown**	**shown**
NONE	not shown	not shown	**shown**	**shown**	**shown**	**shown**

The Logging Level decreases in intensity from top (TRACE) to bottom (NONE). As indicated in Table 16, some Event masks are *always* shown, regardless of Logging Level.

Note: ***TRACE_EVENT_LOGERROR** places an entry in the log file as well as in the Windows Event Viewer System Log. It is good practice to restrict its use to highly important or urgent errors only.*

When using EGate.**traceln()** for debugging purposes, it is good practice to use TRACE_EVENT_TRACE or TRACE_EVENT_DEBUG. This way, after your ETD is up and running properly, you can configure it to use the INFO logging level (which generates a much smaller amount of data).

Note: *When debugging passwords, be sure to keep them encrypted. Use the **ScEncrypt.decrypt** utility to decrypt an encrypted password. For more information, see* **"Encrypting Strings" on page 271**.

13.4.3 Debug Levels and Flags

The e*Gate system has the ability to record many kinds of information in log files, depending on the debug levels and flags you set. This logging control feature helps you log only the information you want to record.

Each system Event that is logged has the following basic properties:

- A *debug level* that describes the general nature of a system Event, for example, whether the Event is fatal, nonfatal error-related, or only informational.
- One or more *debug flags* that describe the source of a system Event. For example, **CB** Events originate from the Control Broker, **EWY** Events originate from e*Ways, **MNK** Events describe the workings of Monk scripts, and so on. Many components produce both verbose-mode and normal-mode (nonverbose) system Events.

When you set up logging for a component in the Schema Designer, you specify the debug level and flags of the system Events you want to log. The resulting log file will contain a record of only those system Events whose level and flags match the level and flags you specified.

13.4.4 Alert Notifications

The e*Gate system continually issues *monitoring Events* to provide information on how well the overall system is functioning. All major e*Gate components and features issue these Events through internal system operations. The Control Broker converts monitoring Events into *notifications (notification Events)* and sends them to Schema Monitors.

Note: *In e*Gate, an Event is a package of data and an e*Gate Event is a packet of information that is exchanged with external applications.*

Notifications that indicate problems or change of status inside e*Gate are called *Alert notifications*. These appear as message readouts in the Schema Manager GUI which provide immediate information on system problems.

The Schema Manager window also displays readouts of monitoring Event and notification status messages with information on the normal functioning of components. However, the Alert notifications, called *Alerts* in the GUI, contain the actual e*Gate error messages.

Alert notifications provide important information on system problems and where to start your troubleshooting. You can also monitor status and Alert notifications by other means.

For complete information, refer to the following:

- The *Alert Agent User's Guide*
- The *e*Gate Integrator Alert and Log File Reference Guide*.
- The *e*Gate Integrator User's Guide*.

13.4.5 Using the eventSend() Method to Send Alert Notifications

The **eventSend()** method is used to send Alert notifications. Complete information about this method is provided in the Javadocs on the e*Gate Integrator Installation CD-ROM and in the *e*Gate Integrator User's Guide*.

Example

```
eventSend(Alerter.ALERTCAT_MESSAGE_CONTENT,
          Alerter.ALERTSUBCAT_USERAUTH,
          Alerter.ALERTINFO_IOFAILED,
          35827, "Disk Full", "", "");
```

Description

The **eventSend()** method sends Alert notification Events to the Control Broker. These are viewed in the Schema Manager or the Alert Agent, if they are configured to display them.

Access to the **eventSend()** method is provided indirectly through the **JCollaboration** class, supplied in the package **com.stc.jcsre.JCollaboration**, from which all Java Collaboration are subclassed.

```
public abstract class JCollaboration
extends java.lang.Object
```

However, **eventSend()** is a member method of the **Alerter** class, supplied in the package **com.stc.common.collabService**, class **JCollabController**:

```
public class JCollabController;
public class Alerter
extends java.lang.Object
```

Syntax

```
public boolean eventSend(alertCategory, alertSubcategory,
                         alertInfoCode, reasonCode, reasonName,
                         eventInfo, additionalInfo)

public boolean eventSend(severityLevel, alertCategory,
                         alertSubcategory, elemType, reasonCode,
                         reasonName, eventInfo, additionalInfo)

eventSend(com.stc.common.collabService.JCollabController jController,
          alertCategory, alertSubcategory, alertInfoCode, reasonCode,
          reasonName, eventInfo, additionalInfo)
```

Parameters

Name	Type	Description
alertCategory	java.lang.String	Alert-Category Constant (see **"alertCategory Constants"**).
alertSubcategory	java.lang.String	Alert-Subcategory Constant (see **"alertSubcategory Constants"**).
alertInfoCode	java.lang.String	Info-Code Constant (see **"alertInfoCode Constants"**).
reasonCode	int	Status or error code generated by the operating system or the application generating the Event.
reasonName	java.lang.String	Reason why the Event occurred.
eventInfo	java.lang.String	Reserved for user agents or other applications using the API to create monitoring Events that use this field. It is an empty string ("").
additionalInfo	java.lang.String	Reserved for future use. It is an empty string ("").
severityLevel	java.lang.String	Severity-level Constant (see **"severityLevel Constants"**).
elemType	java.lang.String	Element-type Constant (see **"elemType Constants"**).

alertCategory Constants

Name	Description
Alerter.ALERTCAT_STATE_ELEM	Element state
Alerter.ALERTCAT_MESSAGE_CONTENT	Message content
Alerter.ALERTCAT_STATE_EXTERNAL	External state
Alerter.ALERTCAT_OPERATIONAL	Operational
Alerter.ALERTCAT_PERFORMANCE	Performance

Name	Description
Alerter.ALERTCAT_RESOURCE	Resource
Alerter.ALERTCAT_USERDEFINED	User-defined

alertSubcategory Constants

Name	Description
Alerter.ALERTSUBCAT_CUSTOM	Custom category
Alerter.ALERTSUBCAT_DOWN	Down
Alerter.ALERTSUBCAT_UP	Up
Alerter.ALERTSUBCAT_UNRESP	Unresponsive
Alerter.ALERTSUBCAT_RESP	Respond
Alerter.ALERTSUBCAT_CANTCONN	Unable to connect
Alerter.ALERTSUBCAT_CONN	Connected
Alerter.ALERTSUBCAT_LOSTCONN	Lost connection
Alerter.ALERTSUBCAT_UNUSABLE	Unusable/cannot ID
Alerter.ALERTSUBCAT_INTEREST	Interest
Alerter.ALERTSUBCAT_EXPIRED	Expired
Alerter.ALERTSUBCAT_INTHRESH	Input threshold
Alerter.ALERTSUBCAT_OUTTHRESH	Output threshold
Alerter.ALERTSUBCAT_USERAUTH	User authentication
Alerter.ALERTSUBCAT_DELIVERY	Alert delivery
Alerter.ALERTSUBCAT_UNQUEUEABLE	Unqueueable
Alerter.ALERTSUBCAT_DISKTHRESH	Disk threshold
Alerter.ALERTSUBCAT_IQLIMIT	IQ limit
Alerter.ALERTSUBCAT_STATUS	Status
Alerter.ALERTSUBCAT_TIMER	Timer

alertInfoCode Constants

Name	Description
Alerter.ALERTINFO_NONE	None
Alerter.ALERTINFO_FATAL	Fatal
Alerter.ALERTINFO_CONTROLLED	Controlled
Alerter.ALERTINFO_USER	User
Alerter.ALERTINFO_LOW	Low
Alerter.ALERTINFO_HIGH	High

Name	Description
Alerter.ALERTINFO_IOFAILED	I/O failure
Alerter.ALERTINFO_BELOW	Below
Alerter.ALERTINFO_ABOVE	Above

severityLevel Constants

Name	Description
Alerter.SEVERITY_LEVEL_UNDEFINED	Undefined
Alerter.SEVERITY_LEVEL_TRACE	Trace
Alerter.SEVERITY_LEVEL_DEBUG	Debugging
Alerter.SEVERITY_LEVEL_INFO	Information
Alerter.SEVERITY_LEVEL_WARNING	Warning
Alerter.SEVERITY_LEVEL_ERROR	Error
Alerter.SEVERITY_LEVEL_FATAL	Fatal

elemType Constants

Name	Description
Alerter.ALERTCAT_STATE_ELEM	Element state
Alerter.ALERTCAT_MESSAGE_CONTENT	Message content
Alerter.ALERTCAT_STATE_EXTERNAL	External state
Alerter.ALERTCAT_OPERATIONAL	Operational
Alerter.ALERTCAT_PERFORMANCE	Performance
Alerter.ALERTCAT_RESOURCE	Resource
Alerter.ALERTCAT_USERDEFINED	User-defined

Return Type

Boolean—Returns **true** when an Alert Event is sent successfully.

Throws

None.

13.4.6 Debugging

You can use the e*Gate Java Debugger to debug Collaborations and e*Way ETDs. The debugger attaches to the JVM and connects to a particular port on the host on which you are running the Collaboration. The e*Gate Java Debugger is based on the Java Platform Debugger Architecture (JPDA) Java Debug Interface (JDI).

To access the debugger

1 In Schema Manager, right-click the Multi-Mode e*Way and, on the shortcut menu, click **Debugger**.

2 The **e*Gate Java Debugger** dialog box appears, and the **Stop in Method** dialog box prompts you to specify the Collaboration Rules you want to debug.

You can use the e*Gate Java Debugger to set and clear breakpoints, go to a specific statement, step into, over, or out of specific blocks of code, stop in a specific class or method, break on a specified exception, and perform other operations.

For complete information on the e*Gate Java Debugger, refer to the *e*Gate Integrator User's Guide*.

13.4.7 Using Internal Templates

When using internal templates, you may encounter the following:

```
GmeekETDNode TemplateNode =
aEtd.addInternalTemplate("INTERNAL_TEMPLATE1");
TemplateNode.addInnerNode("INNERNODE_NAME1");
TemplateNode.addSimpleTypeUsrField("FIELD_NAME1",
GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
TemplateNode.setInternalTemplateName("INTERNAL_TEMPLATE1");
```

This generates the following error message:

```
Exception in thread "main" java.io.IOException: Compiler errors.
F:/java_compile/output/com/sbyn/TestOutput/TestOutput.java:36: cannot
resolve symbol
symbol  : class INTERNAL_TEMPLATE1
location: class com.sbyn.TestOutput.TestOutput
            private INTERNAL_TEMPLATE1 _INTERNAL_TEMPLATE1 = null;
                    ^
```

If you receive the error message, check your code. The **setInternalTemplate()** method should be used against an actual node instead of against the actual templates. To avoid generating an error, you must use internal templates as shown in the following example:

```
GmeekETDNode TemplateNode =
aEtd.addInternalTemplate("INTERNAL_TEMPLATE1");
GmeekETDNode InnerNode =
TemplateNode.addInnerNode("INNERNODE_NAME1");
InnerNode.addSimpleTypeUsrField("FIELD_NAME1",
GmeekTreeNode.TreeNodeType.FIELD_TYPE_STRING);
InnerNode.setInternalTemplateName("INTERNAL_TEMPLATE1");
```

For more information on error messages, see **"Working With the Back-end Builder" on page 252**.

13.4.8 Wrapping Third-Party APIs Using JNI

Some third-party systems provide only C or C++ APIs for certain OS platforms. In such cases, the APIs are usually wrapped using Java Native Interface (JNI) so they are exposed as Java classes. If you use JNI to wrap a third-party API, be sure to specify the

JNI library component in your **.ctl** file, as well as any libraries it depends on. Any other support classes (usually stored in a **.jar** file) that are used by the JNI component should also be specified in the **.ctl** file for your ETD.

13.5 Oracle SeeBeyond JMS

For information on how the Oracle SeeBeyond implementation of JMS appears to the e*Gate end user, refer to the *Oracle SeeBeyond JMS Intelligent Queue User's Guide*.

Information on JMS architecture, design, and APIs is contained in the *e*Gate API Kit Developer's Guide*.

13.6 Working With the Back-end Builder

This section provides best practices for designing the back-end builder. This includes performing incremental tests at each step of the design process, and taking various efficiency measures when designing the front-end GUI.

Incremental Testing

You do not have to finish the GUI portion in order to test the generated ETD. To test the ETD, you can invoke the back-end builder through the command line and it will generate an ETD. To load this ETD to the e*Gate registry, use the **stcregutil** command. For more information on using the **stcregutil** command, see **"Task 6: Creating and Registering the ETD Using the Command Line" on page 148**.

To test the generated runtime package

1. Call the open (Properties connectorProps) method from a test driver, making sure you can connect to the external system.
2. Run the **installEWAY** script to install the runtime package to e*Gate.
3. Modify the **stcew<EwayName>.ctl** file to include the packages you have created, such as the third party **.jar** files and various utility **.jar** files, **.def** files, and runtime **.jar** files.
4. Run the **installEWAY** script to load the **.jar** files. Whenever loading a new e*Way to the registry, we recommend loading it to a default schema.

Designing the Front-End GUI

When designing the front-end GUI for the back-end builder, it is recommended that you take the following measures:

- The first step in the design process is to decide whether to use heavyweight Visual Basic or lightweight Visual Basic. This is imperative because if you have already implemented heavyweight Visual Basic in production, the lightweight Visual Basic wizard cannot modify the ETD built with heavyweight Visual Basic. The same is true if implementing lightweight Visual Basic in production—the heavyweight Visual Basic wizard cannot modify an ETD built with lightweight Visual Basic.

Note: *For more information on using heavyweight or lightweight Visual Basic, see* **"Using Heavyweight or Lightweight Visual Basic to Create an ETD Builder Wizard" on page 101**.

- Try to store as much data as possible in the Visual Basic layer, and pass the simple data structured properties through JINTEGRA. For example, use a Visual Basic Dictionary to store the structured properties, but only pass the individual properties one at a time through JINTEGRA as String, int, long, etc.
- In the sample Visual Basic project, we use QAWizards, an executable which invokes the wizard DLL. We recommend using QAWizards to test the wizard.
- We recommend utilizing the Visual Basic debugger.
- If your wizard DLL is built correctly, the **.xsc** file displays properly in a text editor. Verify that the **.xsc** is correct by viewing it in a text editor.

Chapter 14

e*Way Classes and Methods

The Generic Multi-Mode e*Way Extension Kit contains Java methods that are used to extend the functionality of e*Ways. For example, Java methods are added to make it easier to set information in and get information from the e*Way ETD Editor.

The Generic Multi-Mode e*Way Extension Kit Java methods are organized into the following related Java classes.

- EBobConnector
- EBobConnectorExt
- EBobConnectorExtFactory
- EBobConnectorExtImp
- EBobConnectorFactory
- Egate
- ETD
- GmeekConnectorModel
- GmeekETDMethod
- GmeekETDNode
- GmeekETDRootNode
- GmeekEwayModel
- JCollabController
- JConnectionManager
- JConnectionNotifier
- JTransactionAdapter
- JXAResourceAdapter
- ScEncrypt
- STCException

The various Java classes and methods available for use in Collaborations are described in the Javadocs. The Javadocs provide a complete explanation of each method.

14.1 Viewing Javadocs

For e*Gate version 4.5.3 or later, to view Javadocs from the CD-ROM

1. From the GUI CD-ROM, navigate to **\docs\Javadocs\GMEEK\html**
2. Click on **index.html** to view the Generic Multi-Mode e*Way Extension Kit Javadocs in your Web browser.

