SeeBeyond ICAN Suite

eWay Development Kit User's Guide

Release 5.0

SeeBeyond Proprietary and Confidential

The information contained in this document is subject to change and is updated periodically to reflect changes to the applicable software. Although every effort has been made to ensure the accuracy of this document, SeeBeyond Technology Corporation (SeeBeyond) assumes no responsibility for any errors that may appear herein. The software described in this document is furnished under a License Agreement and may be used or copied only in accordance with the terms of such License Agreement. Printing, copying, or reproducing this document in any fashion is prohibited except in accordance with the License Agreement. The contents of this document are designated as being confidential and proprietary; are considered to be trade secrets of SeeBeyond; and may be used only in accordance with the License Agreement, as protected and enforceable by law. SeeBeyond assumes no responsibility for the use or reliability of its software on platforms that are not supported by SeeBeyond.

SeeBeyond, e*Gate, e*Way, and e*Xchange are the registered trademarks of SeeBeyond Technology Corporation in the United States and/or select foreign countries. The SeeBeyond logo, SeeBeyond Integrated Composite Application Network Suite, eGate, eWay, eInsight, eVision, eXchange, eView, eIndex, eTL, ePortal, eBAM, and e*Insight are trademarks of SeeBeyond Technology Corporation. The absence of a trademark from this list does not constitute a waiver of SeeBeyond Technology Corporation's intellectual property rights concerning that trademark. This document may contain references to other company, brand, and product names. These company, brand, and product names are used herein for identification purposes only and may be the trademarks of their respective owners.

© 2004 by SeeBeyond Technology Corporation. All Rights Reserved. This work is protected as an unpublished work under the copyright laws.

This work is confidential and proprietary information of SeeBeyond and must be maintained in strict confidence.

Version 20041223133955.

Contents

Chapter 1

Introducing the eWay Development Kit 7

About the eWay Development Kit 7

About this Document 8

What's in This Document 8

Scope 8

Intended Audience 8

Document Conventions 9

Related Documents 9

SeeBeyond Web Site 10

SeeBeyond Documentation Feedback 10

Chapter 2

Installing the eWay Development Kit 11

System Requirements 11

Supported Operating Systems 11

Supported ICAN Versions 12

Installing the eDK 12

Directories Created After Installation 13

Additional Files Created During Installation 13

Installing Earlier Versions of the JDK/SDK 14

Installing ESR 78987 14

Installing eDK Sample Projects and Javadocs 14

Chapter 3

Using the eWay Development Kit 15

Overview 15

About the eWay Development Kit Build Tool 16

About the eWay Implementation Environment 17

About the Resource Adapter Framework 17

Setting up Your Implementation Environment 17

Steps Required to Build an eWay 18
Step 1: Acquire an eDK eWay License File 18
Licensing Process 19
Request e-mail Format Conventions 19
Requesting a New License 19
Steps to Request an eDK eWay License 20
Requesting a Current License to be ReIssued 21
Steps to Reissue an eDK eWay License 22
Step 2: Set the Environment Variables 23
Step 3: Start the eWay Development Kit Build Tool 23
Step 4: Create and Specify the New eWay 24
Name and Description 25
Icons 25
Change History 26
Imported Files 26
Step 5: Enter the Required eWay Client Interfaces 26
Defining Your eWay's Java Interface 26
Defining Your eWay's BPEL Interface 27
Creating Methods (JCE) 28
Creating User Defined Class Files 29
Creating Attributes **30**
Creating Operations 30
BPEL Operations in ICAN 30
Step 6: Define the eWay Configuration Template 32
Connectivity Map Configuration Sections and Properties 36
External System Sections and Properties 37
Deleting Sections and Properties 37
Disabling and Enabling Sections 37
Step 7: Run the Code Generator 38
Saving Your Work 39
Opening Previously Saved Work 39
Choosing a Working Directory 40
Step 8: Implement and Build the Generated Shell Code 40
Step 9: Build the .sar File 40

eWay Folders Created After Shell Code Generation 41
connectors Folder 41
eways Folder 43
eWay Code Created After Generation 44

eWay Implementation Details 46

eWay Components 47

Suggested Conventions for Writing JNI Code 47

Extending Third-Party Resource Adapters 48
Providing the AppConn Client Interface 49
Sample MCF Subclass Implementation 51

Chapter 4

eDK eWay Concepts and Best Practices 55

Implementing Connection Logic to the External System 55
Implementing XA 56

Establishing Connections to the EIS 57
Automatic Connection Establishment Mode 57
Dynamic Connection 57
Overriding Configurations at Design-time 59

Specifying Configuration Properties 59
Connectivity Map eWay Properties 60
External System Properties 61

Wrapping Third-Party .jar Files 62

Source Control 62

Maintaining and Persisting State in Java Collaborations 63

Generating Javadocs 63

Chapter 5

Sample eDK eWay Projects 65

Importing eDK Samples 65
Importing a Sample into the eWay Development Kit Build Tool 65

Creating the edkfile Sample in the Build Tool 66
Overview 66
Step 1: Acquire an eDK eWay License File 67
Step 2: Set the Environment Variables 67
Step 3: Start the eWay Development Kit Build Tool 67
Step 4: Create and Specify the New eWay 68
Step 5: Enter the Required eWay Interfaces 69
Step 6: Define the eWay Configuration Template 71
inbound-configuration Properties 71
outbound-configuration Properties 73
Step 7: Run the Code Generator 75
Step 8: Implement the eWay 77
Step 9: Build the .sar File 84
Step 10: Upload the New eWay to the ICAN Repository 84
Step 11: Run the Enterprise Designer Update Center 85
Step 12: Creating, Building, and Deploying Sample Projects 85

Chapter 6

Using eDK-Based eWay Java Methods 86

eWay Development Kit Javadoc 86
eDK-Based eWay Classes and Methods 86

eDK-Based eWay Classes 86

Chapter 7

Adding and Sending Custom Alert Messages 89

eDK Alerts 89

Adding eWay Specific Alert Message Codes 90

Java Code Changes 90

What to Pass for alertMsgCode and alertMsgCodeArgs 90

Installing Alert Code Properties Files (install.xml changes) 91

Sending eWay Specific Alerts 91

Chapter 8

Appendix A 93

J2EE Connector Architecture Overview 93

RA Framework Class Diagram 94

RA Framework Sequence Diagram 96

Client Application Sequence Diagram 97

Application Connection Interfaces 98

eWay Connection Interfaces 99

Chapter 9

Appendix B 101

Generating eDK Code by Command Line 101

eDK Definition File 102

Chapter 1

Introducing the eWay Development Kit

Welcome to the *eWay Development Kit User's Guide*. This document includes information about installing, configuring, and using the eWay Development Kit, also referred to as the eDK throughout this guide.

What's in this Chapter

- **About the eWay Development Kit** on page 7
- **About this Document** on page 8
- **Related Documents** on page 9
- **SeeBeyond Web Site** on page 10
- **SeeBeyond Documentation Feedback** on page 10

1.1 About the eWay Development Kit

The eWay Development Kit is a development tool for creating inbound and outbound eWay components that conform to standard JCA (J2EE Connector Architecture) 1.5. The eDK includes a build tool (GUI application) which allows users to define the eWay interfaces as OTDs (Object Type Definitions) that are exposed in Java Collaboration Definitions and BPEL Business Processes, and an implementation environment for implementing and building the interfaces.

The eDK is designed to alleviate many of the tedious development steps required to build eWays by automating the standard eWay build process. eWays created with the eDK are built, packaged, installed, and executed the same way as standard SeeBeyond eWays.

Using the eDK does not require any specialized knowledge of ICAN APIs to tie the eWay component into the ICAN suite. The only requirement is an understanding how to connect and exchange data with the external system.

1.2 About this Document

This guide explains how to install, configure, and operate the SeeBeyond® Integrated Composite Application Network Suite™ (ICAN) eWay Development Kit, also referred to as the eWay Development Kit throughout this guide.

1.2.1 What's in This Document

This document includes the following chapters:

- Chapter 1 **"Introducing the eWay Development Kit"** provides an overview of the eWay Development Kit.
- Chapter 2 **"Installing the eWay Development Kit"** provides installation instructions, including a list of supported operating systems, system requirements, and JNI protocol considerations.
- Chapter 3 **"Using the eWay Development Kit"** describes the features and functionality of the eWay Development Kit Build Tool and implementation environment.
- Chapter 4 **"eDK eWay Concepts and Best Practices"** provides suggestions and tips on creating eDK based eWays.
- Chapter 5 **"Sample eDK eWay Projects"** describes how to import and use the sample projects included in the eWay Development Kit.
- Chapter 6 **"Using eDK-Based eWay Java Methods"** describes the Class files that include the eWay Java methods.
- Chapter 7 **"Adding and Sending Custom Alert Messages"** describes how to add and send custom alert messages using the Resource Adapter framework.
- Chapter 8 **"Appendix A"** contains additional information on the J2EE Connector Architecture and RA Framework.
- Chapter 9 **"Appendix B"** contains additional information on Generating eDK code by command line, and manually creating an eDK definition file.

1.2.2 Scope

This guide describes how to install and use the eWay Development Kit for the purpose of developing new eWays that function within the ICAN suite of products. Additional detailed information, such as the steps required to create eDK based eWay projects in the ICAN Enterprise Designer are not included in this guide; however, sample eWays are included in the **eWaysDevelopmentKit.sar** file to explain how an implementation of an eDK based eWay might occur.

1.2.3 Intended Audience

This guide is intended for experienced developers who have a Java programming background. Some knowledge of J2EE standards and methodology, and knowlege of

JCA standards are also helpful. Users should also be familiar with the existing ICAN products and the role that eWays play within that product line, the Windows-based operating systems that the eWay Development Kit Build Tool is installed on, and operating systems supported by the completed eDK eWays.

1.2.4 Document Conventions

The following conventions are observed throughout this document.

Table 1 Document Conventions

Text	Convention	Example
Names of buttons, files, icons, parameters, variables, methods, menus, and objects	**Bold** text	▪ Click **OK** to save and close. ▪ From the **File** menu, select **Exit**. ▪ Select the **logicalhost.exe** file. ▪ Enter the **timeout** value. ▪ Use the **getClassName()** method. ▪ Configure the **Inbound** File eWay.
Command line arguments, code samples	`Fixed` font. Variables are shown in ***bold italic***.	`bootstrap -p` ***password***
Hypertext links	**Blue** text	See **"Document Conventions" on page 9**
Hypertext links for Web addresses (URLs) or email addresses	**Blue underlined** text	**http://www.seebeyond.com** **docfeedback@seebeyond.com**

1.3 Related Documents

The following SeeBeyond documents provide additional information about the ICAN product suite:

- *eGate Integrator User's Guide*
- *eGate Integrator Tutorial*
- *Alert Agent User's Guide*

1.4 SeeBeyond Web Site

The SeeBeyond Web site is your best source for up-to-the-minute product news and technical support information. The site's URL is:

http://www.seebeyond.com

1.5 SeeBeyond Documentation Feedback

We appreciate your feedback. Please send any comments or suggestions regarding this document to:

docfeedback@seebeyond.com

Chapter 2

Installing the eWay Development Kit

This chapter describes how to install the eWay Development Kit.

What's in this Chapter

- **System Requirements** on page 11
- **Supported Operating Systems** on page 11
- **Installing the eDK** on page 12
- **Installing ESR 78987** on page 14
- **Installing eDK Sample Projects and Javadocs** on page 14

2.1 System Requirements

To use the eWay Development Kit, you need:

- eGate Repository.
- TCP/IP network connection.

2.2 Supported Operating Systems

This section lists the system requirements for each platform. The Readme.txt file (located in the root directory of the Repository CD-ROM) contains the most up-to-date operating system requirements for the supported platforms. The requirements listed in the following sections are in addition to the operating system requirements.

The eWay Development Kit is designed to run on a the following operating systems.

- Windows 2000, Windows XP, and Windows Server 2003

Note: *A minimum monitor resolution of 1024 x 768 is required to run the eWay Development Kit Build Tool.*

2.2.1 Supported ICAN Versions

eWays developed with the eDK work with eGate version 5.0.5. If you want an eWay created with the eDK to work on eGate 5.0.4, then you must install **ESR 74717** on an eGate 5.0.4 Repository.

eWays created in the eDK run on the following operating systems that are supported by eGate Integrator, including:

- Windows 2000, Windows XP, and Windows Server 2003
- HP Tru64 V5.1A and V5.1B with required patches
- HP-UX 11.0, 11i (PA-RISC), and 11i v2.0 (11.23) with required patches and parameter changes
- IBM AIX 5.1L and 5.2 with required Maintenance level patches
- Red Hat Linux Advanced Server 2.1 (Intel x86)
- Red Hat Linux 8 (Intel x86)
- Sun Solaris 8 and 9 with required patches
- SuSE Linux Enterprise Server 8 (Intel x86)

Refer to the *SeeBeyond ICAN Suite Installation Guide* for additional platform requirements when running HP-UX or IBM AIX.

2.3 Installing the eDK

During the eDK installation process, the Enterprise Manager, a web-based application, is used to select and upload the **eWayDevelopmentKit.sar** file from the installation CD-ROM to the Repository.

The steps required to install the eDK include:

1. Create a new eDK root directory folder (for example: C:\eDK).
2. Login to Enterprise Manager and click the **Admin** tab.
3. Click **Browse** and open the **ProductsManifest.xml** file from the installation CD-ROM. The eWay Development Kit appears as an uploadable product.
4. Click **Browse** and upload the **license.sar** file.
5. Click **Browse** and upload the **eWayDevelopmentKit.sar** file.
6. Click the **Downloads** tab and select the eWay Development Kit.
7. Click **Open** from the File Download window and extract the **edk.zip** file to the eDK root directory folder.

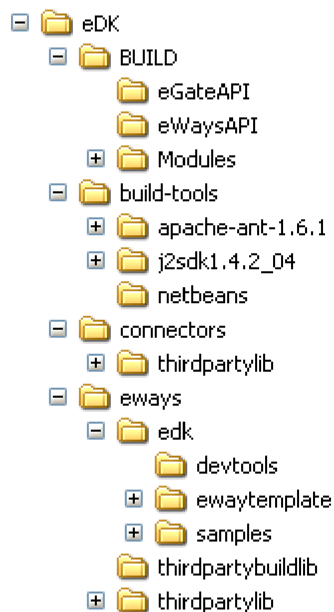
2.3.1 Directories Created After Installation

Several directories are extracted from the **edk.zip** file, including a subdirectory for development tools and a subdirectory containing code templates.

Main directories created after extraction include:

- **BUILD** – contains all the API **.jar** files required to build an eWay.
- **build-tools** – contains all the build tools (**Apache Ant** , **JDK 1.4**™, and **Netbeans**™) that are necessary to build an eWay.
- **connectors** – contains JCA connector code, Apache Ant build scripts for building JCA code, and required third-party **.jar** files.
- **eways** – contains all necessary third-party **.jar** files, and development tools to run the eDK, as well as a folder containing completed eDK samples.

Figure 1 Directories Created After Installation



2.3.2 Additional Files Created During Installation

The following files are also created during installation of the eWay Development Kit Build Tool.

- **env.bat** – used to set up your build environment.
- **stc.properties** – contains global property settings used in **Apache Ant** build scripts.

2.3.3 Installing Earlier Versions of the JDK/SDK

Installation of previous JDK/SDK versions may be required to build your Resource Adapter. As an example, some external APIs or earlier versions of the Logical Host only support JDK/SDK version 1.3.1.

You can download previous versions of the JDK/SDK from:

http://java.sun.com/downloads/index.html

Note: *You must copy the downloaded previous version of the JDK/SDK to the* ***build-tools*** *folder.*

2.4 Installing ESR 78987

ESR 78987 must be installed in the ICAN Repository to permit Enterprise Manager to upload eWays created with the eWay Development Kit.

This section provides an overview of the steps required to install this ESR, and references sections of the *SeeBeyond ICAN Suite Installation Guide* for detailed procedures.

The *SeeBeyond ICAN Suite Installation Guide* (ICAN_Install_Guide.pdf) is located on the SeeBeyond ICAN Suite Repository Disc 1 and Repository Disc 2.

To install this ESR:

1. From the **Admin** tab of Enterprise Manager, extract the ESR0078987-dist.zip file to a temporary directory as described in “Extracting ESR Distribution **.zip** Files.”
2. Install the Repository **.zip** file ESR78987.zip as described in “Installing Repository ESRs.”

2.5 Installing eDK Sample Projects and Javadocs

The following steps are required to install the eDK sample projects and Javadocs.

1. From the **Documentation** tab of the Enterprise Manager, click **eGate eWay** to view the list of files available for this product.
2. Click **Download Sample** to open the **eWayDevelopmentKitDocs.zip** file.
3. Use WinZip to extract the sample files to the desired location.
4. Click **Download Javadocs** to open the **eWay_Development_Kit_Javadoc.zip** file.
5. Use WinZip to extract the Javadocs files to the desired location.

   After you complete the process of installing the Repository, Logical Host, and Enterprise Designer (as described in the *SeeBeyond ICAN Suite Installation Guide*), refer to the *eGate Integrator Tutorial* for instructions on importing the sample project into your repository using the Enterprise Designer.

Chapter 3

Using the eWay Development Kit

This chapter describes how to use the eWay Development Kit.

What's in this Chapter

- **Overview** on page 15
- **About the eWay Development Kit Build Tool** on page 16
- **About the eWay Implementation Environment** on page 17
- **Steps Required to Build an eWay** on page 18
- **eWay Folders Created After Shell Code Generation** on page 41
- **eWay Implementation Details** on page 46
- **eWay Components** on page 47
- **Suggested Conventions for Writing JNI Code** on page 47
- **Extending Third-Party Resource Adapters** on page 48

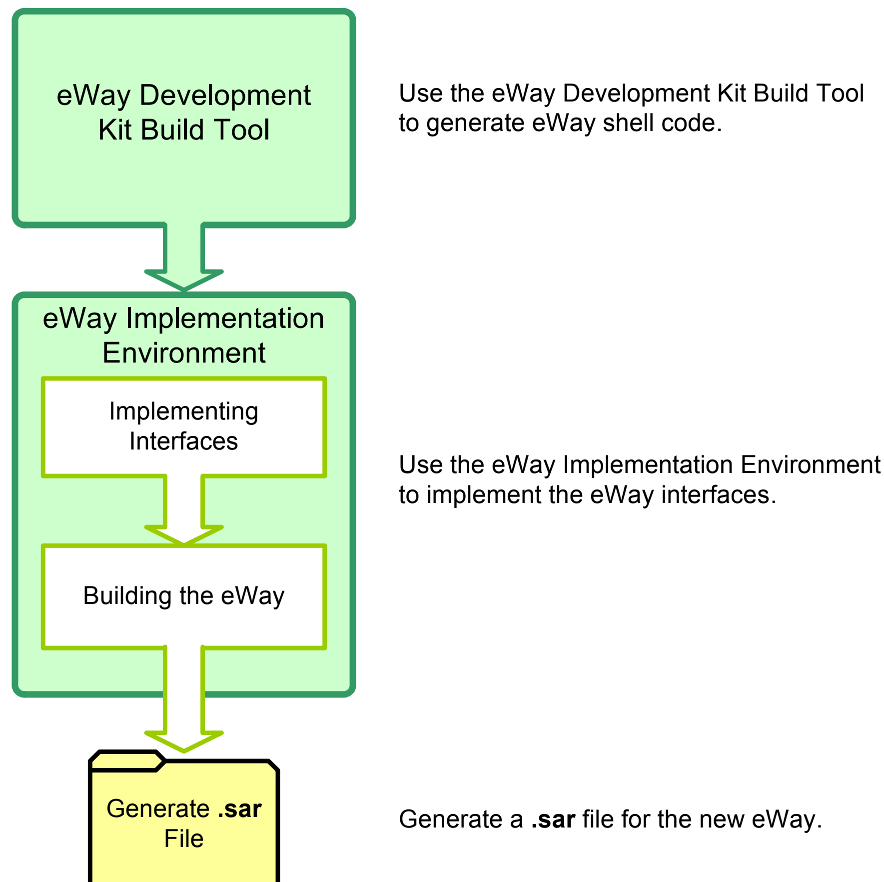
3.1 Overview

The eWay Development Kit is comprised of two main components that work together to create an eWay **.sar** file that can be plugged into ICAN 5.0.x for external system connectivity. These components include:

- **eWay Development Kit Build Tool** – a GUI based tool used to generate eWay shell code.
- **eWay Implementation Environment** – a development environment that is required to implement eWay interfaces generated by the eWay Development Kit Build Tool.

Figure 2 illustrates the relationship between the eWay Development Kit Build Tool and the eWay implementation environment.

Figure 2 eWay Development Kit Component Overview



3.2 About the eWay Development Kit Build Tool

The eWay Development Kit Build Tool is a stand-alone code generator that is used to define the eWay components, generate the Shell Code based on the eWay definition, and provide an implementation environment to build an eWay.

The shell code created with the build tool includes the eWay name and description; methods, attributes, and user defined classes or data containers that make up the eWay's static Object Type Definitions (OTDs); and the attributes that describe the Environment and eWay configuration properties.

You can perform eDK code generation one of two ways:

- The GUI based tool – see **"Steps Required to Build an eWay" on page 18**.
- By command line – see **"Generating eDK Code by Command Line" on page 101**.

The result of shell code generation is the creation of the "connectors" and "eways" folders where eWay implementation is performed. See **"Step 8: Implement the eWay" on page 77** for more information.

3.3 About the eWay Implementation Environment

The eWay implementation environment includes a directory structure and source files, eGate APIs, and third-party **.jar** files that are used for implementing the interfaces defined in the generated shell code. Implementation also includes building the code with the **Apache Ant** build tool, and generating an eWay **.sar** file.

The shell code generated by the build tool creates the necessary Resource Adapter components, (see **"eWay Code Created After Generation" on page 44**) which allows the eWay to run in the ICAN suite.

3.3.1 About the Resource Adapter Framework

SeeBeyond developed a Resource Adapter (RA) framework for developing resource adapters based on the J2EE Connector Architecture. The framework is a set of interfaces and abstract classes which simplify the development of resource adapters. The framework also facilitates the development of the resource adapter client interface which is based on the SeeBeyond AppConn interface used by ICAN's Collaboration framework.

The eDK generates J2EE Connector code based on the SeeBeyond Resource Adapter (RA) framework. The framework was designed so that development is focused on the definition and implementation of the client interface to be exposed by the eWay.

For more information on the framework interfaces, see **"Application Connection Interfaces" on page 98**.

3.3.2 Setting up Your Implementation Environment

An implementation environment is required to compile Java source files and to build an eWay **.sar** file.

Run the **env.bat** file located at the root level of the extracted eDK folder to set up the implementation environment.

3.4 Steps Required to Build an eWay

The following steps outline a typical user experience of using the eDK to build an eDK-based eWay for implementation in an ICAN project. Steps one through six are considered iterative eWay development procedures.

Steps required to build an eWay using the eWay Development Kit include:

1. Acquire an eDK eWay license file.
2. Set the environment variables.
3. Start the eWay Development Kit Build Tool.
4. Create and specify details of the new eWay – such as the eWay Name, Description, Version, and so forth.
5. Enter the required eWay Interfaces (including any return types, parameter names, parameter types, exceptions thrown, and so forth.
6. Define the eWay configuration template.
7. Run the code generator to create the eWay shell code (using either the eWay Development Kit Build Tool or by command line).
8. Implement the generated shell code and run **Apache Ant** to build the **.rar** file.
9. Run the **Apache Ant** build tool to build the eWay **.sar** file.

After running the Apache Ant build tool, upload the newly created eWay **.sar** file to the ICAN Repository using Enterprise Manager and run the Enterprise Designer Update Center. You can now create, build, and deploy a new Project using the new eDK based eWay.

3.4.1 Step 1: Acquire an eDK eWay License File

A valid license file is required to upload your new eWay into the ICAN Repository using Enterprise Manger.

The ICAN licensing module permits the creation and distribution of non-standard eWays into the ICAN suite. SeeBeyond has set-up a centralized registration authority to ensure that newly created eWays do not contain names that conflict with existing products stored in the Repository. This guarantees the uniqueness of each registered and licensed eWay name.

Having a unique name means that any registered and licensed eWay can be safely installed into any ICAN Repository without future conflict. The granting of eWay names is on a “first come, first served” basis. Obtaining the eWay name and license should always be the first step when building an eWay

3.4.2 Licensing Process

The ICAN licensing module only allows one license (i.e. license.sar) to be active at a time; therefore to test or run a newly created eWay requires an updated license.sar. This is a once only process and subsequent requests for new eWay licenses will include any previously issued licenses. Note that although the license.sar file will be updated, the ProductsManifest.xml will not. You will still be able to upload custom eWays by using the browse button of any listed product to select the appropriate custom eWay **.sar** file, and selecting **Upload**.

The entire licensing process is conducted over e-mail. The user will initiate the process by sending a completed eWay license request e-mail to:

Licenserequest@SeeBeyond.com (detailed instructions in readme).

Only one e-mail request is allowed per eWay. When a new e-mail is received, the name is checked to ensure it is unique. If not, a reply is sent back explaining the situation. Once a unique name is found, then the fields in the body are validated and verified. Any incorrect field contents or missing fields result in the request being rejected. When a valid request is received, a license is generated for the eWay name submitted and a new license.sar is then sent to the requestor.

Note: *The readme.txt has more specific details on the exact process and the format of the e-mail messages used to request these licenses.*

3.4.3 Request e-mail Format Conventions

The body of each e-mail request must follow strict formatting rules. There can only be one field per line and each field must consist of a key that is followed by its value, and a format and help section.

The general format of each field is:

```
<key>=<value> (<field_data_format_name>: <any_addtional_qualifers>
[help=<directions to help fill out the field>])
```

An completed example is:

```
eWay.name=myeWay (string: free text [help=name of the eway])
```

Requesting a New License

Use the following checklist when requesting a new license:

- Verify the e-mail looks exactly as specified in the readme, (i.e. subject, body, and so on).
- Make sure all fields are filled out.
- Make sure all fields have the right data type, or has one of the enumerations.
- Make sure the customer ID is valid (check with Support if there are any doubts), and matches the name of the organization requesting the license.

If the requested name is already registered (i.e. a duplicate), then you could either try another requesting another name, or if you really would like that name a suggestion would be to prefix it with the name or abbreviation of the organization that will use it.

Steps to Request an eDK eWay License

To obtain a new eDK based eWay license:

1 Create a new e-mail with the header:

```
to:Licenserequest@Seeeyond.com
```

and with the subject:

```
eWay license request
```

2 For the body of the e-mail, cut and paste the following set of fields. Note that the format and help section is optional. Also, be sure each field below as a single line in your e-mail.

eWay.name= (string: free text [help=name of the eway])

eWay.target.OS.nameversion= (enumeration: Solaris 8 or 9,AIX 5.1 or 5.2, HP-UX 11.0, 11i (PA-RISC) or (11.23), Tru64 V5.1A, RedHat 8 or AS2.1, SUSE 8, Windows 2000, XP, or Server 2003, HP NonStop G06.22, ZOS 1.3 or 1.4 or ALL or subset [help=name or list of the OS the eway is to run on])

external.system.vendor= (string: free text [help=name of the vendor who made the external system])

external.system.name= (string: free text [help=name of the external system or technology e.g. a technology might be TCP/IP or CORBA])

external.system.version= (string: free text [help=version of the external system])

external.system.OS= (string: free text [help=the OS name of where the external system runs])

directionality= (emuneration: outbound, inbound, bi-directional [help=in relation to the eWay, the way the information flows in the interface])

description= (string: free text[help=a brief description of the functionality and interface mechanisms used by the eWay])
customer.ID.number= (string: free text [help=customer number from ONYX])

requestor.organization= (string: free text [help=name of organization that is requesting the eWay license, should be same as in ONYX])

requestor.name.first= (string: free text [help=first name of individual requesting the new eway license])

requestor.name.last= (string: free text [help=last name of individual requesting the new eway license])

requestor.email= (string: free text [help=email address of individual requesting the new eway license])

requestor.phone= (string: free text [help=phone number of individual requesting the new eway license])

The following is an example of a completed eDK eWay license request:

```
eWay.name=myTCPIPeWay (string: free text [help=name of the eway])
eWay.target.OS.nameversion=Solaris 9,AIX 5.2, RedHat AS2.1
(enumeration: Solaris 8 or 9,AIX 5.1 or 5.2, HP-UX 11.0, 11i (PA-RISC)
or (11.23), Tru64 V5.1A, RedHat 8 or AS2.1, SUSE 8, Windows 2000, XP,
or Server 2003, HP NonStop G06.22, ZOS 1.3 or 1.4 or ALL or subset
[help=name or list of the OS the eway is to run on])
external.system.vendor=N/A (string: free text [help=name of the
vendor who made the external system])
external.system.name=TCP/IP (string: free text [help=name of the
external system or technology e.g. a technology might be TCP/IP or
CORBA ])
external.system.version=N/A (string: free text [help=version of the
external system])
external.system.OS=Any (string: free text [help=the OS name of where
the external system runs])
directionality=bi-directional (enumeration: outbound, inbound,
bi-directional [help=in relation to the eWay, the way the information
flows in the interface])
description=This is a generic TCP/IP eWay that handles both inbound
and outbound connections. (string: free text[help=a brief description
of the functionality and interface mechanisms used by the eWay])
customer.ID.number=999999 (string: free text [help=customer number
from ONYX])
requestor.organization=The Company (string: free text [help=name of
organization that is requesting the eWay license, should be same as in
ONYX])
requestor.name.first=Joe (string: free text [help=first name of
individual requesting the new eway license])
requestor.name.last=Smith (string: free text [help=last name of
individual requesting the new eway license])
requestor.email=jsmith@thecompany.com (string: free text [help=email
address of individual requesting the new eway license])
requestor.phone=222-222-3456 (string: free text [help=phone number of
individual requesting the new eway license])
```

Requesting a Current License to be ReIssued

Use the following checklist when requesting a re-issue of a license:

- Verify that the e-mail looks exactly as specified in the readme i.e. subject, body etc.
- Make sure all fields are filled out
- Make sure that all fields have the right data type or has one of the enumerations
- Make sure that the date of the license is the same as the one used in the response to the request for a new license (it is possible to request a previous version of the license as long as the date is correct), or that if the latest license is requested that the date supplied for the dated issued field is the same date as the date the e-mail was generated.