For e*Gate version 4.5.3 or later, to view Javadocs installed on your local drive

1. Navigate to <**eGate>\client\docs\Javadocs\GMEEK\html**
2. Click on **index.html** to view the Generic Multi-Mode e*Way Extension Kit Javadocs in your Web browser.

For e*Gate version 4.5.2 or earlier, to view Javadocs from the CD-ROM

1. From the GUI CD-ROM, navigate to **\docs\Javadocs\GMEEK\html**
2. Click on **index.html** to view the Generic Multi-Mode e*Way Extension Kit Javadocs in your Web browser.

For e*Gate version 4.5.2 or earlier, to copy Javadocs to your local drive

You must manually copy the Generic Multi-Mode e*Way Extension Kit Javadocs from the GUI CD-ROM to your local drive.

1. Navigate to **\docs\Javadocs**
2. Select and copy the **GMEEK** folder to **<eGate>\client\docs\Javadocs**

   The **GMEEK\html** folder contains the com folder and the files shown in Table 17.

Table 17 Contents of the **GMEEK\html** folder

Folder and File names
com
allclasses-frame.html
help-doc.html
index.html
index-all.html
overview-summary.html
overview-tree.html
package-list
packages.html
serialized-form.html
stylesheet.css

Important: *You must copy the entire contents of the* **GMEEK** *folder.*

3. Navigate to the location where you copied the **GMEEK** folder, then to the **html** directory.
4. Click on **index.html** to view the Generic Multi-Mode e*Way Extension Kit Javadocs in your Web browser.

Appendix A

Extending the .def File

This appendix describes how to extend the default configuration-file template (**.def** file) and describes the **.def** file keywords and their arguments. In addition, it also describes how to test and debug the **.def** file and lists some common error messages. It also provides information on configuration parameters and the **.cfg** file.

A.1 Introduction

The e*Way Connection is configured using the e*Way Connection Editor, a GUI that enables you to change configuration parameters quickly and easily. A default configuration-file template (**.def** file) allows the e*Way Connection Editor to gather those parameters by specifying the name and type of each parameter, as well as other information (such as the range of permissible options for a given parameter).

The e*Way Connection Editor stores the values that you assign to those parameters in *two* configuration files, the **.cfg** file and the **.sc** file. These two files contain similar information but are formatted differently:

- The **.cfg** file contains the parameter values in delimited records and is parsed by the e*Way at run time.
- The **.sc** file contains the parameter values and additional information needed by the GUI.

The e*Way Connection Editor loads the **.sc** file—not the **.cfg** file— when you edit the configuration settings for an e*Way. Both configuration files are generated automatically by the e*Way Connection Editor whenever the configuration settings are saved.

The **.def** file for the e*Way Connection contains a set of parameters that are required and must not be modified. You can extend the **.def** file if your modifications to the e*Way Connection require the definition of user-set parameters. This chapter describes the structure of the **.def** and configuration files and the syntax of the keywords used to configure the e*Way Connection Editor to gather the desired configuration parameters.

Important: *We strongly recommend that you make no changes whatsoever to the default* ***sampleETD.def*** *file. However, you should use that file as a base from which you create your extensions. Save a copy of the file under a unique name and make your changes to the copy.*

A.1.1 Layout

The **.def** file has three major divisions:

- The **header** describes basic information about the file itself, such as version number, modification history, and comments.
- The **sub-header** contains several read-only variables that are for internal use only. The default values of these variables must not be modified.
- The **body** contains configuration parameters which are grouped into sections. Two sections (Connector and External Configuration) must be included in all e*Way Connection **.def** files; additional sections are added as needed to support user-created functions.

A.2 .def file Keywords: General Information

All keywords and their arguments are enclosed in balanced parentheses. Keyword arguments are a quoted string, a quoted character, an integer, a parenthesis-bounded list, a keyword modifier, or additional keywords.

Examples:

```
(name “Sample e*Way“ )

(eway-type
     (direction "<ANY">)
)

(set
     (value (1 2 3))
     (config-default (1 2 3))
)

(range
      (value (const 1 const 1024)
)
```

A.2.1 White Space

White space that is not contained within double-quotation marks, including tabs and newlines, is ignored except as a separator between keywords.

For example, the following are equivalent:

- (user-comment (value "") (config-default ""))
- (user-comment
      (value "")
      (config-default "")
)

Whitespace within quotation marks is interpreted literally. For example, **(name “Extra   Spaces”)** will display as

```
Extra     Spaces
```

in the e*Way Connection Editor's list of names.

A.2.2 Integer Parameters

The maximum value for integer parameters ranges from approximately -2 billion to 2 billion (specifically, -2,147,483,648 to 2,147,483,647). Most ranges will be smaller, such as "1 to 10" or "1 to 1,000."

A.2.3 Floating-point Parameters

Floating-point parameters and floating-point arithmetic are not supported.

A.2.4 String and Character Parameters

String and character parameters may contain all 255 ASCII characters. The "extended" characters are entered using an escaped format, as follows:

- Characters such as tab, newline, and carriage return are entered as **\t**, **\n**, and **\c**, respectively.
- Characters may also be entered in octal or hexadecimal format using **\o** or **\x**, respectively (for example, \x020 for ASCII character 32).

Strings are delimited by double quotes, characters by single quotes. Examples:

- Strings: `"abc" "Administrator"`
- Characters: `'0' '\n'`

Single quotation marks, double quotation marks, and backslashes that are not used as delimiters (for example, when used within the text of a description) must be escaped with a backslash:

- `\'` —Use this two-character sequence to code the single-quote character (').
- `\"` —Use this two-character sequence to code the double-quote character (").
- `\\` —Use this two-character sequence to code the backslash character (\ ).

A.2.5 Path Parameters

Path parameters can contain the same characters as other string parameters. However, the characters entered should be valid for path names within the operating system on which the e*Way runs.

Backslashes in Windows path names must be escaped (for example,**\\home\\egate**).

A.2.6 Comments

Comments within the **.def** file begin with a semicolon (;). Any semicolon that appears in column 1, or that is preceded by at least one space character and that does not appear within quotation marks, is interpreted as a comment character. You cannot represent a null in a comment; in other words, 0x00 (or \000) is an impermissible character.

Examples

```
; this is a valid comment, because it begins in column 1
(name "Section name") ; this is also a valid comment, because its semicolon is preceded by a space
```

A.2.7 Header Information

Header information that developers can use to maintain a revision history for the **.def** file is stored within the **(general-info)** section. All the information in this section is maintained by the user; no e*Gate product modifies this information.

Table 18 describes the user-editable parameters in the **(general-info)** section. The use of these fields is not required and they may be left blank, but all the fields must be present. The format and contents of these fields are completely at the developer's discretion as long as rules for escaped characters are observed. For more information, see **"String and Character Parameters" on page 258**.

Note: *Any **(general-info)** parameters that are not shown in Table 18 are reserved and should not be modified except by direction of Oracle Support staff.*

Table 18 User-editable **(general-info)** parameters

Parameter name	Describes
version	The product version name
revision	The revision number
user	The user who last edited the file
modified	The modification date
creation	The creation date
description	A description for this **.def** file, displayed within the e*Way Connection Editor from the File menu's **Tips** option. Quotation marks within the description, whether single (') or double (") must be backslash-escaped (\' or \").
user-comment	Comments left by the user (rather than the developer), accessed within the e*Way Connection Editor from the File menu's **User notes** option. Unless you want to provide a default set of user notes, you should leave this field blank.

A.3 Defining a New Section

The **(section)** keyword defines a section within the **.def** file. The syntax of the new section is described in **"Section Syntax"**. Each section requires at least one parameter. For more information on defining parameters, see **"Parameter Syntax" on page 261**.

Note: *Section names and parameter names within a section must be unique.*

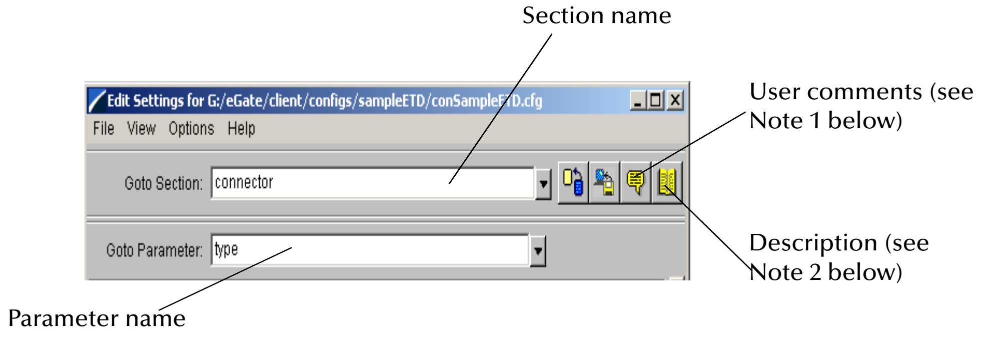
A.3.1 Section Syntax

Sections within the **.def** file have the following syntax:

```
(section
  (name "section name")

... at least one parameter definition ...

(description "description text")
(user-comment
    (value "")
    (config-default "")
  ) ; end of user comment
) ; end of section
```

The section name, description text, and user-comment “value” will appear in the e*Way Connection Editor. An example is shown in Figure 61.

Figure 61 e*Way Connection Editor Main Controls


Section name
User comments (see Note 1 below)
Description (see Note 2 below)
Parameter name
Edit Settings for G:/eGate/client/configs/sampleETD/conSampleETD.cfg
File View Options Help
Goto Section: connector
Goto Parameter: type

Notes

1 The user-comment feature enables users to make and save notes about a section or parameter that will be stored along with the configuration settings. Under most circumstances, we recommend that developers leave the **(user-comment)** fields blank, but you can enter information in the **(user-comment)** field if you want to ensure that all user notes for a given section begin with preset information.

2 The description is displayed when the user clicks the **Tips** button. Use this field to create “online help” for a section or parameter. We recommend that you provide a description for every section and every parameter that you create.

A.3.2 Parameter Syntax

Parameters within the **.def** file use the following basic structure:

```
(param-keyword
     (name "Parameter name goes here")
     (value val)
     (config-default val)

     ...additional keywords (range, units, set) as required...

     (description "description text")
     (user-comment
       (value "")
       (config-default "")
     )
  )  ; end of parameter definition
```

The keywords that are required to define a parameter are

- A parameter keyword
- The parameter's name: **(name)**
- The initial default value: **(value)**
- The "configuration default": **(config-default)**, which the user can restore by clicking the revert button, shown below.

  This value is overridden by the **config-default** keyword specified within a **(set)** command; see **"Parameters Accepting a Single Value From a Set" on page 263** and **"Parameters Accepting Multiple Values From a Set" on page 264** for more information.

Note: *The **(value)** keyword is* **always** *followed immediately by the **(config-default)** keyword.*

- The "description" (for additional information, see the Notes for **"Section Syntax" on page 260**).
- The "user comment" (for additional information, see the Notes for **"Section Syntax" on page 260**), which has its own value and configuration default.

Additional keywords may be required, based upon the parameter keyword and user requirements; these will be discussed in later sections.

Order of Keywords

Keywords must appear in this order:

1. parameter definition*
2. name*
3. value*

4 config-default*

5 set

6 range

7 units

8 show-as

9 factor

10 description*

11 user-comment*

Note: *Keywords marked with * are mandatory for all parameters. The **set** keyword is mandatory for **-set** and **-set-multi** parameters. The remaining keywords (items 6 through 9) are optional and, depending on developer requirements, may appear in any combination, but they must appear in the above order.*

Parameter Types

There are eight types of parameters. Table 19 lists the types of parameters that are defined, the keyword required to define them, and the values that the keyword can accept for the **(value)** and **(config-default)** keywords.

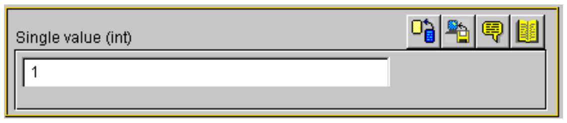
Table 19 Basic Parameter Keywords

Type	Parameter keyword	Accepts values	Example
Integer	int	integer	**7500**
Character	char	single-quoted character	**'a'** **'!'** **'\o123'** (octal)
String	string	double-quoted string	**"Hello, world"**
Date	date	comma-delimited date in *MMM,dd,yyyy* format	**AUG,13,2002**
Time	time	colon-delimited time in 24-hour *hh:mm:ss* format	**15:30:00**
Path	path	path; Windows path names should use escaped backslashes	**/home/egate/client** (UNIX) **\\home\\egate\\client** (Windows)

Parameters Requiring Single Values

Parameters requiring single values are defined within the basic structure shown in **"Section Syntax" on page 260**.

Figure 62 A parameter requiring a single value



The parameter is defined using a parameter keyword, as listed in **Table 19 on page 262**.

Example

To create a parameter that accepts a single integer as input, and to specify “3” as the default and configuration-default value, enter the following:

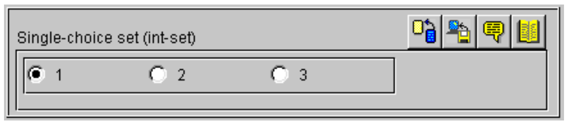
```
(int
    (name "Parameter requiring a single integer")
    (value 3)
    (config-default 3)
    (description "

      This parameter requires a single integer as input.
")
    (user-comment
      (value "")
      (config-default "")
    )
  )  ; end of parameter definition
```

If you want to limit the values that the user may enter, you may include the optional **(range)** keyword; see **“Specifying Ranges” on page 266** for more information.

Parameters Accepting a Single Value From a Set

Adding the suffix **-set** to the basic parameter keyword (**int-set**, **string-set**, **path-set**, and so on) defines a parameter that accepts one of a given list of values.

Figure 63 A parameter requiring one of a set of values



Sets require modifications to the basic parameter syntax (shown in **“Parameter Syntax” on page 261**):

- An additional required keyword, **(set)**, defines the elements of the set.
- Within the **(set)** keyword, **(value)** and **(config-default)** require arguments within parenthesis-bound lists, as in the following:

```
(value (1 2 3))
(config-default (1 2 3))
```

- To prevent a user from adding or removing choices from the list you provide, add the **const** keyword to the “value” declaration:

```
(value const (1 2 3))
(config-default (1 2 3))
```

- To specify an empty set, enter the keyword **none**, as follows:

```
(value none)
(config-default none)
```

Note: *"-set-multi" keywords use a different syntax to define an empty set; see **"Parameters Accepting Multiple Values From a Set"** for more information.*

Other important considerations:

- The value specified as the initial **(value)** for the parameter must match at least one of the values specified for **(config-default)** within the **(set)** keyword.
- The initial value within the **(set)** keyword's **(config-default)** list must be within the **(set)** keyword's **(value)** list. However, we strongly recommend that you make the two lists identical.