Steps to Reissue an eDK eWay License

To reissue an eDK based eWay license:

1. Create a new e-mail with the header:

```
to:Licenserequest@Seeeyond.com
```

   and with the subject:

```
Reissue license
```

2. For the body of the e-mail, cut and paste the following set of fields. Note that the format and help section is optional. Also, be sure each field below as a single line in your e-mail.

```
eWay.name= (string: free text [help=name of the eway])

customer.ID.number= (string: free text [help=customer number from
ONYX])

requestor.organization= (string: free text [help=name of organization
that is requesting the eWay license, should be same as in ONYX])

requestor.name.first= (string: free text [help=first name of
individual requesting the new eway license])

requestor.name.last= (string: free text [help=last name of individual
requesting the new eway license])

requestor.email= (string: free text [help=email address of individual
requesting the new eway license])

requestor.phone= (string: free text [help=phone number of individual
requesting the new eway license])

reissue.reason= (string: free text [help=the reason why a re-issue is
being requested])

license.issue.date= (date: yyyy/mm/dd [help=the date the license that
is being requested was originally issued on or use same date as the
e-mail request to specify the latest license])
```

The following is an example of a completed eDK eWay license reissue request:

```
eWay.name=myTCPIPeWay (string: free text [help=name of the eway])
customer.ID.number=999999 (string: free text [help=customer number
from ONYX])
requestor.organization=The Company (string: free text [help=name of
organization that is requesting the eWay license, should be same as in
ONYX])
requestor.name.first=Joe (string: free text [help=first name of
individual requesting the new eway license])
requestor.name.last=Smith (string: free text [help=last name of
individual requesting the new eway license])
requestor.email=jsmith@thecompany.com (string: free text [help=email
address of individual requesting the new eway license])
requestor.phone=222-222-3456 (string: free text [help=phone number of
individual requesting the new eway license])
reissue.reason=can't find file (string: free text [help=the reason
why a re-issue is being requested])
license.issue.date=2000/01/01 (date: yyyy/mm/dd [help=the date the
license that is being requested was originally issued on or use same
date as the e-mail request to specify the latest license])
```

3.4.4 Step 2: Set the Environment Variables

An implementation environment is required to compile Java source files and to build an eWay **.sar** file.

To set the environment variables for the implementation environment:

1. Open a new command line window.
2. Locate the root directory of the extracted eDK folder (e.g. C:\eDK), and run the **env.bat** file.

3.4.5 Step 3: Start the eWay Development Kit Build Tool

To start the eWay Development Kit build tool:

1. From the same command window change directories to:

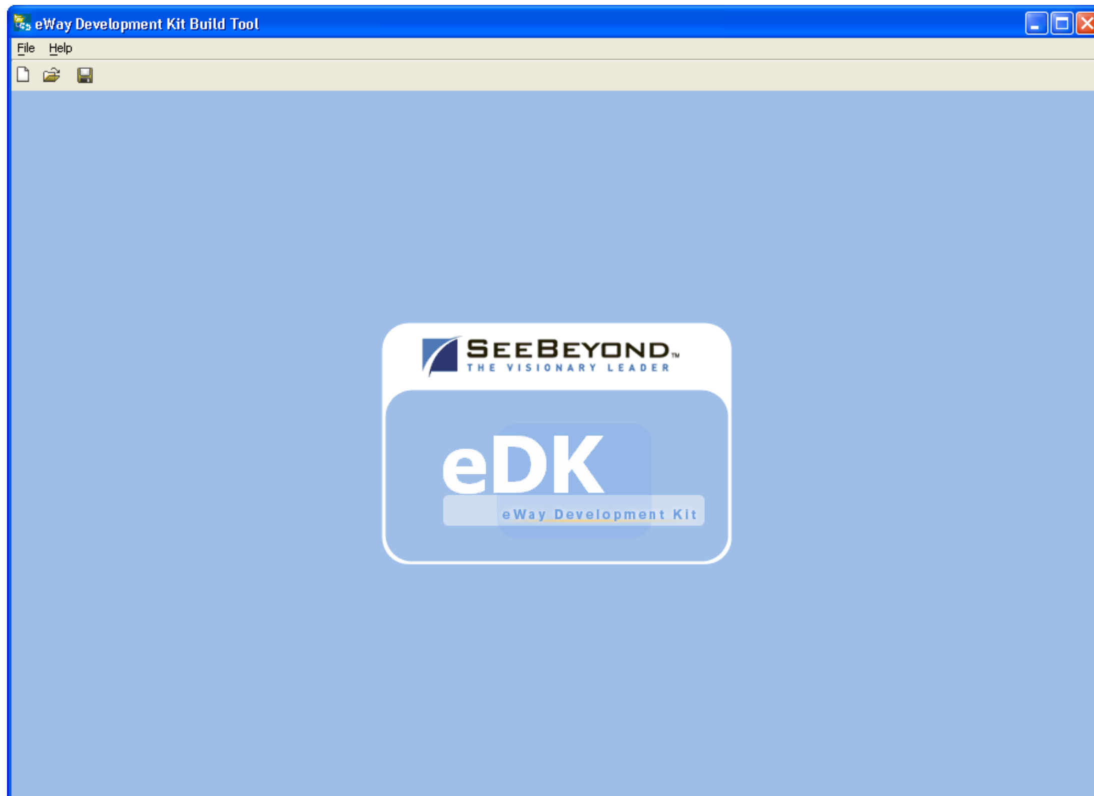
```
<STC ROOT>\eways\edk\devtools
```

2. Run the following command:

```
ant runedkgui
```

The eWay Development Kit Build Tool appears.

Figure 3 eWay Development Kit Build Tool



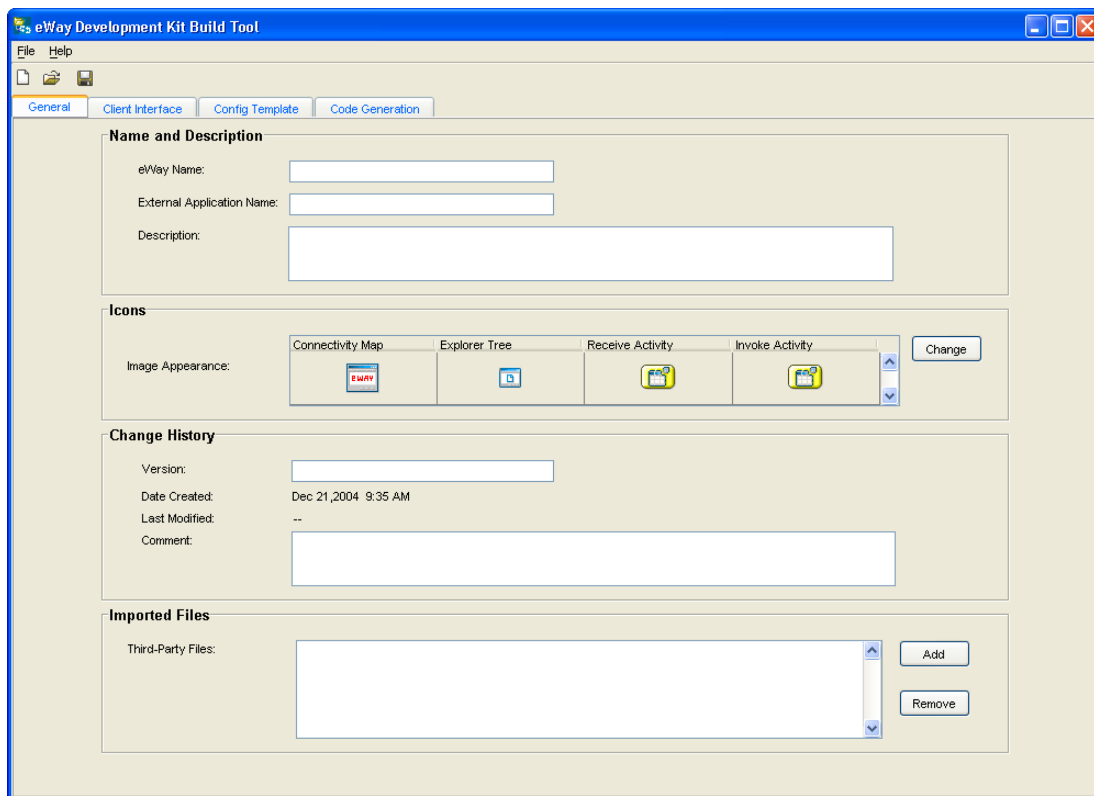
3.4.6 Step 4: Create and Specify the New eWay

Perform the following steps to create a new eWay:

1. From the menu bar, select **File** then select **New eWay**, or click the **New eWay** icon. An empty eWay template appears in the build tool window, see Figure 4.
2. Set up the new eWay by entering or setting the following features on the General tab on the eWay Development Kit Build Tool.
   - Name and Description
   - Icons
   - Change History
   - Imported Files

Figure 4 illustrates the features found on the General tab of the eWay Development Kit Build Tool.

Figure 4 eWay Development Kit Build Tool - General Tab



Name and Description

The Name and Description section contains the following fields:

- **eWay Name** – enter a name that identifies the new eWay name. This field is required to generate the eWay.
- **External Application Name** – enter a name that identifies the external applicaton. The name you enter appears in Enterprise Designer. This field is required to generate the eWay.
- **Description** – enter a description of the new eWay. The description can be used by the developer to enter notes about the eWay.

Naming Restrictions:

The following standard Java Identifier naming conventions apply to the eWay Name and External Application Name:

- **Alphabetic letters** – names should only contain alphabetic characters, such as "AA" or "aa", or the underscore.
- **Digit** – names can only contain numbers between zero and nine (0 - 9). Also, the first position of the name should not contain a digit.

Note: *Using an underscore in the External Application Name causes the eWay component to not appear on Enterprise Monitor Logging Control.*

Icons

- **Image Appearance** – alters the appearance of eWay icons on the Connectivity Map, Explorer Tree, or the Receive or Invoke web service activities. It is required to have a valid icon file location.

  To change an eWay icon:

  A Click **Change**, and browse to the location of a suitable file. File types can be either **.jpeg**, **.tif**, **.gif**, **.bmp**, **.jfif**, **.png**, **.jpg**, **.tiff**, **.jpe**, or **.dip**.

  B Click **Open**. The new icon now appears in the Image Appearance box.

  The maximum size for icons used are:

Table 2 Maximum Icon Size Accepted

Image Appearance Icons	Size in Pixels	Appears On
Connectivity Map	32 x 32	Enterprise Designer Connectivity Map
Explorer Tree	20 x 20	Project Explorer, the Connectivity Map, and the External Application
Receive Activity	32 x 32	eInsight Canvas
Invoke Activity	32 x 32	eInsight Canvas

Change History

- **Version** – enter a version number to identify the latest version number of the eWay. The version number appears on the **ADMIN** tab of the **Enterprise Manager**, and in the **Update Center Wizard** of the **SeeBeyond Enterprise Designer**. This field is required to generate the eWay.
- **Date Created** – identifies the eWay creation date. This field is read-only, and is automatically created by the build tool.
- **Last Modified** – identifies the date and time of the last saved eWay modification. This is a read-only field that is automatically created by the build.
- **Comments** – is used to enter comments each time you create or update the eWay.

Imported Files

- **Third-Party Files** – is used to import all custom third-party **.jar** files required for implementation of eWay shell code generated by the eDK.

  To import a file:

  A Click **Add**, and browse to the location of the **.jar** file.

  B Click **Open**. The name and location of the executable **.jar** file appears in the textbox.

  To remove a file, select a third-party file from the Third-Party text box and click **Remove**.

Note: *Code generation copies these .jar files to the implementation environment.*

3.4.7 Step 5: Enter the Required eWay Client Interfaces

In the eWay Development Kit, the Client Interfaces represent the methods, user defined operations, and attributes exposed to the eWay user. The Client Interface also represents the eWay's OTD.

Client interfaces can contain both inbound and outbound JCE (J2EE Connector Architecture) and BPEL (Business Process Execution Language) components.

Defining Your eWay's Java Interface

Defining a eWay's Java Client Interface, involves defining the Java methods and attributes. The getter and setter methods are automatically added when a attribute is defined. These attributes can be regular JDK Classes or User-Defined classes. User-Defined classes created with the eDK Build Tool correspond to actual Java classes exposed in the JCE.

Table 3 illustrates how JCE Interfaces created in the eDK Build Tool appear in the Enterprise Designer.

Table 3 Java Interface Comparison

Client Interfaces Appearing in the eWay Development Kit Build Tool:	How these Client Interfaces Appear in Enterprise Designer:
Methods (JCE only)	OTD Methods
User Defined (JCE Java Classes)	As part of the OTD
Attribute (JCE only)	OTD getters and setters

Defining Your eWay's BPEL Interface

Unlike the JCE where the interface defined in the eDK Build Tool correspond to actual Java classes, the BPEL Interface is primarily used to create the WSDL (Web Services Description Language) element. The Java code that the eWay Development Kit generates is a shell for implementing the service defined in the WSDL. This generated code includes methods corresponding to operations, and wrapper classes for user-defined data containers.

Table 4 illustrates how BPEL Interfaces created in the eDK Build Tool appear in the Enterprise Designer.

Table 4 BPEL interface Comparison

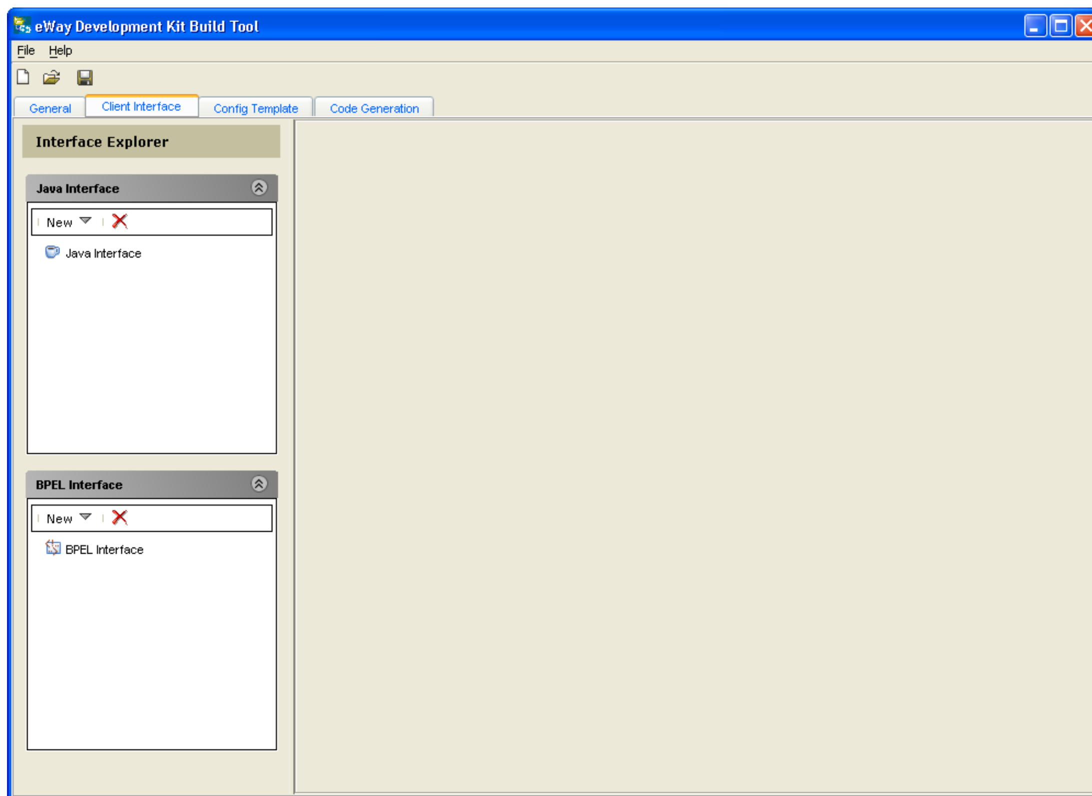
Client Interfaces Appearing in the eWay Development Kit Build Tool:	How these Client Interfaces Appear in Enterprise Designer:
User Defined (BPEL data containers)	Appear in the Business Rule Editor
Operations (BPEL only)	Appear in the Project Explorer and the Business Rule Designer.

Perform the following steps to create eWay Interfaces:

1 Select the Client Interface tab on the eWay Development Kit Build Tool. The Client Interface window appears.

   The Client Interface tab in Figure 5 includes an Interface Explorer for adding placeholders for new Java and BPEL based objects.

Figure 5 eWay Development Kit Build Tool - Client Interface Tab



2 From the Interface Explorer, click **New** and select one of the following:

- **Method** – to create inbound or outbound methods (JCE only).
- **User Defined** – to create inbound or outbound class types (data containers in BPEL)
- **Attribute** – to create inbound or outbound attributes (JCE only).
- **Operation** – to create inbound or outbound operations (BPEL only).

3 Enter the details for the selected Java or BPEL interface.

Naming Restrictions:

The following standard Java Identifier naming conventions apply to Method, Operation, and Attribute names:

- **Alphabetic letters** – names should only contain alphabetic characters, such as “AA” or “aa”, or the underscore.
- **Digit** – the first position of the name should not contain a digit.

Creating Methods (JCE)

You can use the eDK Build Tool to create inbound and outbound JCE methods that can take parameters, return a result, and throw exceptions. Methods can also return a User Defined Type, which can contain a primitive or complex data type or even another User Defined Type. Any method created is exposed in Java Collaboration Definitions.

To create a new method:

1. From the Java Interface, enter a new method name in the **Method Name** field.
2. Select a return type for the method from the **Return Type** drop-down box.
3. Select the **Collection Type** checkbox to specify that the type of object returned is a collection.
4. Click **Add** to create a new parameter in the **Parameter List**.
5. Enter or select an exception to throw in the **Throw Exception** combo-box.
6. Enter a description for the exception in the **Description** textbox.

Creating User Defined Class Files

Classes have attributes (data) and operations (behaviors). Class attributes are implemented in Java programs as fields, while class behaviors are implemented as methods.

You can use the eWay Development Kit Build Tool to define new inbound or outbound classes. You can also choose to use specific classes by importing third-party **.jar** files. Only one class file can be selected at a time for each User Defined class.

BPEL Versus JCE User Defined Class Files

BPEL user-defined classes are referred to as “user defined data containers”. A data container represents the WSDL element “**Type**” that is generated for the eWay's BPEL interface. BPEL user-defined classes are data containers only; they are bean classes that only contain setter or getter methods for attributes. The internal implementation may contain additional private methods or attributes but what is seen by the eWay user is only what is described in the WSDL. For additional information on WSDL files and BPEL, see **“BPEL Operations in ICAN” on page 30**.

For JCE, the classes defined in the eDK Build Tool can be instantiated directly in the Java Collaboration Editor. Wrapper classes are not automatically created. You can call any public method since they are not just data containers.

To create a new User Defined Class:

1. Enter a new name in the **Name** field.

To use a third-party **.jar** file:

2. Select the Use **Third-Party JAR** file checkbox.
3. Select an available **.jar** file from the drop-down list if you previously added one on the **General** tab. If a **.jar** file does not appear, then select **Browse** from the drop-down list and locate one.

   New **.jar** files added to the user defined type are also added to the **Imported Files** section on the **General** tab.
4. Click **Add**. The **Third-Party JAR** file window opens containing class files organized by package.
5. Locate the required class file and click **Select**. The Class file with available attributes appears in the **Attributes List**.

Creating Attributes

JCE Attributes are data fields that describe an object's characteristics. For example, an attribute may be the "capacity" of the elevator class, or it may describe the state of an object, such as the "isMoving" of the car class. Attributes have a name and a type. Data types can be either primitive or a user defined type (such as a previously created class).

The Client Interface tab in Figure 5 includes an Interface Explorer to add placeholders for new Java based Attributes.

To create a new Java Attribute:

1. From the Java Interface, enter a new attribute name in the **Attribute Name** field.
2. Select a data type for the attribute from the **Type** drop-down list.
3. If the attribute is a collection, select the **Collection** checkbox.

Creating Operations

The BPEL Interface represents the operations that make up your web service. The operations entered into the eDK Build Tool appear are used to construct the operation element of the WSDL. This includes the operation's input, output, and fault message.

To create a new BPEL Operation:

1. From the BPEL Interface, enter a new Operation name in the **Operation Name** field.
2. Select a return type for the Operation from the **Return Type** drop-down list.
3. If the return type is a collection, select the **Collection** checkbox.
4. Click **Add** to create a new parameter in the **Parameter List**.
5. Enter or select an exception to throw in the **Throw Exception** combo-box.
6. Enter a description for the exception in the **Description** textbox.

BPEL Operations in ICAN

BPEL for web services is an xml-based language designed to enable task-sharing for a distributed computing environment for Java developers to publish web services and compose them into reliable and transactional business flows.

BPEL is designed to keep internal business protocols separate from cross-enterprise protocols so that internal processes can be changed without affecting the exchange of data from enterprise to enterprise. This means that any programmer using BPEL can formally describe a business process in such a way that any cooperating entity can perform one or more steps in the process the same way.

The standardization of communications protocols and message formats makes it increasingly important to structure the way communications are described. WSDL addresses this need by defining an XML grammar for describing network services as collections of communication endpoints capable of exchanging messages.

According to a W3C note dated March 15, 2001, a WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service.

For additional information on WSDL see:

http://www.w3.org/TR/wsdl#_introduction

By design, BPEL operations must follow WSDL specifications:

There are four types of web service operations, or transmission primitives that an endpoint can support:

- **One-way** – where the endpoint receives a message.
- **Request-response** – where endpoint receives a message, and sends a correlated message.
- **Solicit-response** – where the endpoint sends a message, and receives a correlated message.
- **Notification** – where the endpoint sends a message.

In the case when one or more BPEL operations are defined by the user in the build tool, an WSDL definition is created with the following elements:

- **port** – which specifies an address for a binding, thus defining a single communication endpoint.
- **portType** – which is a set of abstract operations. Each operation refers to an input message and output messages.
- **message** – which represents an abstract definition of the data being transmitted. A message consists of logical parts, each of which is associated with a definition within some type system.
- **types** – which provides data type definitions used to describe the messages exchanged.

Note that while **binding** and **service** elements used in the definition of network services, they are not listed above since they are resolved at runtime by the ICAN’s BPEL engine.

Additional Development Notes:

- BPEL operations are web service operations that should only be stateless with at most one input, output and fault message. The corresponding Java classes, either generated by the eDK build tool or by using a third-party class, must follow the Java bean conventions.
- BPEL operations can be either inbound or outbound (receive and invoke activities). Inbound operations and outbound operations are included in separate portypes in the WSDL file, with corresponding java classes generated for each port type.
- Parameters that are passed into the operation must contain all the field attributes to be displayed on the eInsight BPEL attribute mapper. This means that any BPEL "attribute" must be defined in the User Defined data type which is input to the operation.

3.4.8 Step 6: Define the eWay Configuration Template

The eWay Development Kit Build Tool's **Config Template** tab is used to create a configuration template for an eWay. A configuration template is a hierarchical based model—represented as an **.xml** file—that contains a superset of configuration parameters defined within sections and subsections.

Configuration parameters can be set to contain a number of eWay specific properties to be edited in either the Enterprise Designer **Connectivity Map**, or the **External System** eWay Environment properties.

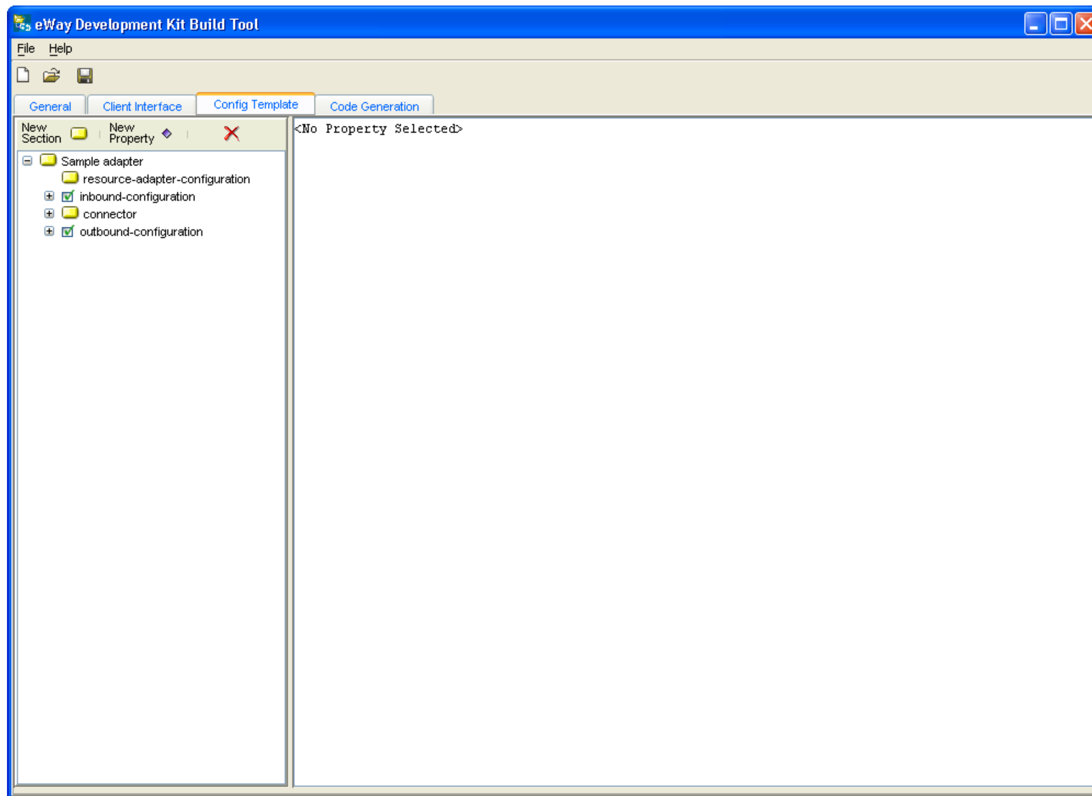
It is up to the eDK developer to expose the necessary sections and properties for the eWay user.

Note: *Additional information on specifying configuration properties are found in* **"Specifying Configuration Properties" on page 59**

To add new sections and properties to the Configuration Template:

1. Select the **Config Template** tab on the eWay Development Kit Build Tool. The Config Template window appears.

Figure 6 eWay Development Kit Build Tool - Config Template Tab



2. Expand the **inbound-configuration** and **outbound-configuration** sections in configuration template tree.

3. Click on the **New Section** icon to add new sections. Click the **New Property** icon to add new properties. Alternately, you can also right-click a section and create a new section or property with the pop-up menu. An example of creating a new property by right-clicking a section is seen in Figure 7.

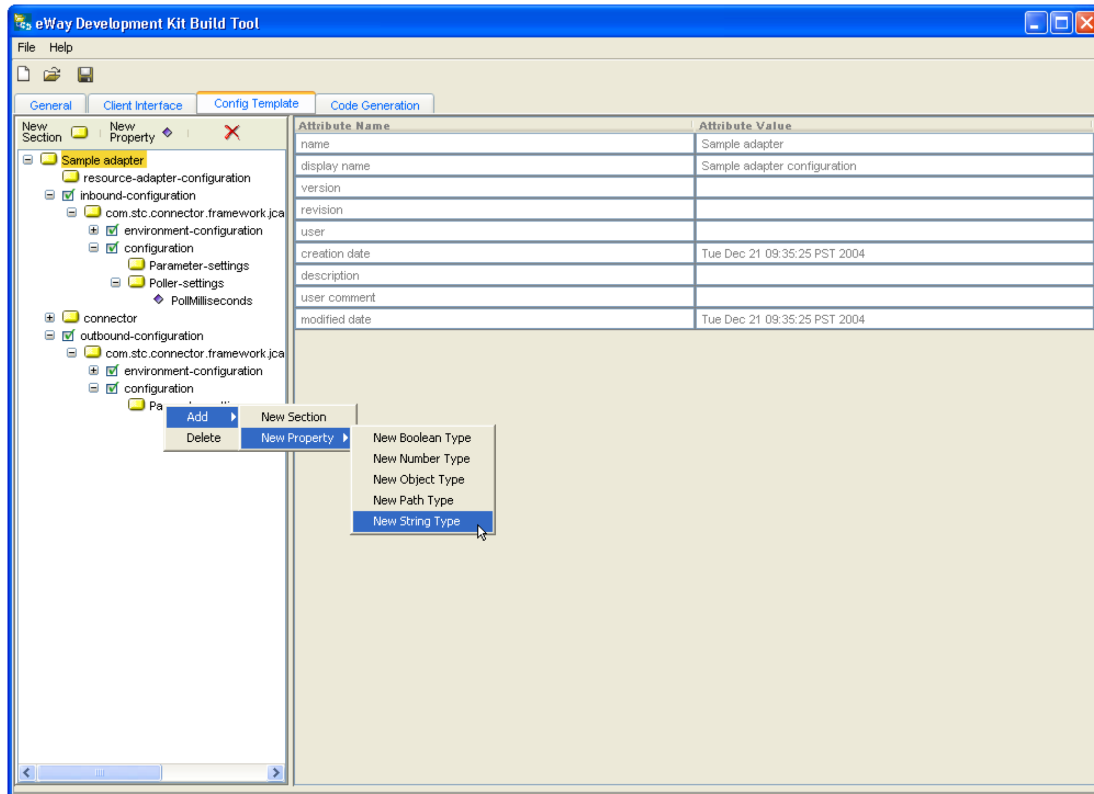
Note: *Enter only those sections that the eWay supports. For example, if the eWay only supports inbound, then do not enter sections for outbound. You can disable sections that are not supported, (see* **"Deleting Sections and Properties" on page 37***, or* **"Disabling and Enabling Sections" on page 37** *for more information).*

Naming Restrictions:

The following standard Java Identifier naming conventions apply to the Section and Property names:

- **Alphabetic letters** – names should only contain alphabetic characters, such as "AA" or "aa", or the underscore.
- **Digit** – the first position of the name should not contain a digit.

Figure 7 Config Template Tab - Adding a New Property



New Sections Contain the following default Attributes:

Table 5 Default Section Attributes

Section	Value
name	The name displayed in the **.java** files during implementation. Attribute names should follow standard Java naming conventions.
display name	The name that appears on the Connectivity Map and the Environment Explorer.
description	The description of the section that appears in Enterprise Designer.

New Parameters Contain the following default Attributes:

Table 6 Default Parameter Attributes

Section	Value
name	The name displayed in the **.java** files during implementation. Attribute names should follow standard Java naming conventions.
display name	The name that appears in the Connectivity Map or the External Properties window in Enterprise Explorer.
description	The description of the section that appears in Enterprise Designer.
default	The default value for the parameter.
is choice*	Determines if a choice is possible for the default value. To enter a new choice attribute: 1 Select True from the **is choice*** attribute value. The Attributes window appears. 2 Enter a choice value and click **OK** or the **Tab** button.
is choice editable	Determines if the choices entered for the **is choice*** attribute values are editable at design time.
type	Refers to the data type of the parameter (see below for more details).