Example

To create a parameter that accepts one of a fixed set of integers (like the one shown in Figure 63), enter the following:

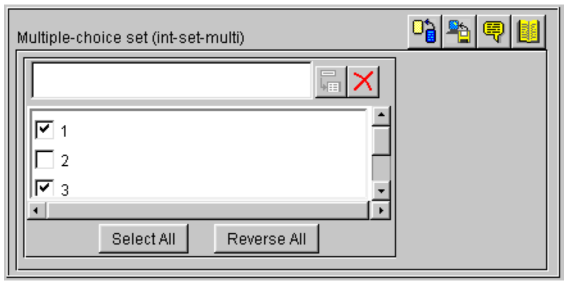
```
(int-set
    (name "Single-choice set (int-set)")
    (value 1)
    (config-default 1)
    (set
      (value const (1 2 3))
      (config-default (1 2 3))
    )
    (description "Provides a single choice from a list of integers.")
    (user-comment
      (value "")
      (config-default "")
    )
  )  ; end of int-set
```

Note: *The values specified by the **(set)** keyword must be within any values specified by the **(range)** keyword. See* **"Specifying Ranges" on page 266** *for more information.*

Parameters Accepting Multiple Values From a Set

Adding the suffix **-set-multi** to the basic parameter keyword (**int-set-multi**, **string-set-multi**, **path-set-multi**, and so on) defines a parameter that accepts one or more options from a given list of values.

Figure 64 A parameter requiring one of a set of values



Sets require modifications to the basic parameter syntax (shown in **“Section Syntax” on page 260**):

- An additional required keyword, **(set)**, defines the elements of the set.
- Within the **(set)** keyword, **(value)** and **(config-default)** require arguments within parentheses-bound lists, as in the following:

```
(value (1 2 3))
(config-default (1 2 3))
```

- To prevent a user from adding or removing choices from the list you provide, add the **const** keyword to the “value” declaration:

```
(value const (1 2 3))
(config-default (1 2 3))
```

- To specify an empty set, enter an empty pair of parentheses “()”, as follows:

```
(value () )
(config-default () )
```

Note: *“-set” keywords use a different syntax to define an empty set; see* **“Parameters Accepting a Single Value From a Set” on page 263** *for more information.*

Other important considerations:

- The value specified as the initial **(value)** for the parameter must match at least one of the values specified for **(config-default)** within the **(set)** keyword.
- The initial value within the **(set)** keyword’s **(config-default)** list must be within the **(set)** keyword’s **(value)** list. However, we strongly recommend that you make the two lists identical.

Example

To create a parameter that accepts one of a fixed set of integers (like the one shown in Figure 64), enter the following:

```
(int-set-multi
    (name "Multiple-choice set (int-set-multi)")
    (value (1 3))
    (config-default (1 3))
    (set
      (value (1 2 3 4 5))
      (config-default (1 2 3 4 5))
    )
    (description "Integer with a modifiable multiple-option set")
    (user-comment
      (value "")
      (config-default "")
    )
  )  ; end of int-set-multi
```

Note: *The order in which keywords appear is very important. See* **"Order of Keywords" on page 261** *for more information.*

A.3.3 Specifying Ranges

The **(range)** keyword enables you to limit the range of options that the user may input as a parameter value for **int** and **char** parameters. You may specify a fixed range, or allow the user to modify the upper limit, the lower limit, or both limits. Range limits are inclusive. The values you specify as limits indicate the lowest or highest acceptable value.

The syntax of (range) is as follows:

```
(range
      (value ([const] lower-limit [const] upper-limit))
      (config-default (lower-limit upper-limit))
    )
```

The optional **const** keyword specifies that the limit is fixed; if the keyword is omitted, the limit modified by the user. The **const** keyword must precede each limit if both limits are to be fixed.

Example

This example illustrates how to define a parameter that accepts an integer as input and limits the range of legal values from zero to ten.

```
(int
    (name "Single integer with fixed range")
    (value 5)
    (config-default 5)
    (range
      (value (const 0 const 10))
      (config-default (0 10))
    )
    (description "Accepts a single integer, limited to a fixed
range.")
    (user-comment
      (value "")
      (config-default "")
    )
  ) ; end of int parameter
```

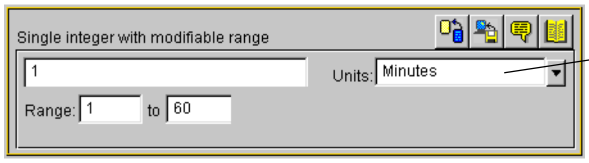
You may also use **(range)** to specify a character range. For example, a range of “A to Z” would limit input to uppercase letters, and a range of “! to ~” limits input to the standard printable ASCII character set (excluding space).

Note: *You may also specify ranges for **-set** and **-set-multi** parameters (**int-set**, **char-set**, and so on).*

A.3.4 Specifying Units

The **(units)** keyword enables **int** parameters to accept input and display the list of available options in different units, provided that each unit is an integer multiple of a base unit.

Figure 65 A parameter that performs unit conversion


Single integer with modifiable range
1
Units: Minutes
Range: 1 to 60

Acceptable groups of units include:

- Seconds, minutes, hours, days
- Bytes, kilobytes, megabytes

Note: *Unit conversions that require floating-point arithmetic are not supported.*

The syntax of the **(units)** keyword is

```
(units
     ("base-unit":1 "first-unit":a "second-unit":b ... "nth-unit":n)
          (value "default-unit")
          (config-default "default-unit")
     )
```

where ***a***, ***b***, and ***n*** are the numbers by which the base unit size should be multiplied to perform the conversion to the respective units. The base unit should normally have a value of 1, as shown above. Although the e*Way Connection Editor will permit other values, it is highly unlikely that an application would require any other number. The units themselves have no meaning to the e*Way Connection Editor other than the relationships you define—in other words, the Editor does not identify or process "seconds" or other common units as such.

Example

To specify a set of time units (seconds, minutes, hours, and days), enter the following:

```
(units
     ("Seconds":1 "Minutes":60 "Hours":3600 "Days":86400)
          (value "Seconds")
          (config-default "Seconds")
     )
```

Units, default values, and ranges

Any time you use the **(units)** keyword within a parameter, be sure that the default values are expressed as integer values of *each* unit. Observing this principle prevents end users from receiving error messages when changing e*Way Connection Editor values in a specific order. For example, if you specified the time units in the example above, but assigned the parameter a default value of "65 seconds," any user who selects the minutes unit *without changing the default value* receives an error message, because the e*Way Connection Editor cannot convert 65 seconds to an integral number of minutes. Ranges are rounded to the nearest integer.

Note: *Regardless of what default value you specify, a user will always see an error message if an inconvertible value is entered and the unit selector is changed. We recommend that you design your parameters so that error messages are not displayed when* ***default*** *values are entered.*

Example

To define a time parameter that displays values in seconds or minutes, with a default of 120 seconds and a fixed range of 60 to 3600 seconds (1 minute to 60 minutes), enter the following:

```
(int
    (name "Single integer with fixed range")
    (value 120)
    (config-default 120)
    (range
      (value (const 60 const 3600))
      (config-default (60 3600))
    )
    (units
      ("Seconds":1 "Minutes":60)
      (value "Seconds")
      (config-default "Seconds")
    )
    (description "Accepts a value between 1 and 60 minutes, with
                 a default units value in seconds.")
    (user-comment
      (value "")
      (config-default "")
    )
  ) ; end parameter
```

Note: *The order in which keywords appear is very important. See* **"Order of Keywords" on page 261** *for more information.*

A.3.5 Displaying Options in ASCII, Octal, Hexadecimal, or Decimal

The **(show-as)** keyword enables you to create **int** or **char** parameters that a user can display in ASCII, octal, hexadecimal, or decimal formats.

The syntax of the **(show-as)** keyword is

```
(show-as
      (format-keyword1 [format-keyword2 ... format-keywordn])
      (value format-keyword)
      (config-default format-keyword)
)
```

where ***format-keyword*** is one of the following:

- ascii
- octal
- hex
- decimal

Format keywords are case-insensitive, and may be used in any combination and in any order.

Be sure that any default values you specify for a parameter that uses **(show-as)** are represented in each of the **(show-as)** formats. For example, if you are using **(show-as)** to show an integer parameter in both decimal and hexadecimal formats, the default value must be non-negative.

Example

To create a parameter that accepts a single character in the character-code range from 32 through 127, and can display the character value in ASCII, hexadecimal, or octal, enter the following:

```
(char
    (name "A single ASCII character")
    (value '\o100')
    (config-default '\o100')
    (range
      (value (const '\o040' const '\o177'))
      (config-default ('\o040' '\o177'))
    )
    (show-as
      (Ascii Octal Hex)
      (value Octal)
      (config-default Octal)
    )
    (description "Accepts one character from ASCII 32 to ASCII 127.")
    (user-comment
      (value "")
      (config-default "")
    )
  ) ; end char parameter
```

Note: *The order in which keywords appear is very important. See* **"Order of Keywords" on page 261** *for more information.*

Factor

The **(factor)** keyword enables users to enter an arithmetic operator (+, –, *, or /) as part of an **int** parameter. For example, to indicate that a value should increase by five units, the user enters the number **5** for single value (int) and the operator **+** for factor.

The syntax of the **(factor)** keyword is

```
(factor
    ('operator1' ['operator2'... 'operatorN'])
    (value 'operator'
    (config-default 'operator')
)
```

where ***operator*** is one of the four arithmetic operators +, –, *, or / (forward slash).

Example

To define a parameter that accepts an integer between 1 and 5 with a factor of + or –, enter the following:

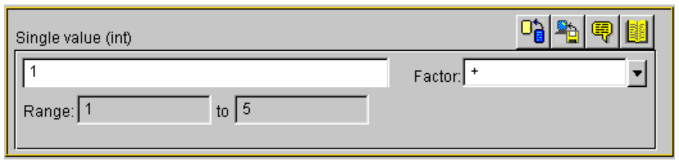
```
(int
    (name "Integer with factor")
    (value 1)
    (config-default 1)
    (range
        (value (const 1 const 5))
        (config-default (1 5))
    )
    (factor
        ('+' '-')
        (value '+')
```

```
                (config-default '+')
            )
            (description "Enter an integer from 1 to 5 and a factor: + or -.")
            (user-comment
                (value "")
                (config-default "")
            )
    )       ; end int parameter
```

Note: *The* ***(factor)*** *keyword must be the final keyword before the* ***(description)*** *keyword. See* **"Order of Keywords" on page 261** *for more information.*

The result seen by the end user would be as in Figure 66.

Figure 66 A parameter using **(factor)**



Encrypting Strings

Encrypted strings (such as for passwords) are stored in string parameters. To specify encryption, use the **encrypt** keyword, as in the following:

```
(string encrypt
    ...additional keywords follow...
```

The e*Way Connection Editor uses the parameter that immediately precedes the encrypted parameter as its encryption key; therefore, be sure that the parameter that prompts for the encrypted data is not the first parameter in a section. The easiest way to accomplish this is to define a "username" parameter that immediately precedes the encrypted "password" parameter. If you need to specify an encryption key other than the user name, you must define a separate parameter for this purpose.

Text entered into an encrypted-string parameter is displayed as asterisks (***).

Example

To create a password parameter, enter the following *immediately following* the parameter definition for the corresponding user name (not shown):

```
(string encrypt
    (name "Password")
    (value "")
    (config-default "")
    (description "The e*Way Connection Editor encrypts this value.")
    (user-comment
      (value "")
      (config-default "")
    )
  )
```

Note: *The **encrypt** keyword can only follow the **string** keyword. The only parameter type that is encrypted is **string**. The **integer**, **character**, **path**, **time**, **date**, or **schedule** parameters cannot be encrypted.*

A.4 Configuration Keyword Reference

Table 20 lists the keywords that may appear in the **.def** file.

Table 20 .def-file keywords

Keyword	Purpose	For more information, see this section
app-protocol	Reserved; do not change from the default "<ANY>".	
cfg-icon	Reserved; do not change from the default "" (null string).	
char	Declares a character parameter.	"Parameter Types" on page 262
char-set	Declares a set of characters, one of which must be selected (via radio button).	"Parameters Accepting a Single Value From a Set" on page 263
char-set-multi	Declares a set of characters, any of which may be selected (via check boxes).	"Parameters Accepting Multiple Values From a Set" on page 264
config-default	Specifies the values that will be restored when the user clicks the e*Way Connection Editor's button.	"Parameter Syntax" on page 261
const	Specifies a value that cannot be changed by the user.	"Specifying Ranges" on page 266
creation	Records the creation date or other information.	"Header Information" on page 259
date	Declares a date parameter.	"Parameter Types" on page 262
date-set	Declares a set of dates, one of which must be selected (through a radio button).	"Parameters Accepting a Single Value From a Set" on page 263
date-set-multi	Declares a set of dates, any of which may be selected (through check boxes).	"Parameters Accepting Multiple Values From a Set" on page 264
delim1	Defines the line-separator delimiter used within .cfg files. It is recommended that you do not modify this value.	

Table 20 .def-file keywords

Keyword	Purpose	For more information, see this section
delim2	Defines the parameter-name delimiter used within .cfg files. It is recommended that you do not modify this value.	
delim3	Defines the value-separating delimiter used within .cfg files. It is recommended that you do not modify this value.	
delim4	Defines the list-item-separating delimiter used within .cfg files. It is recommended that you do not modify this value.	
description	A description for the entry displayed using the e*Way Connection Editor's 💬 button.	"Notes" on page 260
direction	Reserved; do not change from the default "<ANY>".	
encrypt	Encrypts a string, such as for passwords. Valid only after the **string** keyword.	"Encrypting Strings" on page 271
factor	Defines an arithmetic operator to be associated with an integer parameter	"Factor" on page 270
general-info	Defines the "general information" division of the .def file.	"Header Information" on page 259
generated-cfg-path	Specifies the path in which the .cfg file will be stored. It is recommended that you do not modify this field.	
int	Declares an integer parameter.	"Parameter Types" on page 262
int-set	Declares a set of integers, one of which must be selected (through a radio button).	"Parameters Accepting a Single Value From a Set" on page 263
int-set-multi	Declares a set of integers, any of which may be selected (through check boxes).	"Parameters Accepting Multiple Values From a Set" on page 264
modified	Records the modification date or other information.	"Header Information" on page 259
name	Specifies the name of a parameter or a section.	"Parameter Syntax" on page 261
network-protocol	Reserved; do not change from the default "<ANY>".	
os-platform	Reserved; do not change from the default "<ANY>".	
path	Declares a path parameter.	"Parameter Types" on page 262

Table 20 .def-file keywords

Keyword	Purpose	For more information, see this section
path-set	Declares a set of paths, one of which must be selected (through a radio button).	**"Parameters Accepting a Single Value From a Set" on page 263**
path-set-multi	Declares a set of paths, any of which may be selected (through check boxes).	**"Parameters Accepting Multiple Values From a Set" on page 264**
protocol-api-version	Reserved; do not change from the default "<ANY>".	
range	Specifies a range of values that represent the upper and lower limits of acceptable user input.	**"Specifying Ranges" on page 266**
revision	Records revision numbering or other information (entered manually by the developer).	**"Header Information" on page 259**
section	Defines a "section" of the .def file.	See **"Section Syntax" on page 260**
set	Defines the elements in a set.	**"Parameters Accepting a Single Value From a Set" on page 263** and **"Parameters Accepting Multiple Values From a Set" on page 264**
show-as	Selects the format in which character or integer parameters will be displayed.	**"Displaying Options in ASCII, Octal, Hexadecimal, or Decimal" on page 269**
string	Declares a string parameter.	**"Parameter Types" on page 262**
string-set	Declares a set of strings, one of which must be selected (through a radio button).	**"Parameters Accepting a Single Value From a Set" on page 263**
string-set-multi	Declares a set of strings, any of which may be selected (through check boxes).	**"Parameters Accepting Multiple Values From a Set" on page 264**
super-client-type	Reserved; do not change from the default "<ANY>".	
time	Declares a time parameter.	**"Parameter Types" on page 262**
time-set	Declares a set of times, one of which must be selected (through a radio button).	**"Parameters Accepting a Single Value From a Set" on page 263**

Table 20 .def-file keywords

Keyword	Purpose	For more information, see this section
time-set-multi	Declares a set of times, one of which must be selected (through a radio button).	**"Parameters Accepting Multiple Values From a Set" on page 264**
units	Determines in which units a parameter will be displayed.	**"Specifying Units" on page 267**
user	Records the name of the user who last edited the file (entered manually by the developer).	**"Header Information" on page 259**
user-comment	Records a general comment to be applied to the file (accessible through the e*Way Connection Editor).	**"Notes" on page 260**
value	Defines the initial value for a parameter.	**"Parameter Syntax" on page 261**
version	Records the name of the product version.	**"Header Information" on page 259**

A.5 Configuration Parameters and the Configuration Files

Parameters defined within the **.def** file are stored within two "configuration" files (**.cfg** and **.sc**), which are generated by the e*Way Connection Editor's "Save" command. The following rules apply to both **.cfg** and **.sc** files:

- Keywords are not case sensitive, as they are converted to uppercase internally before matching.
- Comments begin with the "#" character, and they must appear in column one. See the example in ".cfg File: conSampleETD.cfg" on page 276.
- Unlike the **.def** file, the **.cfg** and **.sc** files are sensitive to white space. White space consists of single space characters, tabs, and newlines. Be careful not to insert extra white space around delimiters or equal signs. For example: "|value=3|" is legal, but "|value = 3|" and "| value=3 |" are illegal.