New Parameters can be of the following type:

Table 7 New Parameter Types

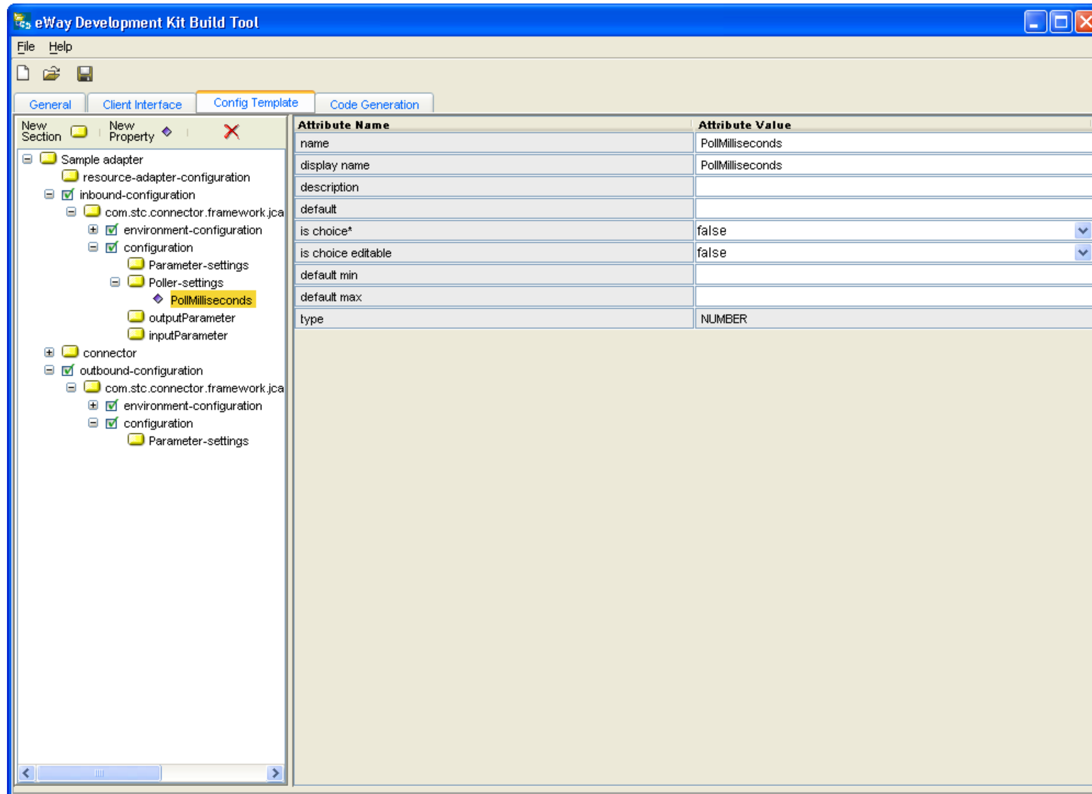
Section	Additional Attribute Values
Boolean (BOOLEAN)	
Number (NUMBER)	Includes the additional default attributes: ▪ **default min** – includes the default minimum value. ▪ **default max** – the default maximum value.
Object (OBJECT)	
Path (PATH)	
String (STRING)	Includes the additional default attributes: **is encrypted** – determines if the string is encrypted. Note: Do not include **is choice*** or **is choice editable** when using **is encrypted**.

4 Highlight and change the Attribute-Value for each new section, then click **Return** to save your changes.

You can remove any inbound or outbound sections not required in the eWay by clicking the Delete icon. Any configuration section that appears in the configuration template also appears in the completed eWay, even if no values are entered for them.

The example in Figure 8 below illustrates the creation of the “inputParameter” and “outputParameter” sections, in addition to “Poller-settings” containing a property called “PollMilliseconds”.

Figure 8 outputParameter set in inbound-configuration



Connectivity Map Configuration Sections and Properties

Connectivity Map configuration properties added to the **Config Template** tab appear in the eDK eWay’s Connectivity Map **Properties** window in Enterprise Designer.

Inbound and Outbound Configuration Parameter Settings

Configuration parameter settings can include both inbound and outbound.

Inbound Configuration properties—and subsequent sub-sections—to a Connectivity Map link attached to an inbound eWay, are located under the configuration template's inbound configurations subsection at:

```
root > inbound-configuration > com.stc.connector.framework.jca.
system.STCActivationSpec > configuration
```

Outbound Configuration properties—and subsequent sub-sections—to a connectivity map link attached to an outbound eWay, are located under the configuration template's outbound configurations subsection at:

```
root > outbound-configuration >
com.stc.connector.framework.jca.system. STCManagedConnectionFactory >
configuration
```

External System Sections and Properties

External System properties added to the **Config Template** tab appear in the eDK eWay's External System **Properties** window in the Environment Explorer tab of Enterprise Designer.

Inbound and Outbound External System Parameter Settings

External System parameter settings can include both inbound and outbound.

Inbound Configuration properties—and subsequent sub-sections—to an External System, are located under the configuration template's inbound environment-configurations subsection at:

```
root > inbound-configuration > com.stc.connector.framework.jca.
system.STCActivationSpec > environment-configuration
```

Inbound Configuration properties—and subsequent sub-sections—to an External System, are located under the configuration template's outbound environment-configurations subsection at:

```
root > outbound-configuration >
com.stc.connector.framework.jca.system. STCManagedConnectionFactory >
environment-configuration
```

Deleting Sections and Properties

Unwanted sections can be deleted from the Config Template tab by either right-clicking the target section and selecting Delete from the pop-up menu, or by clicking the Delete icon located in the tool bar.

Note: *You cannot delete root level sections such as "Sample Adapter" or "Configuraton".*

Disabling and Enabling Sections

Depending on the eWay requirements, you may also choose to disable certain sections. Disabling a section on the Config Template tab hides the details of that section during code generation, and preventing them from appearing in the eDK eWay's External System and Connectivity Map Properties window in the Enterprise Designer.

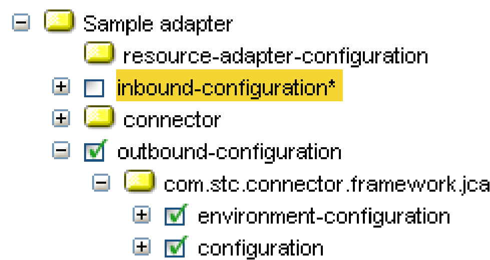
A small green checkbox icon appears beside sections that can be disabled on the configuration template tree. Sections that are disabled appear with an unchecked icon and also contain an asterisk after the section name.

Any section that is disabled can also be enabled. Enabling a disabled section restores the full functionality to that section, including any additional sections or properties that were previously added.

To disable a section on the configuration template tree:

1. Right-click on a section, as for example the "inbound-configuration" section. A pop-up menu appears next to the section.
2. Click **Disable**, and click **Yes** on the Confirm Deletion window that appears. The disabled section appears collapsed and displays an unselected checkbox icon next to it.

Figure 9 Disabled inbound-configuration Section

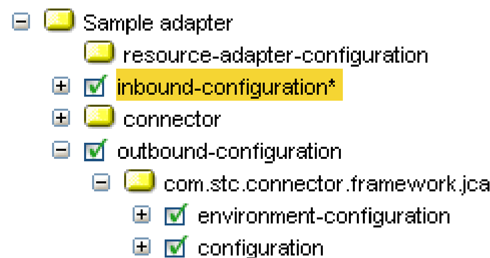


To enable a section on the configuration template tree:

1 Right-click on a previously disabled section. A pop-up menu appears next to the section.

2 Click **Enable**. A selected checkbox icon appears next to the enabled section.

Figure 10 Enabled inbound-configuration Section



3.4.9 Step 7: Run the Code Generator

The Code Generation tab on the eWay Development Kit is used to generate the "connectors" and "eways" folders at a specified location.

To generate code:

1 Select the **Code Generation** tab on the eWay Development Kit.

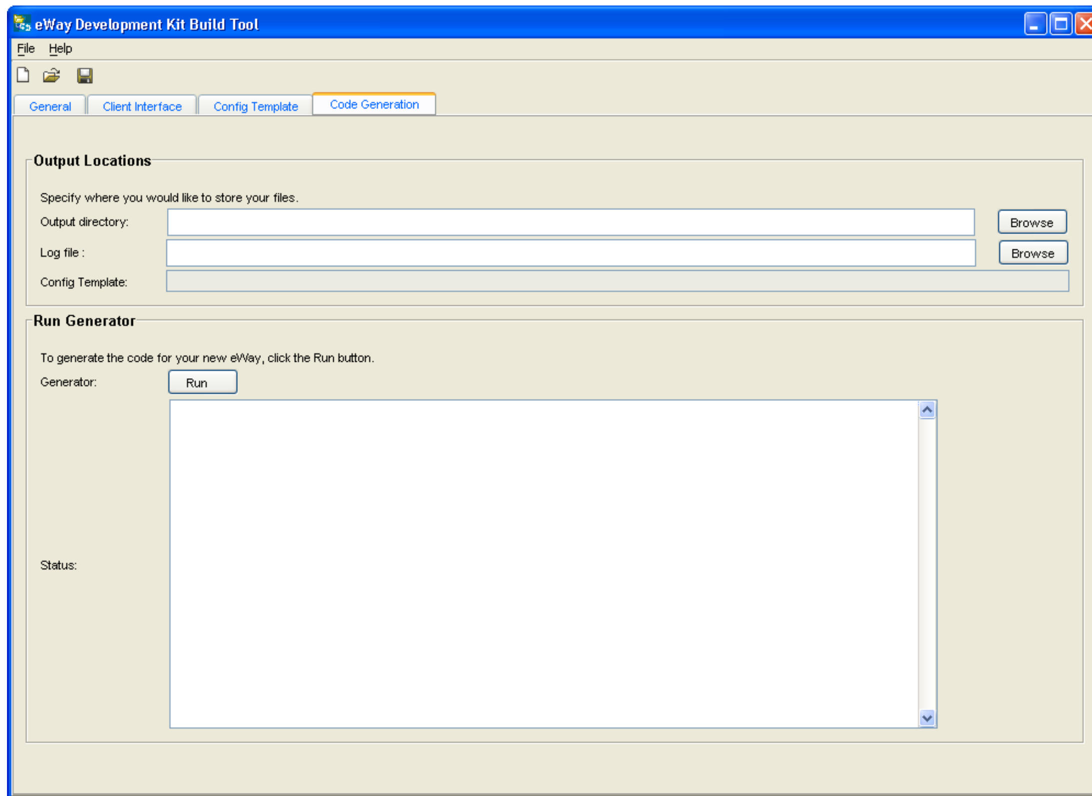
2 Enter or browse to an output directory. This is the location where the "**connectors**" and "**eways**" folders are created, (see **"eWay Folders Created After Shell Code Generation" on page 41**).

3 Enter or browse to an output directory for the log file.

4 Click **Run** from the Run Generator frame.

Note: *You must enter both an eWay Name and an External Application Name to generate eWay code.*

Figure 11 Generator Output Tab of the eWay Development Kit Build Tool



Saving Your Work

You can save your work at any time by clicking **Save** or **Save As** in the file menu, or by clicking the Save eWay icon. An eWay name must be provided prior to saving.

Saving creates the following files:

- **<adapter_name>.xdef** – represents the eDK definition file that stores metadata information about the eWay. For more information on the fields generated in this file, see **"eDK Definition File" on page 102**.
- **<adapter_name>_template.xml** – represents the configuration template which is a hierarchical based model that contains configuration parameters for the eWay. For more information on the parameters described in the configuration template, see **"Connectivity Map Configuration Sections and Properties" on page 36** and **"External System Sections and Properties" on page 37**.

Opening Previously Saved Work

You can open previously saved work by clicking **Open** in the file menu or by clicking the Open eWay icon, then browsing and selecting a previously saved **<adapter_name>.xdef** file.

Choosing a Working Directory

As an optional step, you can choose to either work from the generated output folder, or choose a new working directory by copying the following folders:

- Copy the **<newEway>adapter** folder, located under the **connectors** folder in the output directory (as specified in the definition file) to:

  ```
  <STC_ROOT>\connectors\
  ```

- Copy the **<newEway>adapter** folder, located under the **eways** folder in the output directory (as specified in the definition file) to:

  ```
  <STC_ROOT>\eways\
  ```

3.4.10 Step 8: Implement and Build the Generated Shell Code

The following steps describe how to modify and implement the shell code generated by the eDK.

1. Browse to and copy **<newEway>adapter**, located under the **connectors** folder in the output directory to:

   ```
   <STC_ROOT>\connectors\
   ```

2. Modify the **<external_application>EWayConnection** class for your specific eWay implementations.

3. Implement the **<external_application>ClientApplicationImpl** class under the **appconn\appimpl** sub-folder, and add new classes there if necessary.

4. Implement the **<external_application>WebClientApplication** class under webservice sub-folder, and add new classes there if necessary.

5. Browse to the ${**env.STC_ROOT}\connectors\<newEway>adapter** folder, and run the following:

   ```
   ant clean install -f connector-build.xml
   ```

   This should build the **<newEway>.rar** file and all the other required jar files at the following locations:

   ```
   <STC_ROOT>\BUILD\Modules\connectors\lib\<newEway>.rar
   This is a JCA 1.5 compliant .rar file.

   <STC_ROOT>\BUILD\Modules\connectors\lib\<newEway>_jca10.rar
   This is a JCA 1.0 compliant .rar file.
   ```

3.4.11 Step 9: Build the .sar File

You must run the **Apache Ant** build tool to build the **.sar** file.

1. Browse to eWay working directory that you defined in **“Choosing a Working Directory” on page 40**, then run the following:

   ```
   ant clean install -f eway-build.xml
   ```

   This creates the new **<newEway>adapter.sar** file in the following location:

   ```
   <STC_ROOT>\BUILD\images\products\<newEway>adapter.sar
   ```

3.5 eWay Folders Created After Shell Code Generation

Two new folders containing the eWay shell code are created in the specified output directory.

- connectors folder
- eways folder

connectors Folder

The connectors folder contains the J2EE connector code. Most of the implementation required are contained in the connectors folder.

Both the "src" and "src_jca15" or only the “src” subdirectories are generated in the connectors directory, depending on whether or not there is inbound operations defined.

The **src_jca15** folder is only created if there are inbound operations defined.

The “src” folder contains the following:

- **alerts folder**–The alert subfolder contains one java file, namely <External_application_name>AlertCodes.java.

  This file should define standard and/or customized alert codes. All generic alert codes to be displayed in Enterprise Monitor are already defined there; however, the user can always add more customized alert codes. For details of how to implement alert codes, see **“eDK Alerts” on page 89**.

- **appconn folder** – This subfolder contains all the necessary shell code for application specific implementations.

  - **<External_application_name>ApplicationConnection.java** implements the ApplicationConnection interface which handles EIS connections for the application. This interface can be viewed to represent the EIS connection handle for the application connection. It allows implementations to create an actual application connection.
  - **<External_application_name>ApplicationException.java** provides the exception class for any application specific exceptions.
  - **<External_application_name>ClientApplication.java** is the interface class that contains all user defined interactions with the underlying EIS. Note that all user defined outbound operations to be made available in SeeBeyond's Java Collaboration Editor are listed here.
  - **<External_application_name>Configuration.java** is the bean class that allows accessing of each and every user defined outbound configuration parameters. Note that this class only contains all outbound configurations.

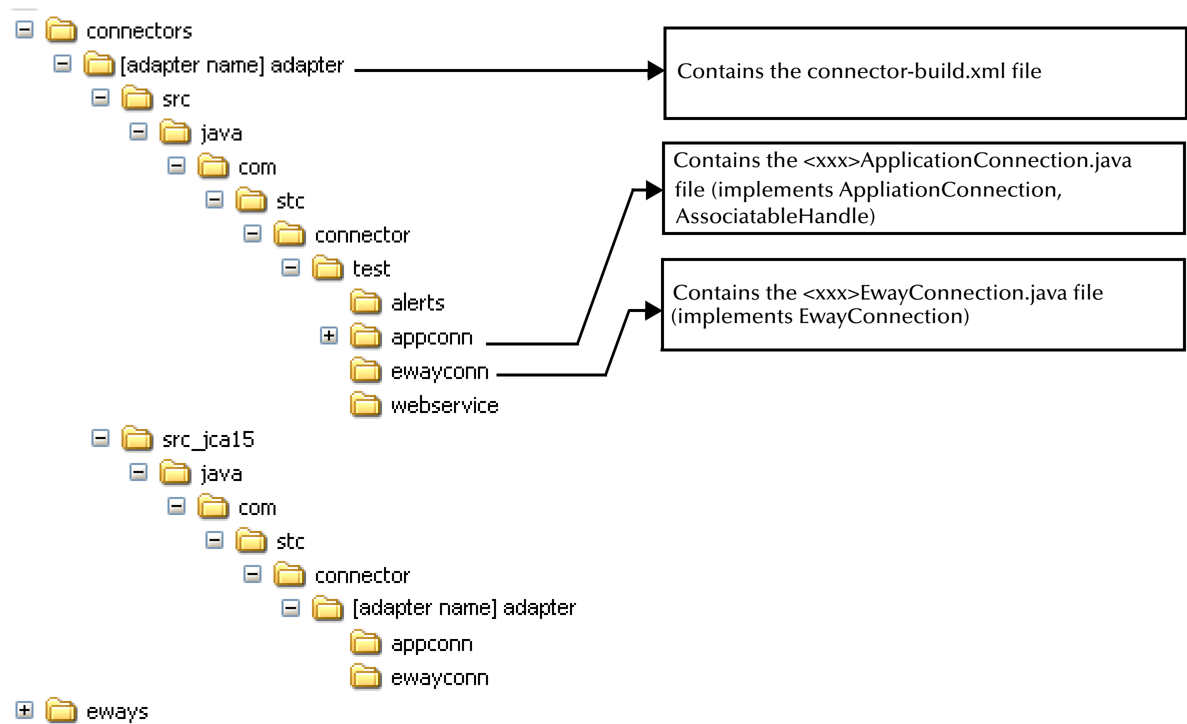
- **appimpl folder** – Contains the implementation class for the <External_application_name>ClientApplication.java interface defined in the appconn folder. This is also where user needs to make the actual application specific implementations for the eway. The user can choose to add more classes in this folder also if necessary. Generated bean classes for any user defined datatypes for the Java Collaboration Interface are also placed here.

- **ewayconn folder** – Contains **<External_application_name>EwayConnection.java** which implements the EwayConnection interface. This class allows implementations to establish and close connection to external EIS system, to match an existing connection, or to clear or release resources prior to the destruction of managed connection, and so on.
- **webservice folder** – This subfolder contains all the generated bean classes for the user-defined data types for webservice operations. These classes follow the java bean paradigm and also implements the necessary persistence methods. This folder also contains the <External_applicaiton_name>WebClientApplication.java file, which contains empty methods for all the defined outbound webservice operations to be implemented by the user.

The "src_jca15" folder contains the following:

- **appconn folder** – This subfolder contains all the necessary java files for inbound communications, including:
  - **<External_application_name>InboundConfiguration.java** is the bean class that allows accessing of each and every user defined inbound configuration parameters. Only accessor methods for inbound configurations are available in this class.
  - **<External_application_name>Listener.java** provides the interface methods for each inbound operation. This is the interface the MDB needs to implement.
- **ewayconn folder** – This subfolder contains all necessary classes to provide inbound connectivity from an EIS system.
  - **<External_application_name>EwayActivationSpec.java** implements methods necessary to represent inbound connectivity information from an EIS instance to an application via a specific resource adapter instance. Other java classes are included here for the support of inbound Work Tasks. If necessary, the user can choose to add more classes in this folder.

Figure 12 Files Found in the connectors Folder

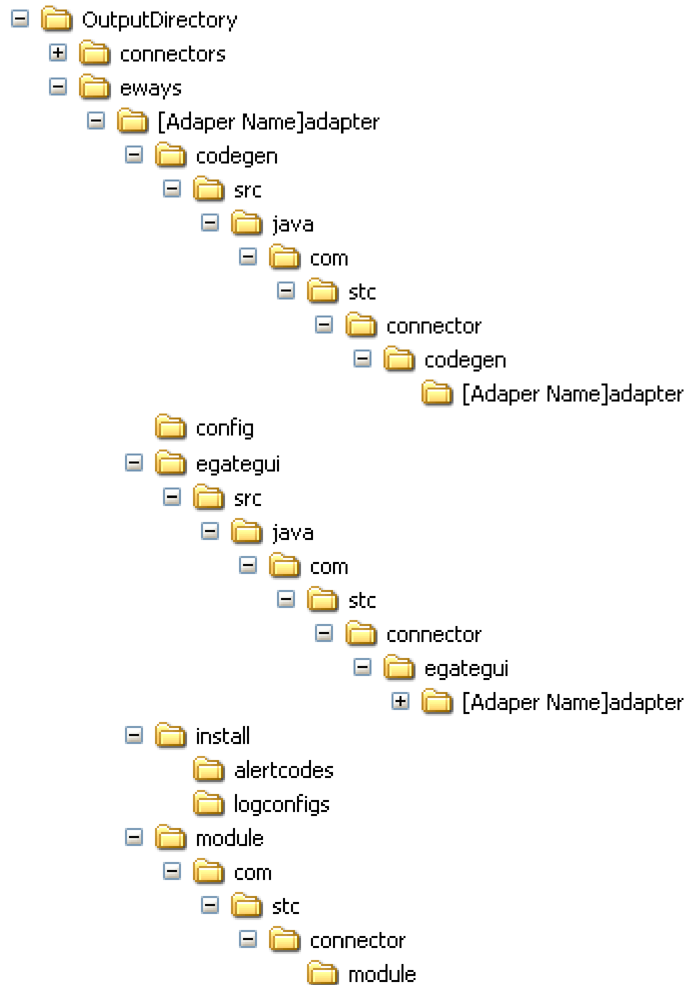


eways Folder

The **eways** folder contains the GUI code used to plug into the ICAN Enterprise Designer code generation components, including codelets, and installation descriptors.

- **codegen** – contains all the codelets and runtime EJBs.
- **egategui** – contains all the GUI plug-in code for the different eDesigner editors.
- **config** – contains the configuration template.
- **install** – contains the WSDL file, descriptor.xml for eway installation, and also logging and alerting properties files.
- **module** – contains the **manifest.mf** and **layer.xml** files for the eWay Netbeans module.
- **Thirdpartylib** – contains all user specified third-party **.jar** files.

Figure 13 Files Found in the build-tools Folder



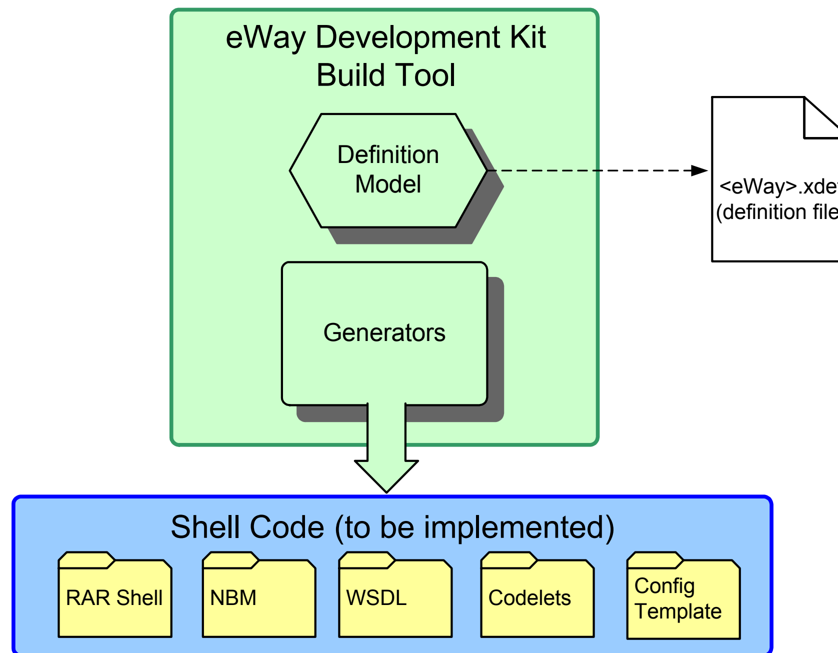
3.5.1 eWay Code Created After Generation

The code generated by the eDK includes the following components:

- J2EE Connector Architecture resource adapter
- GUI code for plugging into the Enterprise Designer
- Code Generation components
- Runtime EJB components

All of these components are packaged in the eWay **.sar** file that can be uploaded into an ICAN repository. The following diagram illustrates the shell code components created after code generation.

Figure 14 Shell Code Created After Generation



J2EE Connector Architecture Resource Adapter

The J2EE specification includes a Connector Architecture specification defining the component which is used to interact directly with external applications (also referred to as Enterprise Information Systems - EISs). This component is normally referred to as a J2EE connector or Resource Adapter (RA).

SeeBeyond developed a framework for developing these resource adapters. The classes generated by the eDK are based on the SeeBeyond RA framework. The framework provides a set of interfaces and abstract classes which simplify the development of J2EE Resource Adapters. It allows the eWay developer to focus on implementing the client interface that needs to be exposed to the eWay user.

Figure 14 on page 45 illustrates how Resource Adapter Archive (**RAR Shell**) Code fit into the eDK component process flow.

GUI code for plugging into the Enterprise Designer

To be able to use an eWay on the Enterprise Designer, the corresponding GUI plug-in code must be provided in the eWay **.sar** file. Installation of the eWay's GUI plug-in first requires uploading the eWay **.sar** file to the ICAN Repository, then using the Update Center to install the eWay on Enterprise Designer. The GUI plug-in includes NetBeans code that runs in the Project Explorer, the Java Collaboration Wizard, the Connectivity Map editor, and the Deployment and Environment editors. The GUI code generated by the eDK does not need to be modified by the eDK user.

Figure 14 on page 45 illustrates how **NBM**, **WSDL**, and **Config Template** fit into the eDK component process flow.

Code Generation components called during deployment

All ICAN products provide a code generation component referred to as Codelets. Codelets are Java classes which are executed during activation (when a user clicks on Activate in the Deployment editor). Codelets are responsible for generating artifacts that get packaged during project deployment. In ICAN, project deployments come in the form of **.ear** files. eWay codelets primarily package its J2EE resource adapter (**.rar)** file, a Message-driven Bean (MDB) implementation (if inbound) and the associated deployment descriptors it generated based on project properties. A runtime-handler EJB is included (if outbound).

Figure 14 on page 45 illustrates how **Codelets** fit into the eDK component process flow.

3.6 eWay Implementation Details

This section describes how to modify and implement the required eWay class files. A complete listing of the methods found within each of the eDK class files is found in the Javadoc, see **"Using eDK-Based eWay Java Methods" on page 86**.

Note: *If the eWay has only outbound operations, then only the "src" folder will be present in the "connectors" folder, otherwise, "src_jca15" will be generated too.*

Class files and Interfaces to consider during implementation include:

- **<external_application_name>EwayConnection**

  This class has all the necessary methods to establish, match, and close connections to an external EIS system.

- **<external_application_name>ApplicationConnection**

  This class provides the client interface to the eWay resource adapter. **ApplicationConnnection** is used to obtain an Application object – <external_application_name>ClientApplication. For more information, refer to the **createApplication()** method in the Javadoc.

- **<external_application_name>ClientApplication**

  This interface is the "OTD" which contains all the outbound methods to be exposed in SeeBeyond's Java Collaboration Editor. Refer to the **<external_application_name>ClientApplicationImpl** class in the "appimpl" subfolder if you need to implement this interface.

- **<external_application_name>WebClientApplication**

  This class is the Object Type Definition (OTD) which contains all the outbound BPEL operations to be exposed in SeeBeyond's BPEL Collaboration Editor. Be sure that each method defined in this class is well implemented. Also note that all BPEL operations are considered to be "stateless".

- **<external_application_name> EwayActivationSpec**

  This class provides the inbound communication functions. It includes the methods to be executed when an endpoint activation is triggered (an Message-Driven Bean subscriber is deployed). Refer to **endpointActivation()** method in the Javadoc for details.

3.7 eWay Components

Several components make up an eWay. When you generate a new eWay using the eDK, make sure the following are included in your **.sar** file.

- **<External_application_name>.wsdl** – if BPEL operations are defined by the user.
- **<eway_name>.nbm** – contains the Netbeans module for the eWay.
- **<eway_name>.rar** and **<eway_name>_jca10.rar** – are the **.rar** files for the resource adapters.
- **Descriptor.xml** – is the **.xml** file the eWay installer requires to install the eway.
- **Images.zip** – contains all the necessary image icons.
- **Jar files** – including user specified third-party **.jar** files and other third-party jar files required for the eWay installer, and other framework **.jar** files.

3.8 Suggested Conventions for Writing JNI Code

The Java Native Interface (JNI) is a native programming interface that allows Java code running inside a Java Virtual Machine (JVM) to invoke platform specific code that runs outside the JVM.

Using JNI code in ICAN 5.0.x requires several steps at the various stages of eWay development, usage, and runtime.

Steps required during the JNI development phase include:

1. Write the JNI code using the native code and compile it into an OS specific native format, such as **.dll** for Windows, or **.so** for Solaris.
2. Create a thin Java wrapper to invoke the JNI code, and then build and package it in a separate **.jar** file.
3. Package both the native library and the jar file created earlier, into a **.zip** file.
4. Modify the eway-build.xml to ensure that the **.zip** file is added to the **.sar** file.
5. Modify the **install.xml**, located under:

   ```
   eways\[adapter name]\install
   ```

   Add the following code:

```
<taskdef name="UserDownloadable"
classname="com.stc.installer.UserDownloadableInstallTask" />

<UserDownloadable downloadableName="<DownloadableModuleName>"
file="${basedir}/<TheZipFile>.zip" repDir="InstallManager/50Base/
<eWayExternalName>/" repURL="${stc.rep.url}"
distURLBase="${stc.module.distURLBase}" />
```

This will add the **.zip** file as a downloadable object in Enterprise Manager.

6 Download the **.zip** file to a well-known location. The Integration Server must be configured to be aware of the JNI code.

7 In the Property Sheet for the integration server, update the "Append Classpath" property to point to the **.jar** file of the JNI-wrapper classes.

8 Update the path to the JNI code. The path is operating system dependent and is easiest in done in the shell right before invoking the **bootstrap.bat** file that starts your integration server.

9 For Windows, update your **PATH** variable to include the path to the JNI code. For Solaris, update your **LD_LIBRARY_PATH** variable to include the path to the JNI code. Refer to your specific OS documentation for setting up other libraries.

3.9 Extending Third-Party Resource Adapters

It is possible to use J2EE Connectors from third-party vendors for the connector component of an eWay.

Such modifications are primarily designed to:

1 Allow ICAN Collaborations to invoke the third-party connector.

The ICAN Collaboration framework invokes Connectors using the SeeBeyond AppConnn client interface. In order to use a third-party connector, it must be extended to provide this AppConn interface.

2 Specify connector configuration properties

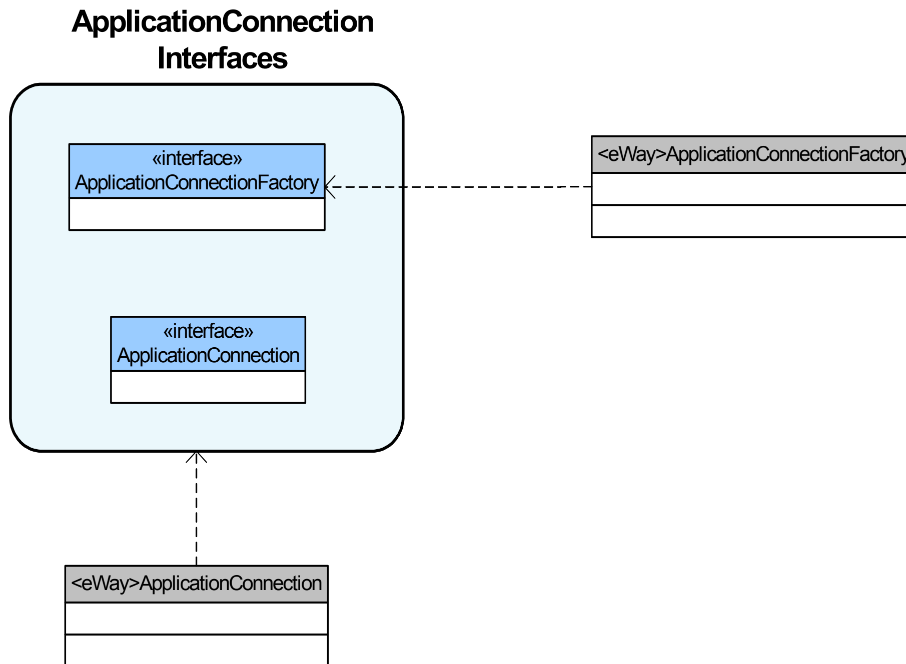
Connector configuration properties such as Resource Adapter class properties, Managed Connection Factory Bean properties, and Activation Spec properties, are specified in the connector's deployment descriptors. You must add these properties to the eWay's Configuration Template if the intention is to expose them in the Enterprise Designer (through the Connectivity Map link or External System configuration in the Environment).