The following rule applies only to the **.cfg** file:

- Each line and each element in the **.cfg** file is separated using delimiters (see **delim1, delim2**, **delim3**, and **delim4** starting in Table 20 on page 272). We strongly recommend that you do not modify any of the default delimiters.

Note: *The e*Way Connection Editor creates a* ***.cfg*** *and* ***.sc*** *file automatically when you save your configuration changes in the e*Way Connection Editor. You should not modify either file manually unless directed to do so by Oracle support personnel.*

*Although e*Ways are shipped with default* ***.def*** *files, no configuration files are* *provided, because there is no "standard" configuration for any given e*Way. Users must manually create a configuration profile using the e*Way Connection Editor for every e*Way component.*

Examples

.cfg File: conSampleETD.cfg

This example is excerpted from the "General Settings" section of a **.cfg** file that is generated by the default **sampleETD.def** file.

```
# -------------------------------------------------------------------
#               Delimiters To Use
# -------------------------------------------------------------------
#
File/CFG/Version:0.0/Delim1:\o012/Delim2:\o174/Delim3:\o075/
Delim4:\o054
#
#
# -------------------------------------------------------------------
#                General Info
# -------------------------------------------------------------------
#
#  version:eGate
#  revision:$Revision: 1.1.2.1 $
#  user     :$Author: jdeveloper $
#  modified:$Date: 2002/05/07 02:35:23 $
#  creation:initial
#
#
# -------------------------------------------------------------------
#             e*Way Type
# -------------------------------------------------------------------
#
#  network-protocol:<ANY>
#  os-platform:<ANY>
#  protocol-api-version:<ANY>
#  app-protocol:<ANY>
#  direction:<ANY>
#
#
# -------------------------------------------------------------------
#   Section:Connector
# -------------------------------------------------------------------
#
connector|type|value=SampleETD|set=SampleETD
connector|class|value=sample.SampleETDConnector|set=sample.SampleETDC
onnector
connector|Connection Establishment
Mode|value=Automatic|set=Automatic,Manual,OnDemand
connector|Connection Inactivity Timeout|value=|set=
connector|Connection Verification Interval|value=|set=
#
# -------------------------------------------------------------------
#   Section:External Configuration
# -------------------------------------------------------------------
#
External
Configuration|Filename|value=Output.properties|set=Output.properties
External Configuration|Directory|value=\OutData|set=\OutData
External Configuration|Username|value=admin|set=admin
External Configuration|Password|value=03EB8C38
```

.sc File: conSampleETD.sc

This example is excerpted from the **connector** section of an **.sc** file that is generated by the default **sampleETD.def** file. Notice the amount of additional information as compared to the **.cfg** file example in **".cfg File: conSampleETD.cfg" on page 276**.

```
; -----------------------------------------------------------------
;                General Info
; -----------------------------------------------------------------
(general-info
  (version "eGate")
  (revision "$Revision: 1.1.2.1 $")
  (user "$Author: jdeveloper $")
  (modified "$Date: 2002/05/07 02:35:23 $")
  (creation "initial")
  (description "Sample e*Way:

    High level functionality:

    This is a sample configuration definition template for an
    e*Way connection.
    ")
  (user-comment "")
  (generated-cfg-path "configs/sampleETD/conSampleETD.cfg")
  (delim1 '\n')
  (delim2 '|')
  (delim3 '=')
  (delim4 ',')
  (cfg-icon "")
)
; -----------------------------------------------------------------
;                 e*Way Type
; -----------------------------------------------------------------
(super-client-type
  (network-protocol "<ANY>")
  (os-platform "<ANY>")
  (protocol-api-version "<ANY>")
  (app-protocol "<ANY>")
  (direction "<ANY>")
)
; -----------------------------------------------------------------
;   Section:"Connector"
; -----------------------------------------------------------------
(section
  (name "connector")
  (string-set
    (name "type")
    (value "SampleETD")
    (config-default "sampleETD")
    (set
      (value ("SampleETD"))
      (config-default ("sampleETD"))
    )
    (description "Connector type :
 The value is always defaulted to sampleETD for sampleETD connection
")
    (user-comment
      (value "")
      (config-default "")
    )
  )
  (string-set
    (name "class")
    (value "sample.SampleETDConnector")
    (config-default "sampleETD.SampleETDConnector")
    (set
      (value ("sample.SampleETDConnector"))
      (config-default ("sampleETD.SampleETDConnector"))
    )
    (description "Connector class:
 This parameter specifies the class name of the ETD connector object.
")
    (user-comment
      (value "")
      (config-default "")
    )
  )
  (string-set
    (name "Connection Establishment Mode")
    (value "Automatic")
    (config-default "Automatic")
    (set
      (value const ("Automatic" "Manual" "OnDemand"))
      (config-default ("Automatic" "Manual" "OnDemand"))
    )
    (description "Connection Establishment Mode:

 This parameter specifies how connection with the external
 system is established and closed.  Automatic indicates that
 the connection is automatically established when the
 collaboration is started, and it keeps the connection alive
 as needed. OnDemand indicates that the connection will be
 established on demand, as business rules requiring a
 connection to the external system are performed.  The
 connection will be closed after the methods are completed.
 Manual indicates that the user will explicitly call the
 connection connect and disconnect methods in their
 collaboration as business rules.

 Default is Automatic.

 If running 4.5.1 or earlier version of e*Gate core, this
 option is ignored.
")
    (user-comment
      (value "")
      (config-default "")
    )
  )
  (string-set
    (name "Connection Inactivity Timeout")
    (value none)
    (config-default none)
    (set
      (value ())
      (config-default ())
    )
    (description "Connection Inactivity Timeout:

 This value is used to specify timeout (in milliseconds) for
 the Automatic connection establishment mode.  If this is
 not set or if it is set to 0, the connection will not be
 brought down due to inactivity.  The connection is always
 kept alive; if it goes down, re-establishing connection
 will automatically be attempted.  If a non-zero value is
 specified, the connection manager will try to monitor for
 inactivity so the connection is brought down if the value
 specified is reached.

 If running 4.5.1 or earlier version of e*Gate core, this
 option is ignored.
")
    (user-comment
      (value "")
      (config-default "")
    )
  )
  (string-set
    (name "Connection Verification Interval")
    (value none)
    (config-default none)
    (set
      (value ())
      (config-default ())
    )
    (description "Connection Verification Interval:

 This value is used to specify the minimum period of time
 (milliseconds) between checks for connection status to
 the database server.  If the connection to the server is
 detected to be down during verification, the user
 collaboration's onConnectionDown method is called.  If
 the connection comes from a previous connection error,
 the user collaboration's onConnectionUp method is called.
 If no value is specified, 60000 ms is used.

 If running 4.5.1 or earlier version of e*Gate core, this
 option is ignored.
")
    (user-comment
      (value "")
      (config-default "")
    )
  )
  (description "Connector:

 This section contains a set of top level parameters:

        o type
        o class
        o Connection Establishment Mode
        o Connection Inactivity Timeout
        o Connection Verification Interval
")
  (user-comment
    (value "")
    (config-default "")
  )
)
; ------------------------------------------------------------------
;   Section:"External Configuration"
; ------------------------------------------------------------------
(section
  (name "External Configuration")
  (string-set
    (name "Filename")
    (value "Output.properties")
    (config-default none)
    (set
      (value ("Output.properties"))
```

```
        (config-default ())
      )
      (description "Filename:

 Mandatory.
 This is the filename ...
")
      (user-comment
        (value "")
        (config-default "")
      )
    )
    (string-set
      (name "Directory")
      (value "\OutData")
      (config-default none)
      (set
        (value ("\OutData"))
        (config-default ())
      )
      (description "Directory:

 Mandatory.
 This is the directory ...
")
      (user-comment
        (value "")
        (config-default "")
      )
    )
    (string-set
      (name "Username")
      (value "admin")
      (config-default none)
      (set
        (value ("admin"))
        (config-default ())
      )
      (description "Username:

 Mandatory.
 This is a sample for a config variable for specifying the username
for connecting to an external server.
")
      (user-comment
        (value "")
        (config-default "")
      )
    )
    (string encrypt
      (name "Password")
      (value "03EB8C38")
      (config-default "")
      (description "Password:

 Mandatory.
 This is a sample for a config variable for specifying the password
for connecting to an external server.
")
      (user-comment
        (value "")
        (config-default "")
      )
    )
```

```
  (description "Connection:

 This section contains information for connecting to Portal Infranet:

        o Filename
        o Directory
        o Username
        o Password
")
  (user-comment
    (value "")
    (config-default "")
  )
)
```

A.6 Testing and Debugging the .def File

To test the **.def** file, open it with the e*Way Connection Editor. If the syntax of all parameters is correct, the e*Way Connection Editor will launch, and you can confirm that your sections, parameters, ranges, and options are as you intended.

There are two types of errors that you may encounter:

- Logical errors: The e*Way Connection Editor loads the **.def** file and displays no error message, but the parameters are not defined as desired (for example, default options are omitted, or a range was not properly defined). These errors are corrected by replacing the incorrect values with the correct ones.
- Syntax errors: These "mechanical" errors involve missing parentheses, invalid keywords and similar problems. These errors cause the e*Way Connection Editor to display an error message and exit. This section primarily addresses errors of this type.

Note: *You may also encounter syntax errors if you try to edit an existing configuration profile that contains a corrupted **.sc** file. You should not attempt to modify **.sc** or **.cfg** files outside of the e*Way Connection Editor unless specifically instructed to do so by Oracle personnel.*

The e*Way Connection Editor component that interprets the **.def** file provides only elementary error messages when it encounters an error. This section describes the most common errors you may encounter, and the steps you should take to debug a **.def** file under development.

By far, the most common errors are:

- Missing parentheses. Proper indentation will help you catch most of these, and some editors have features that find matching parentheses (such as the **vi** editor's SHIFT+% function).
- Missing quotation marks. Be sure that characters are delimited by single quotes and strings (including path names) by double quotes.

- Quotation marks that should be escaped but are not. This usually occurs in the argument to the **(description)** keyword; double-check that all quotations within descriptions use \"escaped\" quotation marks.
- Missing parameters. Refer to the examples in this chapter, or to the sample **.def** file for the required parameters for each keyword.
- Keywords out of order. See **"Order of Keywords" on page 261**.

Note: *Using the templates provided in the sample* ***.def*** *file will help prevent many errors before they occur; see* **"Sample .def File" on page 283** *for more information.*

A.6.1 Common Error Messages

The following section contains common error messages and their most common causes. Each error message will contain the string **L<*nnn*>**, which indicates a line number (for example, **L<124>** signifies "line 124").

SCparse : parse error, expecting `LP_*keyword-name*'
The keyword ***keyword-name*** was expected but not found. The keyword could be missing or out of order, the keyword's initial parenthesis could be missing, or the previous keyword could have been terminated prematurely (for example, by an out-of-place parenthesis or quote-parenthesis combination) or misspelled.

SCparse : parse error, expecting `RIGHT_PAREN'
The right parenthesis is missing, a close-quote is missing, as in **(user-comment "   )**, or there is an extra or unescaped close-quote within a **(description)** keyword argument.

SCparse : parse error, expecting `LEFT_PAREN'
This error appears under a very wide range of conditions. A keyword could be misspelled, there could be extraneous or unbalanced quotes or parentheses, a keyword could be missing a left parenthesis, or extraneous material may have been found between parameter declarations. Sometimes this error appears in conjunction with **expecting `LP_*keyword-name*'**.

***Param-Type<keyword>*: Value is not within the allowed range.**
An argument to a keyword has exceeded the limits defined by its accompanying **(range)** keyword. Change either the **(value)** argument or the **(range)** limit.

***param-type*TypeSet<*keyword*> : "*n*" is not in this Set.**
A default value for a parameter has been specified that does not appear within the default value of the **(set)** keyword.

SCparse : parse error, expecting `*arg-type*'
One type of argument was expected, but another has been found (for example, an integer where as string was expected). Errors expecting LITERAL_STRING are commonly caused by missing quotation marks. Errors expecting TIME_VAL, DATE_VAL, or SCHEDULE_VAL can also be due to invalid data (such as a time of 12:00:99), or missing/extra delimiters.

CharVal : "\*sequence*" is not legal character.
There is an error in an escape sequence.

SCparse : parse error
This “general” error is caused by a number of problems, such as misspelled arguments within keywords.

A.7 Sample .def File

A **.def** file containing commented samples for sample parameter definitions is available on the e*Gate Integrator Installation CD-ROM. When you extracted it from **gmeek.taz**, it was installed to:

```
gmeek\installETD\SampleETD\sampleETD.def
```

You can use this **sampleETD.def** file as a template from which to build your own extensions to a **.def** file that you create. Open the file with a text editor, select the desired parameter-definition template, and then copy and paste the template into your own **.def** file, where you can modify it as needed.