For more information on adding components to the Configuration Template using the eWay Development Kit GUI, see **"Step 6: Define the eWay Configuration Template" on page 32**.

3.9.1 Providing the AppConn Client Interface

The AppConn interface is the client interface used by the ICAN Collaboration EJBs when communicating with eWay connectors.

Figure 15 AppConn Client Interface Class Diagram



To provide an AppConn interface to a third-party connector, the following must be created:

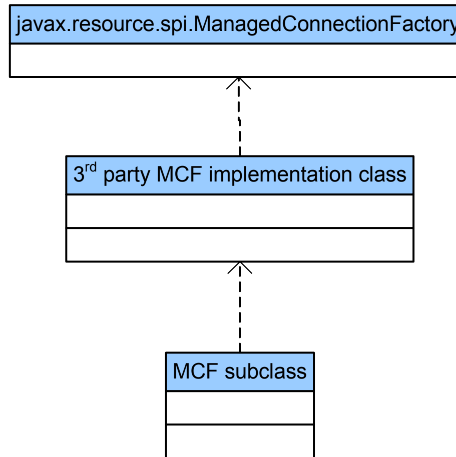
- An **ApplicationConnectionFactory** class, which implements the interface com.stc.connector.appconn.common.ApplicationConnectionFactory (found in com.stc.appconnapi.jar). This represents the client connection factory exposed by the eWay to EJB clients obtained via JNDI lookup. This connection factory must return an **ApplicationConnection** class through its **getConnection()** method.
- An **ApplicationConnection** class which implements com.stc.connector.appconn.common.ApplicationConnection (found in com.stc.appconnapi.jar). This represents the client connection obtained by EJB clients obtained by calling the **getConnection()** method on the **Connection** factory class.
- An **Application** class. The ApplicationConnection implementation described above must return an **Application** class in its **createApplication()** method. This **Application** class represents the Object Type Definition containing the methods exposed by the eWay. For example, the File eWay implements a **FileApplication** object which exposes the **setText()**, **getText()**, **write()**, and **writeBytes()** methods.
- A subclass of the third-party connector's **ManagedConnectionFactory** (MCF) Bean, see Figure 16 below. The third-party connector's MCF class cannot be used “as is”. Any MCF configuration properties must first be put in the eWay's configuration template in the appropriate outbound-configuration subsection, see **“Step 6: Define the eWay Configuration Template” on page 32**.

  The MCF subclass must provide the following methods.

  - a **setConfigurationTemplate()** method
  - a **setConfigurationInstance()** method
  - a **getConfigurationTemplate()** method
  - a **getConfigurationInstance()** method
  - a **createConnectionFactory()** method, which returns the eWay's application connection factory implementation, (see above).
  - a **createManagedConnection()** method

  Sample implementations of these methods are shown in **“Sample MCF Subclass Implementation” on page 51**.

Figure 16 Order of the ManagedConnectionFactory Subclass



3.9.2 Sample MCF Subclass Implementation

The sample code below extends the Managed Connection Factory implementation class called **FileManagedConnectionFactoryImpl**.

```
/*******************************************************************
*
*          Copyright (c) 2004, SeeBeyond Technology Corporation,
*          All Rights Reserved
*
*          This program, and all the routines referenced herein,
*          are the proprietary properties and trade secrets of
*          SEEBEYOND TECHNOLOGY CORPORATION.
*
*          Except as provided for by license agreement, this
*          program shall not be duplicated, used, or disclosed
*          without written consent signed by an officer of
*          SEEBEYOND TECHNOLOGY CORPORATION.
* *******************************************************************/
package com.stc.connector.fileadapter;

import javax.resource.ResourceException;
import javax.resource.cci.ConnectionFactory;
import com.stc.configuration.IConfiguration;
import com.stc.configuration.factory.Factory;
import java.io.ByteArrayInputStream;
import com.stc.connector.framework.util.Base64;

import org.apache.log4j.Logger;
import javax.resource.spi.ResourceAdapterInternalException;

/**
 * Extends MCF implementation class for a CCI based client connection factory
 *
 */
public class FileManagedConnectionFactoryImplExt
    extends FileManagedConnectionFactoryImpl implements STCManagedMaster {

    private Logger mLogger =
        Logger.getLogger("STC.eWay.file." + getClass().getName());

    /**
     * Creates a new instance of FileManagedConnectionFactoryImplExt
     */
    public FileManagedConnectionFactoryImplExt() {
    }

    /**
     * Provides the client connection factory
     *
     * @return AppConn based connection factory
     *
     * @exception javax.resource.ResourceException An exception from the
     *            Resource Adapter
     *
     */
    public Object createConnectionFactory()
```

```
        throws javax.resource.ResourceException {

        return new FileApplicationConnectionFactory(
                (ConnectionFactory) super.createConnectionFactory());
    }

    public javax.resource.spi.ManagedConnection createManagedConnection(
        javax.security.auth.Subject subject,
        javax.resource.spi.ConnectionRequestInfo connectionRequestInfo)
        throws javax.resource.ResourceException {

        return super.createManagedConnection(subject, connectionRequestInfo);
    }

   /** Configuration information are obtained from deployment descriptor as
* Base64 encoded strings.  A property for the base64 encoded configuration
* template XML and a property for the base64 encoded configuration instance
* XML are used.  The setter methods for the configuration template and instance
* are used to create the configuration model using the Configuration API.  Once
* the model is loaded, configurations are obtained from the model.
*/
    private java.lang.String configTemplate = null;
    private java.lang.String configInstance = null;
    private ByteArrayInputStream configTemplateBIS = null;
    private ByteArrayInputStream configInstanceBIS = null;
    private IConfiguration mcfModelConfig = null;

    private void loadTheConfigModel() throws Exception {
        if (this.configTemplateBIS != null
            && this.configInstanceBIS != null) {
            mcfModelConfig = (new Factory()).getConfiguration(
                    configTemplateBIS,
                    configInstanceBIS);
        }
    }

    /**
     * Gets the value of the ConfigurationTemplate property.
     *
     * @return java.lang.String containing the value of the ConfigurationTemplate property.
     */
    public java.lang.String getConfigurationTemplate() {
        return this.configTemplate;
    }

    /**
     * Sets the value of the ConfigurationTemplate property.
     *
     * @param configTemplate java.lang.String containing the value to be assigned to
     *        ConfigurationTemplate.
     *
     * @throws java.beans.PropertyVetoException error firing vetoable property
     *         change
     */
    public void setConfigurationTemplate(java.lang.String configTemplate)
        throws java.beans.PropertyVetoException {
        String configTemplateDecoded = null;
        try {
            configTemplateDecoded = Base64.decode (configTemplate, "UTF-8");
        } catch (Throwable th) {
            mLogger.error (th.toString(), th);
        }

        this.configTemplate = configTemplate;

        try {
            this.configTemplateBIS =
                new ByteArrayInputStream (configTemplateDecoded.getBytes("UTF-8"));
        } catch (Exception ex) {
            mLogger.error (ex.toString(), ex);
            throw new java.beans.PropertyVetoException(
                    ex.toString(),
                    new java.beans.PropertyChangeEvent(this,
                            "configTemplate",
                            this.configTemplate,
                            configTemplate));
        }

        if (this.configTemplateBIS == null) {
            throw new java.beans.PropertyVetoException(
                    "invalid config template",
                    new java.beans.PropertyChangeEvent(this,
                            "configTemplate",
                            this.configTemplate,
                            configTemplate));
        }

        try {
            loadTheConfigModel();

            // Only gets executed if configInstance and
            // configTemplate are already set
```

```
            //
            if (this.mcfModelConfig != null) {

                ConfigurationHelper configHelper =
                    new ConfigurationHelper(mcfModelConfig.getSection("parameter-settings"));

                super.setDirectory(
                    configHelper.getParameter("Directory").getValue().toString());

                super.setAddEOL(
                    configHelper.getParameter("AddEOL").getValue().toString());

                super.setMultipleRecordsPerFile(
                    configHelper.getParameter("MultipleRecordsPerFile").getValue().toString());

                super.setOutputFileName(
                    configHelper.getParameter("OutputFileName").getValue().toString());
            }

        } catch (Exception ex) {
            mLogger.error(ex.toString(), ex);
            throw new java.beans.PropertyVetoException(
                    "invalid config template",
                    new java.beans.PropertyChangeEvent(this,
                            "configTemplate",
                            this.configTemplate,
                            configTemplate));
        }
    }

    /**
     * Gets the value of the ConfigurationInstance property.
     *
     * @return java.lang.String containing the value of the ConfigurationInstance property.
     */
    public java.lang.String getConfigurationInstance() {
        return this.configInstance;
    }

    /**
     * Sets the value of the ConfigurationInstance property.
     *
     * @param configInstance java.lang.String containing the value to be assigned to
     *        ConfigurationInstance.
     *
     * @throws java.beans.PropertyVetoException error firing vetoable property
     *         change
     */
    public void setConfigurationInstance(java.lang.String configInstance)
        throws java.beans.PropertyVetoException,
               javax.resource.spi.ResourceAdapterInternalException,
               java.io.UnsupportedEncodingException {

        String configInstanceDecoded = Base64.decode (configInstance, "UTF-8");
        this.configInstance = configInstance;

        try {
            this.configInstanceBIS =
                new ByteArrayInputStream (configInstanceDecoded.getBytes("UTF-8"));
        } catch (Exception ex) {
            mLogger.error(ex.toString(), ex);
            throw new java.beans.PropertyVetoException(
                    "invalid config instance",
                    new java.beans.PropertyChangeEvent(this,
                            "configInstance",
                            this.configInstance,
                            configInstance));
        }

        if (this.configInstanceBIS == null) {
            throw new java.beans.PropertyVetoException(
                    "invalid config instance",
                    new java.beans.PropertyChangeEvent(this,
                            "configInstance",
                            this.configInstance,
                            configInstance));
        }

        try {
            loadTheConfigModel();

            // Only gets executed if configInstance and
            // configTemplate are already set
            //
            if (this.mcfModelConfig != null) {

                ConfigurationHelper configHelper =
                    new ConfigurationHelper(mcfModelConfig.getSection("parameter-settings"));

                super.setDirectory(
                    configHelper.getParameter("Directory").getValue().toString());
```

```
                super.setAddEOL(
                    configHelper.getParameter("AddEOL").getValue().toString());

                super.setMultipleRecordsPerFile(
                     configHelper.getParameter("MultipleRecordsPerFile").getValue().toString());

                super.setOutputFileName(
                    configHelper.getParameter("OutputFileName").getValue().toString());
            }

        } catch (Exception ex) {
            mLogger.error(ex.toString(), ex);
            throw new java.beans.PropertyVetoException(
                    "invalid config instance",
                    new java.beans.PropertyChangeEvent(this,
                            "configInstance",
                            this.configInstance,
                            configInstance));
        }
    }

    /**
     * Gets the configuration model for the instance of the
     * ManagedConnectionFactory.  The configuration model holds the properties
     * of the ManagedConnectionFactory.
     *
     * @return An instance of IConfiguration which can be used to get the
     *         properties of the ManagedConnectionFactory.
     *
     * @throws Exception upon error.
     */
    public IConfiguration getConfigurationModel()
        throws Exception {
        if (mcfModelConfig == null) {
            throw new Exception("Error loading config model");
        }
        return mcfModelConfig;
    }

}
```

Chapter 4

eDK eWay Concepts and Best Practices

This chapter describes some of the concepts used to successfully create eDK based eWays.

What's in this Chapter

- **Implementing Connection Logic to the External System** on page 55
- **Establishing Connections to the EIS** on page 57
- **Specifying Configuration Properties** on page 59
- **Wrapping Third-Party .jar Files** on page 62
- **Source Control** on page 62
- **Maintaining and Persisting State in Java Collaborations** on page 63
- **Generating Javadocs** on page 63

4.1 Implementing Connection Logic to the External System

The eDK generated code includes the shell for implementing establishing the physical connection to the external system. The shell code is based on the SeeBeyond Resource Adapter (RA) framework interfaces. The interface called **EwayConnection** provides the methods below. Some of these methods are also called from the connector's **Managed Connection Factory** class.

- **initialize()** – calls the connect method which establishes the physical connection to the external system, depending on the connection establishment mode
- **getConnection()** – returns the client connection to return to the Resource Adapter (RA) client. An implementation is provided in the generated code.
- **matchConnection()** – an implementation of connection matching is provided in the generated code.
- **cleanup()** – called every time the RA client calls **close()**.

The integration server calls the **cleanup()** method to force a clean-up on the ManagedConnection instance. The **ManagedConnection.cleanup()** method initiates a cleanup of the any client-specific state as maintained by a ManagedConnection instance. The clean-up invalidates all connection handles created using this ManagedConnection instance. Any attempt by an application component to use the connection handle after cleanup of the underlying ManagedConnection results in an exception.

The clean-up of ManagedConnection is always driven by an application server. An application server should not invoke ManagedConnection.cleanup when there is an uncompleted transaction (associated with a ManagedConnection instance) in progress.The invocation of **ManagedConnection.cleanup()** method on an already cleaned-up connection should not throw an exception.

The cleanup of ManagedConnection instance resets its client specific state and prepares the connection to be put back in to a connection pool. The cleanup method should not cause the resource adapter to close the physical pipe and reclaim system resources associated with the physical connection.

- **destroy()** – called by the integration server to destroy the physical connection.

  The destroy method destroys the physical connection to the underlying resource manager. To manage the size of the connection pool, an application server can explicitly call **ManagedConnection.destroy()** to destroy a physical connection. A resource adapter should destroy all allocated system resources for this ManagedConnection instance when the method destroy is called.

Additional information on how EJBs interact with Resource Adapters to obtain external connections, see **“RA Framework Sequence Diagram” on page 96**.

4.1.1 Implementing XA

The EwayConnection interface implementation shell code generated by the eDK includes a **getXAResource()** method. This corresponds to the JCA specified method used by the Integration server to obtain the eWay's **javax.transaction.xa.XAResource** implementation.

- **getXAResource()** – returns an **javax.transaction.xa.XAresource** instance. An application server enlists this XAResource instance with the Transaction Manager if the ManagedConnection instance is being used in a JTA transaction that is being coordinated by the Transaction Manager. If XA is not supported, this method should throw a javax.resource.ResourceException.

4.2 Establishing Connections to the EIS

An Enterprise Information System (EIS) is a standalone deployed application or system that provides a set of services to an enterprise for accessing, manipulating, and managing information.

The eDK supports establishing connections to the EIS both automatically and dynamically. The code that is generated by the eDK includes an Enterprise Java Bean (EJB) which calls the **getConnection()** method on the eWay resource adapter for obtaining a connection from the integration server managed connection pool. The **getConnection()** method implementation is delegated to the implementation of the **EwayConnection** interface.

4.2.1 Automatic Connection Establishment Mode

Automatic mode means a connection to the external system is established when the eWay is initialized. The **EwayConnection** interface's **initialize()** method implementation must be coded to call **getConnection()** to establish a physical connection to the EIS.