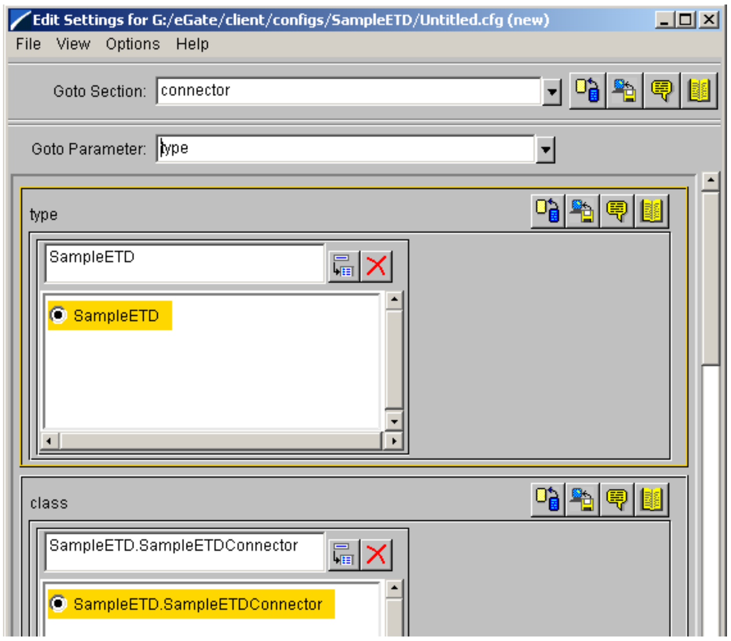
When you finish editing the **.def** file, you must then commit it to the Registry. For instructions on committing the **.def** file to the Registry, see **“Installing the Sample Files to e*Gate” on page 179**.

To open the sample.def file in the e*Way Connection Editor

1. Use the **installETD** scripts to install the sampleETD files to a schema. For more information, see **“Running the installETD Script” on page 75**.
2. Create or select an e*Way Connection, and then display its properties.
3. From the drop-down menu, select **SampleETD**.
4. Under **Configuration file**, click **New**.

   The e*Way Connection Editor displays several sections of sample parameters—“type,” “class,” and so on—as shown in Figure 67.

Figure 67 The **sampleETD.def** file in the e*Way Connection Editor


Edit Settings for G:/eGate/client/configs/SampleETD/Untitled.cfg (new)
File View Options Help
Goto Section:
Goto Parameter:

Goto Section: connector

Goto Parameter: type

type

SampleETD

SampleETD

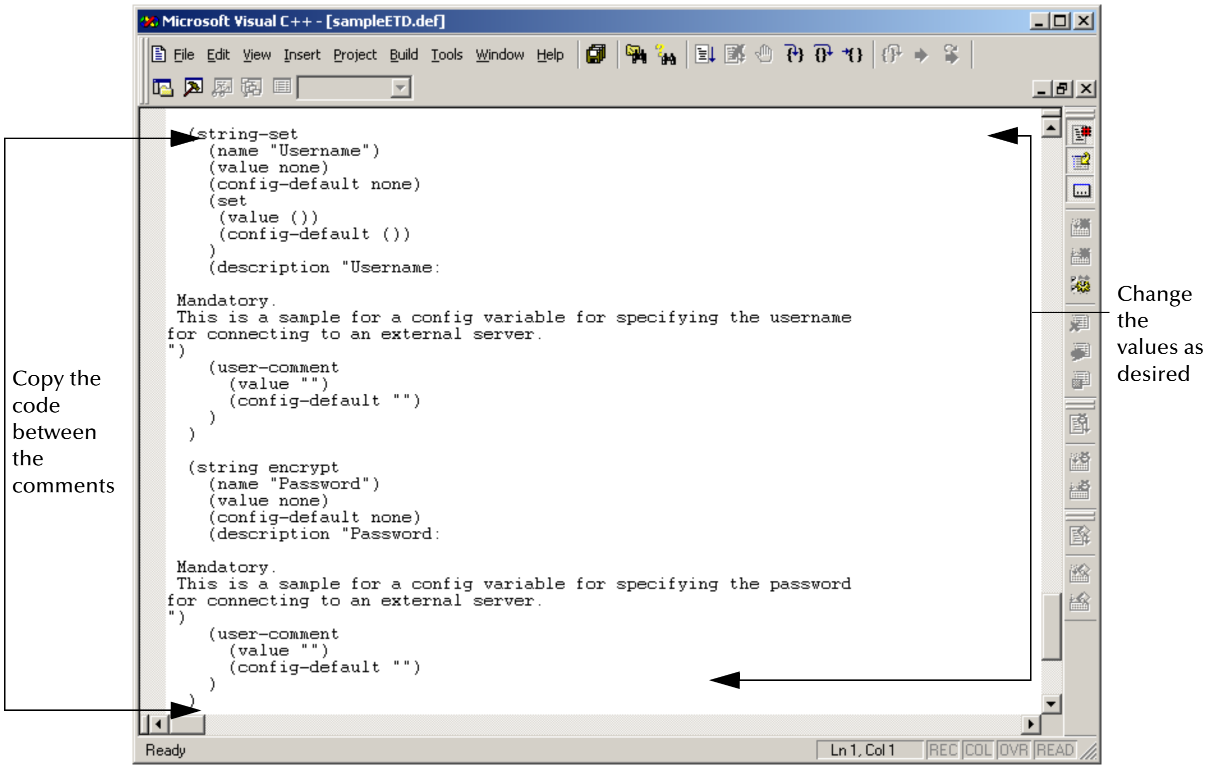
class

SampleETD.SampleETDConnector

SampleETD.SampleETDConnector

After identifying the parameter you wish to copy, open **sampleETD.def** in a text editor and search for the parameter name. Then, copy the parameter and change the sample values to the values you want to use. See Figure 68.

Figure 68 The **sampleETD.def** File in Text Editor


Microsoft Visual C++ - [sampleETD.def]
File Edit View Insert Project Build Tools Window Help
Ready
Ln 1, Col 1
REC COL OVR READ
Copy the code between the comments
Change the values as desired

```
  (string-set
    (name "Username")
    (value none)
    (config-default none)
    (set
     (value ())
     (config-default ())
    )
    (description "Username:

 Mandatory.
 This is a sample for a config variable for specifying the username
for connecting to an external server.
")
    (user-comment
      (value "")
      (config-default "")
    )
  )

  (string encrypt
    (name "Password")
    (value none)
    (config-default none)
    (description "Password:

 Mandatory.
 This is a sample for a config variable for specifying the password
for connecting to an external server.
")
    (user-comment
      (value "")
      (config-default "")
    )
  )
```

A.8 Accessing Configuration Parameters Within the APIs

The e*Way Connection automatically loads configuration parameters stored in the **.cfg** file into variables within the APIs.

A.8.1 Format for Variable Names

We recommend that you name variables using the format

SECTION-NAME_PARAM-NAME

where ***SECTION-NAME*** is the name of the section and ***PARAM-NAME*** is the name of the parameter. The value of the parameter is referenced as the value of the variable.

Convention

Variable names should be in all upper case. The section and parameter names should be joined by an underscore, and any spaces contained within section or parameter names should also be converted into underscores.

We recommend using this variable name as a property name to consistently access the value of the .cfg parameter throughout your ETD code. See **"Getting Variable Values" on page 286**.

Examples

The value of the parameter named "Directory" within the section "External Configuration" should be declared as the variable "EXTERNAL_CONFIGURATION_DIRECTORY" (all upper case).

The value of the parameter named "Gateway ID" within the section "External Configuration" should be declared as the variable "EXTERNAL_CONFIGURATION_GATEWAY_ID".

A.8.2 Referencing the Parameter

If you have a space between the section name and the parameter name, you must replace this space with an underscore. As a result, this allows you to reference the parameter correctly.

To distinguish between the section name and the parameter, use a period "**.**", as shown in the following example.

Examples

`"External_Configuration.Directory"`

where

`External_Configuration`

is the section name, and

`Directory`

is the parameter name.

A.8.3 Getting Variable Values

Variable values are read using the helper function **getProperty()**.

The **getProperty()** method retrieves the configuration parameters that the e*Way extracted form the corresponding configuration file.

Examples

```
public static final String
EXTERNAL_CONFIGURATION_DIRECTORY="External_Configuration.Directory";
this.cfgProps = myETDConnector.getProperties();

String propsDirectory =
cfgProps.getProperty(EXTERNAL_CONFIGURATION_DIRECTORY);
```

Appendix B

The XSC Format

This appendix provides detailed information on the following aspects of the XSC format:

- **Entities** on page 289
- **Table of XSC Entities and Their Attributes** on page 300
- **Method Signature Syntax** on page 304
- **Identifier Characters** on page 305
- **JCS Properties** on page 305

This is a *prescriptive* guide to the recommended usage of the XSC format. It is not intended to be exhaustive; in other words, there are additional *supported* entities, attributes, and constructions that are permitted but not recommended (for example, to provide backward compatibility with XSC 0.4 and XSC 0.3).

The XSC 0.6 format was introduced at e*Gate version 4.5.2 (superseding the XSC 0.4 format of e*Gate version 4.5.1), and XSC 0.6 is recommended. However, you can rely on this information even if you are developing XSC 0.4 files for use in e*Gate version 4.5.1, as all usages that are incompatible with XSC 0.4 are flagged as such throughout this appendix.

B.1 Overview

All **.xsc** files, regardless of the XSC format version level, consist of a single <etd> entity and the entities it contains (which must include exactly one <javaProps> entity and at least one <node> or <class> entity).

For an **.xsc** file to be viewed in the e*Gate GUIs, every entity must have a **uid** attribute, and no two **uid** attributes in the same file can have the same value.

Every entity can have an optional **comment** attribute, a string of up to 1024 characters that is used for documentation purposes.

Example of Required Entities

Syntax

Note: *Curly braces {...} indicate a* ***possibly repeating*** *sequence (zero or more times). Square brackets [...] indicate an* ***optional*** *sequence (exactly zero times or once). Parentheses (...) indicate a sequence that is grouped to make it more legible.*

```
<etd [name=root-name]
        xscVersion=”0.6” type=etd-type
        [editable=boolean] [derived=boolean]
        [sscEncoding=string] [dataEncoding=string]
        uid=text [comment=string]>

  <javaProps package=text [class=text]
             [codeAvailable=boolean] [jarFile=filepath]
             [source=filepath]
             uid=text [comment=string]>
    [<jar file=filepath uid=text [comment=string]>]
    {<interface fqClass=text uid=text [comment=string]>
       <method ...> {<method ...>} } }

  [<delimiters uid=text [comment=string]>
     <delim [endOfRec=boolean] [separator=boolean]
            [required=boolean] [array=boolean]
            [anchored=boolean]
                 [beginAnchored=boolean] [endAnchored=boolean]
            uid=text [comment=string]>
       <delimGroup>
         <beginDelim>
         <endDelim>
    {<delim ...> } ]

  <node type=“CLASS”
        name=text [javaName=text]
        [javaType=datatype]
        [public=boolean]
        [... other optional attributes ...]
        uid=text [comment=string]>
    <node type=( “CLASS” | “FIELD” | ”ENUMERATION” | ”REFERENCE”)
          name=text [javaName=text]
          [readOnly=boolean]
          [minOccurs=numZPU] [maxOccurs=numUZPU] [optional=boolean]
          [ ( fixedValue=string |
              defaultValue=string [defaultBytes=string
                                  [defaultEncoding=name] ] )
          [inputMatch=boolean [avoidMatch=boolean] ]
          [exact=boolean]
          [group=boolean]
          [ ( length=“DECIMAL” lengthFrom=numZP lengthSize= numP |
              length=numNZPU
                [lengthFrom=”undefined”] [lengthSize=”undefined”] ) ]
          [offset=numZPU]
          [... other optional attributes specific to delimiters...]
          [structure=( “delim” | “fixed” | “array” | “set” )]
          [order=( “sequence” | “any” | “choice” )]
          [childMin=numZPU] [childMax=numUPU] ]
          uid=text [comment=string]>

  {<node type=( “CLASS” | “FIELD” | ”ENUMERATION” | ”REFERENCE”) ...>
    {<delim ...>
    {<method name=text [signature=text] returnType=datatype >
```

```
             uid=text [comment=string]>
        {<param name=text paramType=datatype
               uid=text [comment=string]> }
        {<throws> excepType=datatype uid=text [comment=string]> } }
     {<node ...>
        {<delim> ...} {<method> ...} {<node> ... } }

   {<node ...>
     {<delim> ...} {<method> ...} {<node> ... } }
```

B.2 Entities

Every **.xsc** file you create must be a valid XML string composed of a valid combination of the following types of entities.

Required entities

- <etd>
- <javaProps> (This is required in XSC 0.6, and highly recommended in XSC 0.4.)
- <node> (or <class>)

Optional entities for non-.ssc-based ETDs

- <jar>
- <interface>
- <method>
- <param>

.ssc-specific entities

- <delimiters>
- <delim>
- <delimGroup>
- <beginDelim> and <endDelim>

B.2.1 The <etd> Entity

The <etd> entity is the top-level entity in an **.xsc** file. It represents a complete ETD (or a self-contained part thereof, such as a reusable template). This entity is required and non-repeating; in other words, every **.xsc** file must contain exactly one <etd> entity.

Syntax

```
<etd [name=root-name]
        xscVersion="0.6" type=etd-type
        [editable=boolean] [derived=boolean]
        [sscEncoding=string [dataEncoding=string]]
        uid=text [comment=string]>
```

```
    <javaProps ...>

    [<delimiters ...>]

    <node name=root-name ...>
    {<node ...>}

  </etd>
```

Required Attributes for <etd>

name=*root-name* (required for type="SSC" only)

In XSC 0.6, the **name** attribute is required when type="SSC" but optional for other ETD types; see **"Optional Attributes for <etd>"**.

xscVersion="0.6"

Default value: (not applicable; in XSC 0.6, this must be set to "0.6")

Identifies the XSC syntax version. This attributes serves as a way to provide backward compatibility in later XSC versions, so that a reader can identify and process XSC input depending on the version.

type=*etd-type*

Default value: (undefined)

Identifies the message type and, therefore what kind of parser and rendering function will be used to convert between external and internal representations of the message.

Examples of *etd-type* include:

- "DB"—generated by the database builder from an SQL-accessible relational database schema
- "DTD"—XML description, generated from a DTD
- "IDOC"—IDOC document description in SAP's IDOC meta-language
- "SEF"—Standard Exchange Format (Foresight Corporation)
- "SSC"—converted from Monk SSC or written from scratch using the ETD editor
- "X12"—from the EDI format
- "XSD"—XML description, written in the XML Schema notation

Optional Attributes for <etd>

name=*root-name*

Default value: (undefined)

The **name** attribute of the <etd> entity, if specified, must match the name attribute of one of the <node> entities contained immediately below the <etd> entity. This identifies that <node> entity as the top node to show in the GUI.

Note: *For **type**="SSC", the **name** attribute is mandatory.*

editable=*boolean*

Default value: “false”

Indicates whether the ETD Editor is used to modify the contents. You should set this to “true” only if the ETD can invoke a back-end to produce code based solely on the ETD contents. I

In e*Gate version 4.5.2, this can occur only when type=“SSC” or when derived=“true”.

derived=*boolean*

Default value: “false”

When set to “true”, the XSC represents a derived ETD.

comment=*string*

Default value: (undefined)

An arbitrary string for documentation purposes. Attribute values must be in normal-safe form, which protects them against alteration by attribute normalization.

sscEncoding=*string*

Note: *This attribute of the <etd> entity is valid only when type="SSC"*

Default value: “US-ASCII”

Indicates the preferred encoding of the output **.ssc** file. Supported values in XSC 0.6 are:

- “US-ASCII” (corresponds to “ASCII” in Monk)
- “UTF-8” (corresponds to “UTF8” in Monk)
- “SJIS” (a common Japanese character encoding)
- “MS949” (corresponds to “UHC” in Monk)

dataEncoding=*string*

Note: *This attribute of the <etd> entity is valid only when type="SSC"*

Default value: (the value of the **sscEncoding** attribute)

Contains the “assumed input encoding” to be passed to the Monk Event parser at run time. The parser, which is otherwise oriented towards byte-stream-processing, uses the assumed input encoding to avoid inappropriate delimiter matching (misidentifying the second or further byte of a multi-byte sequence encoding a single input character with the first byte of a delimiter).

In Monk, the assumed input encoding is derived implicitly from the encoding of the metadata itself in the **.ssc** file. Because the output SSC in e*Gate is normally rewritten to UTF-8, the original input encoding information is preserved in the **dataEncoding** attribute. In e*Gate version 4.5.2, for example, the SSC Builder can handle **.ssc** files coded in ASCII, ISO-8859-1, UTF-8, Shift-JIS (Japanese), UHC (Korean), and Big5 (Traditional Chinese).

Entities Directly Contained by <etd>

The optional <delimiters> entity occurs only in <etd> entities for which type="SSC". The required <javaProps> entity contains data on the Java implementation of the XSC. The required <class> and/or <node> entity or entities set up the structure of the data described in the XSC.

Compatibility Notes for <etd>

Attributes

Prior to XSC 0.5, the **name** attribute was mandatory for all ETD types. The lowercase variants of the **type** attribute values were deprecated but accepted in XSC 0.2; they were dropped in XSC 0.3 onwards. The **xscVersion** attribute was introduced in XSC 0.4 (with a value of "0.4"). The **sscEncoding** and **dataEncoding** attributes were introduced in XSC 0.4. The **derived** attribute was introduced in XSC 0.6.

Entities directly contained

XSC 0.3 and 0.4 permitted an <extra> entity for extension purposes, but it was never used.

B.2.2 The <javaProps> Entity

The <javaProps> entity contains information regarding the generated Java code for the ETD, and can only occur directly below the <etd> entity. It is mandatory in XSC 0.6 and highly recommended in XSC 0.4. It is not repeatable.

Syntax

```
<javaProps package=text class=text
    [codeAvailable=boolean] [jarFile=path] [source=path]
    uid=text [comment=string]>
  [<jar>]
  {<interface>}
</javaProps>
```

Required Attributes for <javaProps>

package=*text*

Default value: (undefined)

class=*text*

Default value: (ETD name)

Together, the **package** and **class** attributes specify the fully qualified Java class name of the Java class that implements the root node of the XSC. The Java file name and location are inferred from the class name and package name. For example, if class="Z" and package="x.y", the file name is "x/y/Z.java". The (unqualified) class name defaults to the ETD name; this is analogous to the **javaName** attribute for <class> and <node> entities.

The full package name consists of zero or more dot-separated components, where each component is both a directory name (below the e*Gate Java package root directory) and a valid Java ID. In order to support platforms such as Windows, on which directory names must differ by more than just case, the components must all be:

- case-insensitive
- portable directory names: The POSIX portable character set comprises ASCII letters, digits, hyphen (-), underscore (_), dot (.), tilde (~), and hash (#).
- valid Java identifiers (see section B.5)

Omitting uppercase characters (to preserve case-insensitivity and Java package name convention), and omitting characters not valid in a Java ID, the component syntax is a non-empty sequence of letters, digits and underscores, not starting with a digit, and (for legibility) prohibiting consecutive underscores, as shown in the following examples:

- package="abc.xyz._123" (three components)
- package="my.company-name.com.~user-id.#01_A001" (five components)

This enforces a one-to-one correspondence between Java package names and directory paths and removes the need for managing distinct entities in parallel. For convenience, to avoid allowing tools do the conversion themselves, the explicit directory path is a derived string.

Optional Attributes for <javaProps>

codeAvailable=*boolean*

Default value: "false"

Indicates whether the associated code has been successfully produced and compiled. If **codeAvailable** is unspecified or set to "false", then the **jarFile** attribute is optional.

jarFile=*path*

Default value: (undefined)

Defines the path (relative to the e*Gate home directory) to the **.jar** file. This attribute is required if codeAvailable="true", but is otherwise optional.

source=*path* (valid for derived ETDs only)

Default value: (undefined)

Specifies the path (relative to the e*Gate home) to the associated Java source file with the wrapper code. This attribute is only valid in a <javaProps> entity in a derived ETD. (A derived ETD is an <etd> entity for which derived="true".)

Entities Directly Contained by <javaProps>

The optional <jar> entities are valid only for derived ETDs. The optional <interface> entities are valid for any ETD.

Compatibility Notes for <javaProps>

The <interface> entity was introduced at XSC 0.3. The <jar> entity was introduced at XSC 0.6.

B.2.3 The <jar> Entity

The <jar> entity can only occur directly below a <javaProps> entity in a derived ETD. It is optional and, for XSC 0.6 and earlier, not repeatable. It specifies the **.jar** file that implements the parent ETD's root class.

At run time, the Collaboration Editor checks to be sure that the **.jar** file mentioned in each <jar> entity is available and in the run-time classpath.

Syntax

```
<jar file=path uid=text [comment=string]>
</jar>
```

B.2.4 The <interface> Entity

The <interface> entity can only occur directly below a <javaProps> entity. It is optional and repeatable. In e*Gate, any Collaboration that uses the XSC as an ETD includes the declaration that its class implements the given interface or interfaces.

Note: *At XSC 0.6, the <interface> entity does not support interface elements besides methods. For example, static final variables are not supported.*

Syntax

```
<interface fqClass=text uid=text [comment=string]>
  {<method>}
</interface>
```

Required Attribute for <interface>

fqClass=*text*

Each fqClass value must be unique within the ETD, and must be a fully qualified Java interface name.

Entities Directly Contained by <interface>

The <interface> entity can optionally contain one or more <method> entities. In XSC 0.6, it cannot contain any other types of entities.

Compatibility Notes for <interface>

This entity was introduced at XSC 0.3. Its **uid** attribute was introduced at XSC 0.5.

B.2.5 Delimiter-Related Entities (SSC only)

The following describes four delimiter-related entities:

- The **<delimiters>** entity is optional and not repeatable. It can only occur directly below the <etd> entity in an SSC-based ETD. (An SSC-based ETD is an <etd> entity for which type="SSC".) It corresponds to the Global Delimiters list in an **.ssc** file, and like the **.ssc** file it contains a sequence of zero or more delimiters in descending order of application.
- The **<delim>** entity is optional and repeatable. It can only occur directly below the <delimiters> entity. Each instance of a <delim> entity specifies the next lower level of global delimiters, either by specifying a pair of strings in its **beginDelim** and **endDelim** attributes, or via a <delimGroup> entity that specifies a set of candidate begin-delimiter strings and a paired set of candidate end-delimiter strings.
- The **<delimGroup>** entity is optional and repeatable. It can only occur directly below the <delim> entity, which specifies the level for which this group applies. Each instance of a <delimGroup> entity specifies a set of candidate begin-delimiter strings and a paired set of candidate end-delimiter strings for the current level.
- The **<beginDelim>** (and **<endDelim>**) entities are optional and repeatable. They can only occur directly below the <delimGroup> entity. Each such entity specifies a set of candidate delimiter characters or strings, any one of which is to be recognized as a begin-delimiter (or end-delimiter) for that level.

Syntax for <delimiters> entity

```
<delimiters uid=text [comment=string]>
  {<delim>}
</delimiters>
```

Syntax for <delim> entity

```
<delim [beginDelim=string] [endDelim=string] [value=string]
    endOfRec=boolean separator=boolean required=boolean
    array=boolean anchored=boolean
    beginAnchored=boolean endAnchored=boolean
    uid=text [comment=string]>
</delim>
```

Syntax for <delimGroup> entity

```
<delimGroup {<beginDelim>} {<endDelim>}
    uid=text [comment=string]>
</delimGroup>
```

Syntax for <beginDelim> entity

```
<beginDelim ( string-delim | encoded-delim)
    uid=text [comment=string]>
</beginDelim>
```

Syntax for <endDelim> entity

```
<endDelim ( string-delim | encoded-delim)
    uid=text [comment=string]>
</endDelim>
```

Required Attributes for <delim>

endOfRec=*boolean*

Default value: "false"

separator=*boolean*

Default value: "false"

required=*boolean*

Default value: "false"

array=*boolean*

Default value: "false"

anchored=*boolean*

Default value: "false"

beginAnchored=*boolean*

Default value: "false"

endAnchored=*boolean*

Default value: "false"

Optional Attributes for <delim>

beginDelim=*string*

Default value: (undefined)

Specifies the value for the begin-delimiter at this level.

endDelim=*string*

Default value: (undefined)

Specifies the value for the end-delimiter at this level.

Entities Directly Contained by <delim>

One or more optional <delimGroup> entities occur only in <delim> entities that lack **beginDelim** and **endDelim** attributes.

Required and Optional Attributes for <delimGroup>

Other than the required **uid** attribute and the optional **comment** attribute, the <delimGroup> entity takes no attributes.

Entities Directly Contained by <delimGroup>

One or more optional <beginDelim> and <endDelim> entities are specified within each <delimGroup> entity.

Attributes for <beginDelim> and <endDelim>

Each <beginDelim> and <endDelim> entity must have exactly one **string-delim** set of attributes or else exactly one **encoded-delim** set of attribute.

B.2.6 The <node> and <class> Entities

Note: *The <node> and <class> entities are synonyms. The current SSC Builder uses <class> for template roots and <node> for nested entities, but this is only a convention. We recommend that you use <node> exclusively.*

A <node> entity describes a single node in the ETD—in other words, a simple data field or a delineated group of data fields. It can occur optionally or repeatedly in actual instances of the Event.

Syntax

```
<node type=( “CLASS” | “FIELD” | ”ENUMERATION” | ”REFERENCE”)
    name=text [javaName=text]
    [readOnly=boolean]
    [minOccurs=numZPU] [maxOccurs=numUZPU] [optional=boolean]
    [ ( fixedValue=string |
        defaultValue=string [defaultBytes=string
                            [defaultEncoding=name] ] )
    [inputMatch=boolean [avoidMatch=boolean] ]
    [exact=boolean]
    [group=boolean]
    [ ( length=“DECIMAL” lengthFrom=numZP lengthSize= numP |
        length=numNZPU
          [lengthFrom=”undefined”] [lengthSize=”undefined”] ) ]
    [offset=numZPU]
    [... other optional attributes specific to delimiters...]
    [structure=( “delim” | “fixed” | “array” | “set” )]
    [order=( “sequence” | “any” | “choice” )]
    [childMin=numZPU] [childMax=numUPU] ]
    uid=text [comment=string]>
  {<node ...>}
  {<method ...>}
</node>
```

Syntax for <node> Entities of type=”CLASS”

```
<node type=“CLASS”
      name=text [javaName=text]
      [javaType=datatype]
      [public=boolean]
      [... other optional attributes ...]
      uid=text [comment=string]>
  {<node ...>}
  {<method ...>}
</node>
```

Required Attributes for <node>

name=*text*

Default value: (undefined)

type

Each node must be of one of the following types:

- A *template* node has **type**=”CLASS” and can have a **public**=*boolean* attribute. It is characterized by being a top-level node. The parent of a template node is the <etd> entity. The sequence of <node> entities immediately inside the <etd> entity is known as its *local template list*. Each local template must have a name that is unique in the list (no two top-level nodes may have the same name). If the <etd> entity has a **name** attribute, its value must match the name attribute of a template node.
- A *composite* node also has **type**=”CLASS”, but lacks a **public** attribute. A node that is a parent element (as opposed to a leaf) is composite if it is not a template.

  For both composite nodes and template nodes, the **javaType** attribute value defaults to the fully qualified Java class name formed by the <javaProps> package value, followed by all ancestor node names and the name of the node itself, separated by ".". Node names in this case are the <node> **javaName** values if present, and *name* values otherwise.
- A *simple* node has **type**=”FIELD”. Simple nodes describe data fields that are not further subdivided or described elsewhere in the ETD. The **javaType** attribute value defaults to “java.lang.String”. In this case, the **encoding** attribute can specify the character encoding name used to convert between raw input/output byte data and internal string values.
- An *enumeration* node has **type**=”ENUMERATION”, and is also a leaf node. Nodes of this type require a list of zero or more members that represent enumeration elements. Enumeration elements are <member> entities with allowable attributes **name** and **value**. If any of the members of an enumeration has the **value** attribute, then all its member must have it. In this case, all values should be distinct strings.
- A *reference* node has **type**=”REFERENCE”, and is a surrogate for an ETD part that is defined either within the same **.xsc** file (in which case the node is called an *internal reference* to a *local template*), or else to a global template defined in another **.xsc** file (in which case the node is called an *external template*). The distinction is signaled by the **reference** attribute. When **reference**=*filepath* is defined, *filepath* is the relative path and filename of the external **.xsc** file being referenced.

B.2.7 The <method> Entity

A <method> entity describes an explicit public method associated with a particular generated class (if the parent of the <method> is a <node> entity) or implemented interface (if the parent is an <interface> entity) in the ETD.

Implicit methods, on the other hand, are generated automatically for each node depending on its attributes. Thus, you need not create Bean-style getter/setter methods for each node unless you want to override them. Examples include:

- **get**<*nodeName*>**()** is always generated.
- **set**<*nodeName*>**()** is generated for writable nodes—in other words, nodes for which **readOnly**=”false”.
- **has**<*nodeName*>**()** is generated for nodes that might not receive data— in other words, nodes for which **minOccurs**=”0”.
- **count**<*nodeName*>**()** is generated for repeating nodes— in other words, nodes for which **maxOccurs** is greater than ”1” or is equal to “unbounded”.

Syntax

```
<method name=text [type=”METHOD”
    signature=text retunType=javatype
    uid=text [comment=string]>
  {<param name=text [type=”PARAM”] paramType=javatype ...>}
  {<throws excepType=javatype ...>}
</method>
```

Required Attributes for <method>

There are zero or more <method> entities in every <node> or <interface> entity. A <method> entity can contain zero or more <param> entities. As of this release, it cannot contain any other entities, but provisions have been made for a future <throws> entity. The parent of a <method> entity must be a <node> or <interface> entity.

The <method ...> tag has four required attributes: **name**, **signature**, **returnType**, and **uid**. (The **signature** attribute is syntactically optional, but highly recommended.)

- **returnType**. The value of the **returnType** attribute is required to specify the data type of the method. Examples:
  - **returnType**=”void”
  - **returnType**=”java.lang.String”

  Array types are denoted by a trailing pair of brackets: **returnType**=*datatype*[]
- **signature**. The presence of a valid **signature** attribute allows the e*Gate system to load the ETD more quickly, since it need not parse every method entity on the fly. For complete details on the **signature** attribute of the <method> entity, see **“Method Signature Syntax” on page 304**.

Optional Attributes for <method>

The **type** attribute is now unnecessary, but continues to be supported to maintain compatibility with XSC 0.2.

B.2.8 The <param> Entity

The <param> entity is a child of a <method> entity that specifies a particular parameter name and datatype in the argument list for the method.

Syntax

```
<param name=text [type="PARAM"] paramType=javatype
    uid=text [comment=string]>
</param>
```

Required Attributes for <param>

name

- The value of the **name** attribute must be a valid Java identifier. For details, see **"Identifier Characters" on page 305**.

paramType

- Specifies the data type of the parameter. Array types are denoted by a trailing pair of brackets: **paramType**=*datatype*[]

Optional Attributes for <param>

The **type** attribute is now unnecessary, but continues to be supported to maintain compatibility with XSC 0.2.

B.3 Table of XSC Entities and Their Attributes

B.3.1 Default Values

Most attributes have a default value. When an attribute has its default value, it should not normally be present in the XSC file. This avoids clutter and saves space. As a rule, all attributes of type "string" and "normal-safe" default to an empty string, all attributes of type "boolean" default to "false", and all numeric types default to "1". In **Table 21 on page 301**, only fields that do not conform to these default values have been noted.

B.3.2 Types

The types used are as follows:

- string = simple XML string (no escapes)
- normal-safe = a string escaped with **\u***xxxx* escapes to protect against attribute normalization
- boolean = "true" | "false" value

- u-number = non-negative integer or "undefined"
- u/u-number = like u-number, but also "unbounded" (with alias "-1")
- precedence = "parent" | "child" value.

Table 21 Attributes and Entities

attribute name	data type [1]	default value	Notes (for numbered notes, see end of table)
attributes for the <etd> entity:			
comment	string: ns		Provides documentation to help the end user understand the ETD's purpose and function and/or to provide warnings and tips.
dataEncoding	string	= *sscEncoding*	Not applicable. (For SSC-based **.xsc** files only.)
derived	boolean	"false"	Not supported in XSC 0.4 or earlier.
editable	boolean	"false"	Indicates whether the ETD Editor is used to modify the contents of the ETD.
name	string: ns	*none*	Contains the root name. Must match the value of the name attribute of exactly one of the <class> or <node> entities in the **.xsc** file.
sscEncoding	string	"US-ASCII"	Not applicable. (For SSC-based **.xsc** files only.)
type	string	*none*	Identifies the kind of message the ETD expects to parse/render/convert/transport/transform (such as DB, DTD, IDOC, SEF, SSC, X12, XSD)
uid	string		Must be unique to the values of all other uid attributes in the **.xsc** file.
xscVersion	string	*none*	In 0.6-compliant **.xsc** files, set xscVersion="0.6" to avoid unnecessary compatibility checking.
attributes for the <javaProps> entity:			
class	string		
codeAvailable	boolean		
comment	string: ns		
jarFile	string: ns		
package	string	*none*	
source	string: ns	*none*	
uid	string		Must be unique to the values of all other uid attributes in the **.xsc** file.
attributes for the <jar> entity:			
comment	string: ns		
file	string: ns	*none*	
uid	string		Must be unique to the values of all other uid attributes in the **.xsc** file.

Table 21 Attributes and Entities (Continued)

attribute name	data type [1]	default value	Notes (for numbered notes, see end of table)
attributes for the <interface> entity:			
comment	string: ns		
fqClass	string	*none*	
name	string	*none*	
uid	string		Must be unique to the values of all other uid attributes in the **.xsc** file.
attributes for the <method> entity:			
comment	string: ns		
name	string	*none*	
resultType	string	*none*	
signature	string	*none*	
type	string	"METHOD"	
uid	string		Must be unique to the values of all other uid attributes in the **.xsc** file.
attributes for the <param> entity:			
comment	string: ns		
name	string	*none*	
paramType	string	*none*	
type	string	"PARAM"	
uid	string		Must be unique to the values of all other uid attributes in the **.xsc** file.
attributes for the <throws> entity:			
comment	string: ns	*none*	
excepType	string	*none*	
uid	string		Must be unique to the values of all other uid attributes in the **.xsc** file.
attributes for the <class> and <node> entities:			
anchored	boolean	"false"	
array	boolean	"false"	
avoidMatch	boolean		Not supported in XSC 0.4 or earlier.
beginAnchored	boolean	"false"	Not applicable. (For SSC-based **.xsc** files only.)
beginDelim	string: ns	"false"	Not applicable. (For SSC-based **.xsc** files only.)
childMax		"undefined"	Not supported in XSC 0.4 or earlier.
childMin		"undefined"	Not supported in XSC 0.4 or earlier.
comment	string: ns	*(not defined)*	
defaultBytes	string: ns	= *defaultValue*	Not supported in XSC 0.4 or earlier.

Table 21 Attributes and Entities (Continued)

attribute name	data type [1]	default value	Notes (for numbered notes, see end of table)
defaultEncoding	string	= *etd.sscEncoding*	Not supported in XSC 0.4 or earlier.
defaultValue	string: ns		
encoding	string		Not supported in XSC 0.4 or earlier.
endAnchored	boolean	“false”	Not applicable. (For SSC-based **.xsc** files only.)
endDelim	string: ns		Not applicable. (For SSC-based **.xsc** files only.)
endOfRec	boolean	“false”	
exact	boolean	“false”	Introduced in XSC 0.6.
fixedValue	string: ns		
format	string: ns		Not supported in XSC 0.4 or earlier.
group	boolean	“false”	Introduced in XSC 0.6.
inputMatch	string: ns		
javaName	string	= *name*	
javaType	string	“java.lang.String”	Not supported in XSC 0.4 or earlier.
length		“undefined”	
lengthFrom		“undefined”	Not supported in XSC 0.4 or earlier.
lengthSize		“undefined”	Not supported in XSC 0.4 or earlier.
maxOccurs		“1”	
member	string: ns		Not supported in XSC 0.4 or earlier.
minOccurs		“1”	
name	string: ns	*none*	
nickName	string: ns	*none*	Not supported in XSC 0.4 or earlier.
offset		“undefined”	
optional	boolean	“false”	
order	string	“sequence”	
precedence	(string)	“child”	Value must be either “child” or “parent”. Not supported in XSC 0.4 or earlier.
public			Not supported in XSC 0.4 or earlier.
readOnly	boolean		
reference	string: ns		
required	boolean	“false”	
scavOutput	boolean		Not supported in XSC 0.4 or earlier.
scavenger	string: ns		Not supported in XSC 0.4 or earlier.
separator	boolean	“false”	
structure	string	*none*	

Table 21 Attributes and Entities (Continued)

attribute name	data type [1]	default value	Notes (for numbered notes, see end of table)
type	string	*none*	Some values legal in XSC 0.4, such as template name or Java type, are unsupported in XSC 0.6 and are not recommended.
uid	string		Must be unique to the values of all other **uid** attributes in the **.xsc** file.
Notes			
[1] Data types: ▪ **boolean** — either “true” or “false” ▪ **int|und** — either a nonnegative integer or “undefined” ▪ **int|und|unb** — one of the following: ♦ a nonnegative integer ♦ “unbounded” (“-1” is used as an alias) ♦ “undefined” ▪ **string** — a simple XML string (no escapes) ▪ **string: ns** — Normal Safe string (escapes use "\u" to protect against attribute normalization)			

B.4 Method Signature Syntax

The Java specification defines the **signature** attribute of the **<method>** entity and is included in this section. The low-level syntax is:

```
signature ::=   name "(" { array-type } ")" result-type
result-type ::= array-type | "V"
array-type ::=  { "[" } basic-type
basic-type ::=  primitive | class
primitive ::=   "Z" | "C" | "B" | "S" | "I" | "J" | "F" | "D"
class ::=       "L" package name ";"
package ::=     { name "/" }
```

The signature consists of a method name, followed by the parenthesized list of method parameter types, followed by the result type. Types are either one of the built-in primitive Java types like **int** (encoded as a single letter), or a fully qualified class, where the package components of the class qualification are separated by / (slash) instead of . (dot). The encoding of the primitive types is shown in Table 22.

Table 22 Signature Key Letters for Java Primitive Types

Key Letter	Java Type
V	void
Z	boolean
C	char
B	byte
S	short

Table 22 Signature Key Letters for Java Primitive Types (Continued)

Key Letter	Java Type
I	int
J	long
F	float
D	double

So, for example, a method declared like this in Java:

```
public void myNewMethod(byte[] ba, String str)
```

should be given the following **signature** attribute in the <method ...> tag:

```
signature="myNewMethod([[BLjava/lang/String;)V"
```

B.5 Identifier Characters

In general, Java identifiers consist of a start character followed by zero or more further characters.

- The first character must be a "Java letter"—that is, it must belong to the set of characters for which the method **isJavaIdentifierStart()** in class **java.lang.Character** returns true. This set includes alphabetic characters, currency symbols like "$" (dollar), and several separation characters like "_" (underscore).
- The remaining characters must be all be "Java letter-or-digit" characters—that is, they must be in the set of characters for which the method **isJavaIdentifierPart()** returns true. This set includes the alphanumeric characters, currency symbols, the underscore, and ignorable control characters.

For complete details, refer to the documentation for Java 2 SDK Standard Edition.

As of the current work's publication date, section 3.8 "Identifiers" is found at:

http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html#40625

B.6 JCS Properties

The class **com.stc.jcs.JCSProperties** provides a mechanism for setting global, per-user configuration flags for Java code. This is because environment variables, the method of choice in C, do not work well in Java. In effect, the static method **getProperty(String name)** checks the system properties for the given name (in other words, the **-D** option values); if not found there, it looks (cached) in the **.jcsrc** file in the user's home as given by the **user.home** system property; if not found there, it looks in a short list of predefined properties.

To get the list of currently predefined properties and properties define in the **.jcsrc** file, run the following command in the development environment:

```
java -classpath "<...>/ReleaseJava" com.stc.jcsre.JCSProperties -p -j
```

Table 23 lists properties currently used in the classes of the **com.stc.jcs.ssc** package.

Table 23 JCS Properties

Property Name	Type	Default	Meaning
Builder.jpure	boolean	true	Forces use of pure-Java conversion (no stctrans).
Emit.dent	int	4	Default indentation size for output (Java, XML).
Emit.tabs	int	8	Default assumed size of tab-stop for indentation.
GenSsc.utf8	boolean	true	Emits output-SSC in UTF-8 encoding, regardless of XSC's sscEncoding attribute value.
JGen.antlrLess	boolean	true	If not set, generates ANTLR-based unmarshaling code; otherwise, generates unmarshaling without ANTLR grammar/parser (faster).
JGen.bing	boolean	false	Byte-array interface not generated (no marshal/ unmarshal) if set. Not currently implemented.

Important: *These properties are intended for internal use, and are subject to change.*

```
String getProperty(String key, boolean now)
```

Get property from system; if not defined, try to load the **.jcsrc** file (unless previously cached and now="false") and read the property from it. When reading the **.jcsrc** file, cache it for future reference. In other words, the **now** parameter determines whether to use the current **.jcsrc** file contents or any old cached contents.

```
String getProperty(String key)
```

Like the previous method, but with now="false"—in other words, cached.

```
boolean getFlag (String key, boolean deft)
```

Like cached **getProperty()**, but translate the property value to a boolean value; if property not found, return the value of **deft**.

- Acceptable values for false are: "0", "F", "f", "false", "N", "n", "no".
- Acceptable values for true are: "1", "T", "t", "true", "Y", "y", "yes".
- If the property uses another string value, this method will throw a run-time exception.

Appendix C

The RMI Server

This appendix describes the use of an RMI server to simulate a generic external system. The RMI server is not part of the kit and is only provided to illustrate what you need to do to create a connector class and use the builder API. The discussion in this appendix focuses exclusively on the role played by the RMI server for the sample e*Way in **Chapter 9**.

Experimenting with Other Simulations of External Systems

A sample RMI server is provided for use as an external system for testing purposes. The sample illustrates the relationship between the methods and metadata in the "external system" being simulated by the RMI Server and the corresponding modifications to your e*Way and ETD builder that are required.

Note: *If developing a new e*Way using the e*Gate API Kit, use your own system to simulate an external system for testing purposes.*

C.1 Overview

The ETD and this sample schema require the sample RMI server provided in the SDK. An RMI application consists of the following four layers:

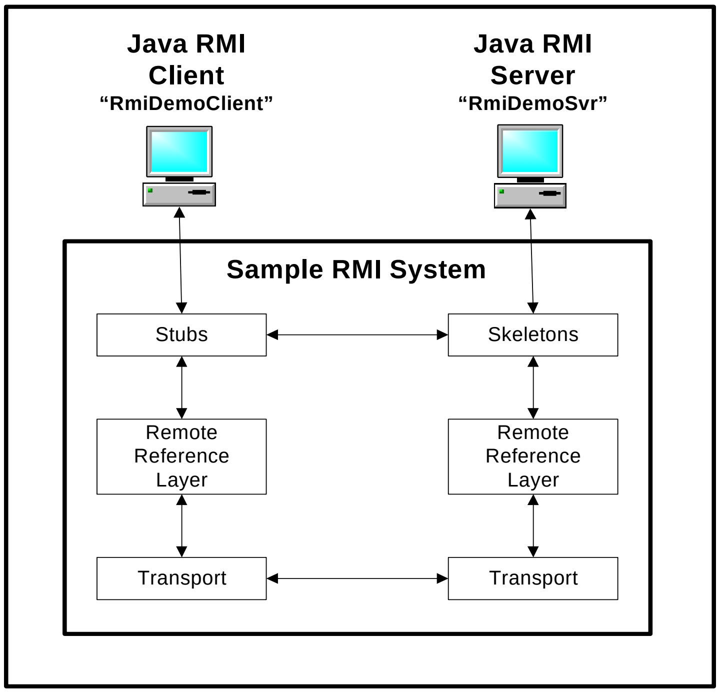
- Application layer
- Stubs or Skeletons layers
- Remote Reference layer
- Transport layer

The RMI client talks to its stub, which then sends the message to the remote reference layer. The remote reference layer then passes the message through the transport layer to the RMI server.

From the RMI server, the message is then passed from the transport layer to the remote reference layer. The message is retranslated to the skeleton and then to the server's object implementation.

Figure 69 shows the architecture of the sample RMI server system.

Figure 69 Sample RMI System Architecture



The sample RMI server has the following properties:

registered server name	**RmiDemoSvr**
package name	**com.stc.eways.samples.gmeek.builder.rmiDemoSvr**

The sample also has a stand-alone RMI client to validate and populate data in the server. In the sample RMI server, the data is saved to a file.

If you set up the sample and run it without any modification, you will have an RMI server named **RmiDemoSvr**, that receives calls from an RMI client, named **RmiDemoClient**. See Figure 69.

The RMI server allows you to create, delete, retrieve, or update an **account** object.

- The **account** object is a **java.util.HashMap** object with one key, "ACCOUNT_ID".

The server serializes the **account** object to a file, **account.data**, and responds to the client's request in one of the following ways:

- For **createAccount()**, the server appends the client's data to the file.
- For **deleteAccount()**, the server searches the file for a matching ACCOUNT_ID and deletes the data from the file.
- For **retrieveAccount()**, the server searches the file for a matching ACCOUNT_ID and returns the marshaled (parsed) **java.util.HashMap** to the calling RMI client.
- For **updateAccount()**, the server searches the file for a matching ACCOUNT_ID and replaces its data with the data supplied by the client.

Experimenting with Other APIs for the Provided “External System”

The sample ETD and RMI server were designed to be extremely flexible. For example:

- The RMI source code is provided, so you can see the effect of adding or deleting your own APIs and the corresponding methods in the ETD.
- Because the **account** object defined by the ETD is a simple java.util.**HashMap** object, you can experiment by adding or deleting fields and nodes. You can use any kind of key/value pair in the java.util.**HashMap** object, provided that the top level has the required key “ACCOUNT_ID” and all values are serializable.
- You can use the supplied **compile** scripts to recompile source code changes and rebuild the **RmiDemoSvr.jar** file. This allows you to easily test any changes you make to the RMI server and/or client.

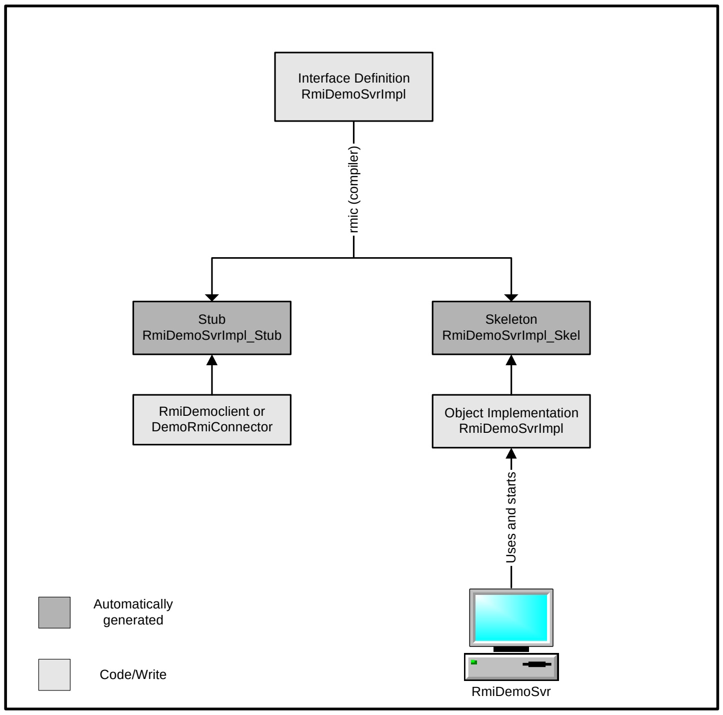
C.2 Sample Code for RMI

This section describes the source code files for the RMI server used in **“Developing an e*Way Using the Builder API” on page 119**.

- **RmiDemoSvrIntf.java** on page 310: Sample program that defines methods for the accounting system simulated by the RMI server.
- **RmiDemoClient.java** on page 310: Sample program that tests the RMI server by acting as a sample client.
- **RmiDemoSvrImpl.java** on page 312: Sample program demonstrating the use of java.util.HashMap saved to a file.
- **RmiDemoSvr.java** on page 317: Sample program showing how to start an RMI server.

Figure 70 on page 310 shows the process of creating an RMI application which includes an RMI server and RMI client.

Figure 70 Creating an RMI Application



RmiDemoSvrIntf.java

RmiDemoSvrIntf.java is a short sample program that defines the methods provided by the server for the sample discussed in **"Developing an e*Way Using the Builder API" on page 119**.

```
package com.stc.eways.samples.gmeek.builder.rmiDemoSvr;
import java.rmi.*;
/** Remote interface specifying methods that must be provided by the
 * server.
 *
 * @version $Revision: 1.1.2.1 $
 */
public interface RmiDemoSvrIntf extends java.rmi.Remote
{
  public String  sayEcho(String myName) throws RemoteException;
  public boolean createAccount(java.util.Map  data) throws
      RemoteException;
  public boolean deleteAccount(java.util.Map criteria) throws
      RemoteException;
  public boolean updateAccount(java.util.Map criteria) throws
      RemoteException;
  public java.util.HashMap retrieveAccount(java.util.HashMap criteria)
      throws RemoteException;
  public java.util.List retrieveAllAccountId() throws RemoteException;
}
```

RmiDemoClient.java

RmiDemoClient.java is a sample client program that runs stand-alone. This program is used to populate the initial set of data for the RMI server.

```
package com.stc.eways.samples.gmeek.builder.rmiDemoSvr;
import java.io.*;
import java.rmi.Naming;
/** Client for the RMI server RmiDemoSvr
 *
 * @version $Revision: 1.1.2.1 $
 */
public class RmiDemoClient
{
  /** Creates new RmiDemoClient
   */
  public RmiDemoClient()
  {
  }

  public static void usage()
  {
    System.err.println("Usage: ");
    System.err.println("  java RmiDemoClient <server name> <port> <var>
or");
    System.err.println("  or");
    System.err.println("  java RmiDemoClient <server name> <var>");
  }

  public static void main(String args[])
    throws Exception
  {
    String url;

    if (args.length == 3 )
    {
      url = new String("//"+args[0]+":"+args[1]+"/RmiDemoSvr");
    }
    else if (args.length == 2)
    {
      url = new String("//"+args[0]+"/RmiDemoSvr");
    }
    else
    {
      RmiDemoClient.usage();
      return;
    }
    // RMI registry lookup
    //
    RmiDemoSvrIntf echoRef = (RmiDemoSvrIntf)Naming.lookup(url);
    // Call echo method on server
    //
    System.out.println(echoRef.sayEcho(args[args.length-1]));
    // Retrive accounts from server
    //
    java.util.List alllist = echoRef.retrieveAllAccountId();
    System.out.println("total number of accounts: "+alllist.size());
    for (int i = 0; i < alllist.size(); i++)
    {
      System.out.println("ACCOUNT_ID: "+(String)alllist.get(i));
    }
    // Retrieve a particular account then update it or
    // create it if it does not exist.
    //
    java.util.HashMap accnt = new java.util.HashMap();
    accnt.put("ACCOUNT_ID", "1000");
    java.util.HashMap  accnt1=  echoRef.retrieveAccount(accnt);
    if(accnt1 != null)
    {
      System.out.println(accnt1.toString());
      accnt1.put("ACCOUNT_TYPE","Checking");
      accnt1.put("ADDRESS","102 Main St.");
      accnt1.put("CITY","Monrovia");
      accnt1.put("STATE","CA");
      accnt1.put("ZIP","91006");
      echoRef.updateAccount(accnt1);
      System.out.println("update account: " +accnt1.toString());
    }
    else
    {
      accnt.put("ACCOUNT_TYPE","saving");
      accnt.put("ADDRESS","103 Main St.");
      accnt.put("CITY","Monrovia");
      accnt.put("STATE","CA");
      accnt.put("ZIP","91006");
      echoRef.createAccount(accnt);
      System.out.println("create account: " +accnt.toString());
    }
  }
}
```

RmiDemoSvrImpl.java

RmiDemoSvrImpl.java is a sample program that implements the RMI remote interface.

```
package com.stc.eways.samples.gmeek.builder.rmiDemoSvr;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.MalformedURLException;
import java.io.*;

/** Unicast remote object implementing RmiGmeek2Test interface.
 *
 * @version $Revision: 1.1.2.2 $
 */
public class RmiDemoSvrImpl extends java.rmi.server.UnicastRemoteObject
implements RmiDemoSvrIntf
{
    File theFile = null;
    /** Constructs RmiDemoSvrImpl object and exports it on default port.
     */
    public RmiDemoSvrImpl() throws RemoteException
    {
        super();
        theFile = new File("account.data");
        try
        {
            if (!theFile.exists())
            {
                theFile.createNewFile();
            }
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }

    /** Constructs RmiDemoSvrImpl object and exports it on specified
port.
```

```
     * @param port The port for exporting
     */
    public RmiDemoSvrImpl(int port) throws RemoteException
    {
        super(port);
    }

    /** Register RmiDemoSvrImpl object with the RMI registry.
     *
     * @param name     name identifying the service in the RMI registry
     * @param create   create local registry if necessary
     *
     * @throw RemoteException if cannot be exported or bound to RMI registry
     * @throw MalformedURLException if name cannot be used to construct a valid
     * URL
     * @throw IllegalArgumentException if null passed as name
     */
    public static void registerToRegistry(String name, Remote obj,
       boolean create) throws RemoteException, MalformedURLException
    {
        if (name == null)
        {
            throw new IllegalArgumentException(
              "registration name can not be null");
        }

        try
        {
            Naming.rebind(name, obj);
        }
        catch (RemoteException ex)
        {
            if (create)
            {
                Registry r = LocateRegistry.
createRegistry(Registry.REGISTRY_PORT);
                r.rebind(name, obj);
            }
            else
            {
                throw ex;
            }
        }
    }

    /** return a string saying hello back to the client
     *
     */
    public String sayEcho(String myName) throws RemoteException
    {
        return "\nHello "+ myName + "!!\n";
    }

    /** Create an account object in the output file and add it to the list
     *  of accounts
     *
     * @param  data  map of accounts
     */
    public boolean createAccount(java.util.Map data) throws RemoteException
    {
        try
        {
            java.util.Vector allAccounts = new java.util.Vector();
            {
                FileInputStream fileinstrm = new FileInputStream(theFile);
                if (fileinstrm.available() > 0)
                {
                    ObjectInputStream pin = new ObjectInputStream(fileinstrm);
                    allAccounts = (java.util.Vector)pin.readObject();
                    pin.close();
                }
            };

            allAccounts.add(data);
            {
                FileOutputStream fileonstrm = new FileOutputStream(theFile);
                ObjectOutputStream pout = new ObjectOutputStream(fileonstrm);
                pout.writeObject(allAccounts);
                pout.flush();
                pout.close();
            }
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            return false;
        }
        return true;
    }

    /** Delete an account object from the output file and remove it from
     *  the list of accounts
     *
     * @param  data  map of accounts
     */
    public boolean deleteAccount(java.util.Map criteria) throws RemoteException
    {
        try
        {
            FileInputStream fileinstrm = new FileInputStream(theFile);
            java.util.Vector allAccounts = new java.util.Vector();

            if (fileinstrm.available() > 0)
            {
                ObjectInputStream pin = new ObjectInputStream(fileinstrm);
                allAccounts = (java.util.Vector)pin.readObject();
                pin.close();
            }
            else
                return false;

            for(int i =0; i< allAccounts.size();i++)
            {
                java.util.HashMap account =
                  (java.util.HashMap)allAccounts.get(i);
                String queryAccountID = (String)criteria.get("ACCOUNT_ID");
                if (queryAccountID.compareToIgnoreCase(
(String)account.get("ACCOUNT_ID")) == 0)
                {
                    allAccounts.remove(i);
                    return true;
                }
            }
            {
                FileOutputStream fileonstrm = new
FileOutputStream(theFile);
                ObjectOutputStream pout = new
ObjectOutputStream(fileonstrm);
                pout.writeObject(allAccounts);
                pout.flush();
                pout.close();
            }
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
        return false;
    }

    /** Update an account object with the matching account id
     *
     * @param  data  map of accounts
     */
    public boolean updateAccount(java.util.Map criteria) throws
RemoteException
    {
        boolean updated = false;
        try
        {
            FileInputStream fileinstrm = new FileInputStream(theFile);
            java.util.Vector allAccounts = new java.util.Vector();

            if (fileinstrm.available() > 0)
            {
                ObjectInputStream pin = new
ObjectInputStream(fileinstrm);
                allAccounts = (java.util.Vector)pin.readObject();
                pin.close();
            }
            else
                return false;

            for(int i =0; i< allAccounts.size();i++)
            {
                java.util.HashMap account =
                    (java.util.HashMap)allAccounts.get(i);
                String queryAccountID =
(String)criteria.get("ACCOUNT_ID");
                if (queryAccountID.compareToIgnoreCase(
                     (String)account.get("ACCOUNT_ID")) == 0)
                {
                    allAccounts.remove(i);
                    allAccounts.add(criteria);
                    updated = true;
                }
            }

            if(updated)
            {
                FileOutputStream fileonstrm = new
FileOutputStream(theFile);
                ObjectOutputStream pout = new
ObjectOutputStream(fileonstrm);
                pout.writeObject(allAccounts);
                pout.flush();
                pout.close();
            }

        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
        return updated;
    }

    /** Return an account object with the matching account id
     *
     * @param  data  map of accounts
     */
    public java.util.HashMap retrieveAccount(java.util.HashMap
criteria)
        throws RemoteException
    {
        try
        {
            FileInputStream fileinstrm = new FileInputStream(theFile);
            java.util.Vector allAccounts = new java.util.Vector();
            if (fileinstrm.available() > 0)
            {
                ObjectInputStream pin = new
ObjectInputStream(fileinstrm);
                allAccounts = (java.util.Vector)pin.readObject();
                pin.close();
            }

            for (int i =0; i< allAccounts.size();i++)
            {
                java.util.HashMap account =
                  (java.util.HashMap)allAccounts.get(i);
                String queryAccountID =
(String)criteria.get("ACCOUNT_ID");
                if (queryAccountID.compareToIgnoreCase(
(String)account.get("ACCOUNT_ID")) == 0)
                {
                    return account;
                }
            }
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
        return null;
    }

    /** Retrieve all accounts in the list
     *
     * @param  data  map of accounts
     */
    public java.util.List retrieveAllAccountId() throws RemoteException
    {
        java.util.ArrayList theList = new java.util.ArrayList();
        try
        {
            FileInputStream fileinstrm = new FileInputStream(theFile);
            java.util.Vector allAccounts = new java.util.Vector();

            if (fileinstrm.available() > 0)
            {
                ObjectInputStream pin = new
ObjectInputStream(fileinstrm);
                allAccounts = (java.util.Vector)pin.readObject();
                pin.close();
            }

            for (int i =0; i< allAccounts.size();i++)
            {
                java.util.HashMap account =
                 (java.util.HashMap)allAccounts.get(i);
                String queryAccountID =
(String)account.get("ACCOUNT_ID");
                theList.add(queryAccountID);
            }
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
        return theList;
    }
}
```

RmiDemoSvr.java

RmiDemoSvr.java is a sample program that starts the server and registers the server with the RMI registry using the name “RmiDemoSvr.”

```
package com.stc.eways.samples.gmeek.builder.rmiDemoSvr;
import java.rmi.Naming;
/** RMI server which instantiates RmiGmeek2TestServerImpl
 *
 * @version $Revision: 1.1.2.1 $
 */
public class RmiDemoSvr
{
  /** Creates new RmiGmeek2TestServer
   */
  public RmiDemoSvr()
  {
  }

  /** Usage banner
   */
  public static void usage()
  {
    System.err.println("Usage:");
    System.err.println("  java
com.stc.eways.samples.gmeek.builder.RmiDemoSvr.RmiDemoSvr
                        <rmi registry port>");
    System.err.println("  or");
    System.err.println("  java
        com.stc.eways.samples.gmeek.builder.RmiDemoSvr.RmiDemoSvr");
  }

  public static void main(String args[]) throws Exception
  {
    String url = "//localhost/RmiDemoSvr";
    RmiDemoSvrImpl echoRef = null;
    if (args.length > 1 )
    {
      RmiDemoSvr.usage();
    }
    else if (args.length == 1)
    {
      echoRef = new RmiDemoSvrImpl();
      url = new String("//localhost:"+args[0]+"/RmiDemoSvr");
    }
    else if (args.length == 0)
    {
      echoRef = new RmiDemoSvrImpl();
    }
    else
    {
      RmiDemoSvr.usage();
    }

    // register this server
    //
    Naming.rebind(url,echoRef);

    System.out.println("RmiDemoSvr object ready and bound to the name:
"+url);
  }
}
```

Index