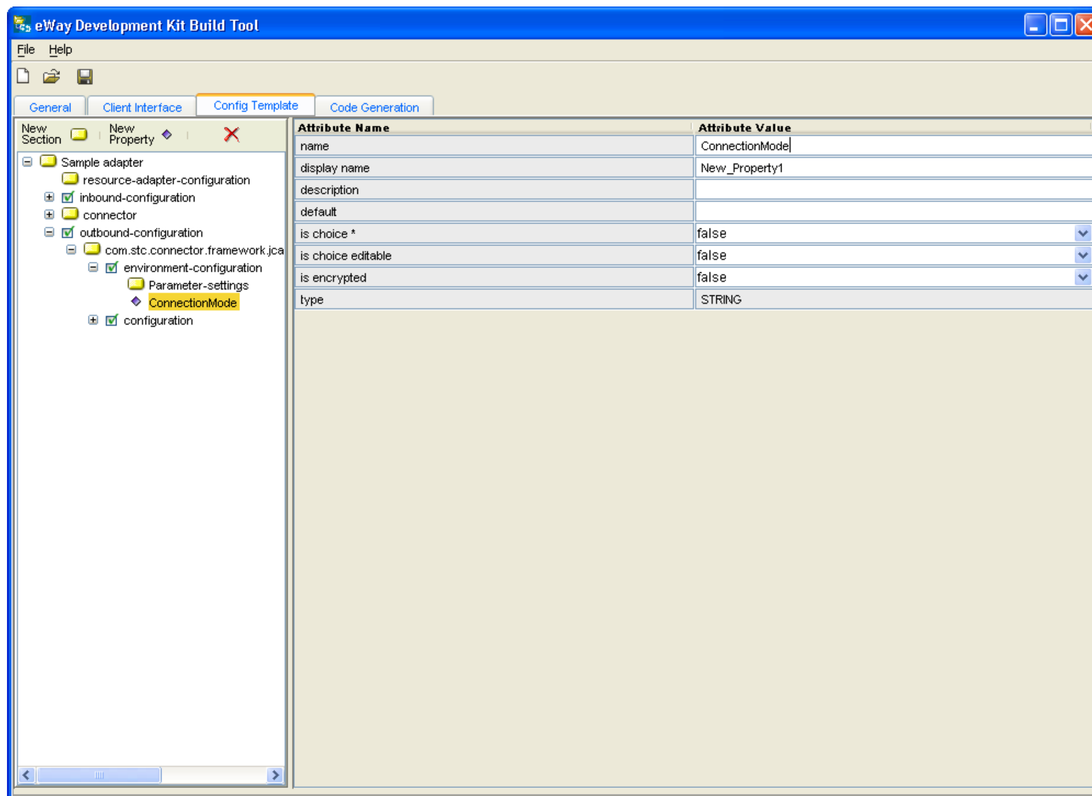
4.2.2 Dynamic Connection

In a dynamic connection (manual mode), the connection is not established when the **initialize()** method is called. Instead, the eWay end-user must explicitly call a method from Enterprise Designer.

Dynamic connections must be implemented as follows:

1. Add a property called "Connection-Mode" to the configuration template for specifying connection establishment mode. For example, add a Connection Mode section containing a choice property where the choices are "manual" and "automatic". You must add this under the outbound environment-configuration section, see Figure 17.

Figure 17 Configuration Template



2. Implement EwayConnection class to check for the value set for the property added. The initialize() method in the EwayConnection interface implementation must call connect() if automatic mode is set; it must not call connect() if manual mode is set.

Note: *The initialize(), connect(), disconnect(), and destroy() methods are already present in the Client Interface and do not need to be added using the build tool.*

3. Look in the **EwayConnection** class for the initialize() method (to connect) and the destroy() method (to disconnect). If the connection mode is to be set to "manual" mode, then the following try/catch block must be commented out, otherwise leaving it in leaves the connection mode in "automatic" mode

```
try {
            connect();
        } catch (Exception e) {
            e.printStackTrace();
            throw new ResourceException("Failed to connect: [" +
e.getMessage() + "]");
        }

    }
```

Setting the Connection mode to "manual" allows the eWay connection to be manipulated in a Collaboration since the OTD permits manual connections.

Overriding Configurations at Design-time

It is possible to override default configuraiton values at design-time. This is achieved by calling the appropriate setter methods in the configuration bean class. You can get an instance of the config bean class via the appropriate getter method in the OTD.

```
<eWay>ClientApplicationImpl.java

    /**
     * Returns the testConfiguration object.
     *
     * @return the testConfiguration instance.
     */
    public testConfiguration getEwayConfiguration() {
        return appConn.getEwayConfiguration();
    }
}
```

4.3 Specifying Configuration Properties

eDK based eWay configuration properties are created on the Config Template tab of the eWay Development Kit Build Tool. The type of eWay being developed determines the type of inbound or outbound configuration properties added to the eWay.

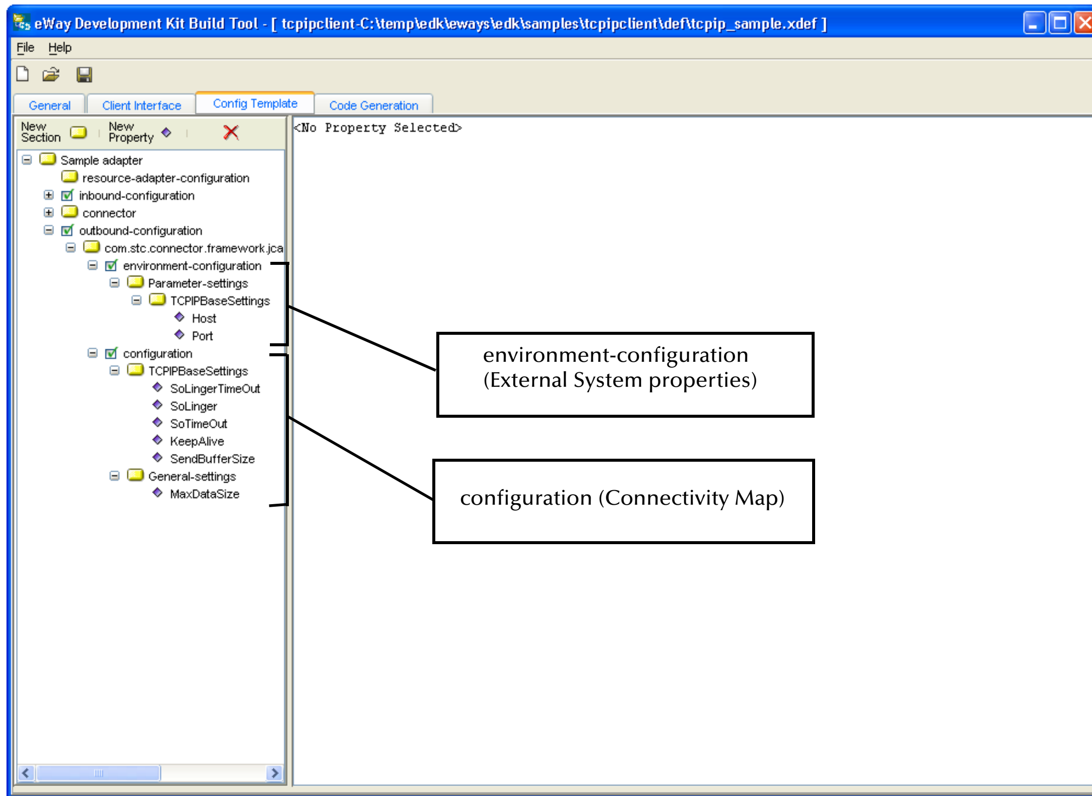
eWay configuration properties created on the Config Template tab are normally exposed to the ICAN user in the Enterprise Designer via:

- **Connectivity Map link**
- **External System Properties**

The Config Template is a superset template that contains a number of designated sections under which configurations for the Connectivity Map link and configurations for the External System properties are specified.

Figure 18 illustrates how configuration settings on an eWay (TCPIPClientAdapter in this example), contain the base settings for both the configuration (Connectivity Map) and environment-configuration (External System properties).

Figure 18 Config Template

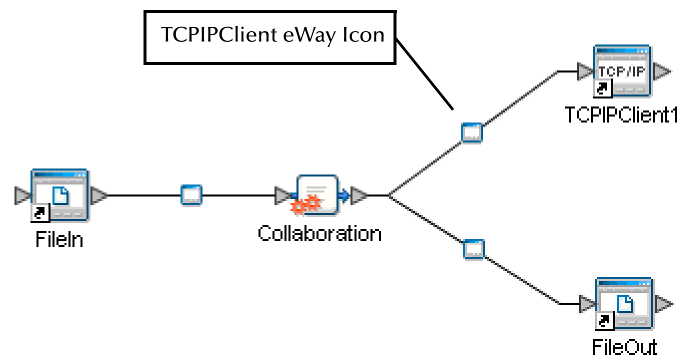


4.3.1 Connectivity Map eWay Properties

To access the Connectivity Map properties:

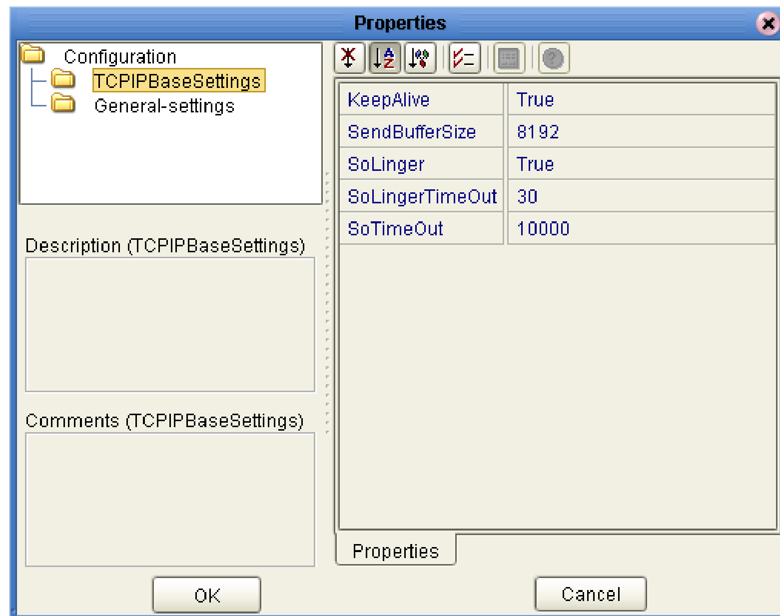
1. Open the SeeBeyond Enterprise Designer – Connectivity Map Editor for the eDK based eWay that you created.
2. Double-click the eDK based eWay icon to access the Properties window.

Figure 19 Connectivity Map



3 The eWay Properties window appears with the sections and parameters defined under the configuration section of the eWay Development Kit Build Tool – Config Template. Note that the configuration properties should only provide properties that are independent of the external system's physical location.

Figure 20 Connectivity Map Properties – TCPIPClientAdapter



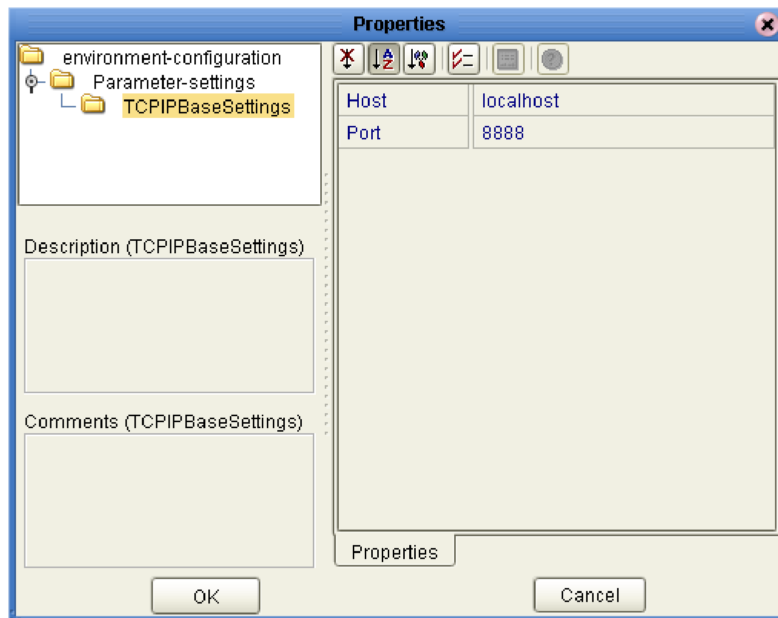
4.3.2 External System Properties

To access the external system properties:

1 Open the SeeBeyond Enterprise Designer – Environment Explorer for the eDK based eWay that you created.

2 Right-click the eDK based eWay (TCPIPClientAdapter in this example), and select **Properties**. The Properties window opens to the environment-configuration properties. Note that physical connection properties are normally provided here.

Figure 21 Environment Explorer Properties – TCPIPClientAdapter



4.4 Wrapping Third-Party .jar Files

The JCE client interface allows specification of attributes and methods in third-party classes. While it is possible to refer directly to third-party class files, it is recommended to create wrapper classes that the Java Collaboration Editor client interface can directly reference. Encapsulating third-party class files in wrapper classes eliminates any direct interaction, ensuring future flexibility when newer versions of the third-party class files introduce incompatible method signature changes.

4.5 Source Control

The eDK and eDK Build Tool support any source control system (e.g. CVS), although they have no direct tie into them. The eDK Build Tool does provide the ability to label different versions of the **.xdef** file, but it does not keep prior versions of the file. It is the responsibility of the eDK user and a third-party source control system to ensure adequate source control.

To work with a source control system, it is recommended to use a separate directory as the target for the stub code from the generator (i.e. the output directory on the generation tab of the eDK Build Tool), and another to implement the methods and operations defined for the OTD. These can also be thought of as the “pre” and “post” implementation directories.

Check eWay code into the source control system after generating for the first time. You can then check it out to directories where the stub methods and operations are to be implemented. After this, you can follow the usual source control operations and procedures from the implementation directories.

If modifications to the OTD are required after implementing the code, then the procedure is to first delete all of the files in the current output directory and then regenerate the stubs. Next, merge the new stub code with the existing files and code in the implementation directories. Check your merged code back into the source control system. You can now use the implementation directories to build the eWay and **.sar** file.

4.6 Maintaining and Persisting State in Java Collaborations

Collaboration rules set in the Enterprise Designer are executed in an Enterprise Java Beans (EJB) instance per incoming message. This means that data does not persist between EJB instantiations servicing different messages. Collaborations are normally triggered by messages received by an inbound eWay. The eWay's inbound connector sends the message to the Collaboration through EJBs. In a deployment, every message received by the inbound connector triggers instantiation of EJBs or retrieval from an EJB pool.

There are a number of options to keep track of information such as state across multiple messages:

- Save state to JMS queue or topic. This requires designing Collaborations to include publishing and reading data from the Java Messaging Service (JMS).
- Call methods in the inbound eWay's client interface which persist and retrieve the state. An eWay may expose a method in the client interface for the purpose of persisting information. For example, a Java Collaboration Editor (JCE) client interface may provide a **setState()** and **getState()** method which can be called from a Collaboration. The methods implemented in the eWay's connector can save the information to a file system or database.

4.7 Generating Javadocs

You can generate Javadocs for both the connectors folder and the eWays folder after completing eWay implementation.

To generate Javadocs for the connectors folder:

1. From the command window, browse to the eWay adapter under the connectors folder.

```
<STC_ROOT>\connectors\<Adapter Name>adapter
```

2. Run the following to generate Javadocs for the connectors folder:

```
ant -f connector-build.xml build-javadoc
```

3 Browse to the following to access the generated Javadoc:

```
<STC_ROOT>\BUILD\Modules\connectors\<Adapter Name>adapter\javadoc
```

To generate Javadocs for the eways folder:

1 From the command window, browse to the eWay adapter under the eways folder:

2 Run the following to generate Javadocs for the eways folder:

```
ant -f eway-build.xml build-javadoc
```

3 Browse to the following to access the generated Javadoc:

```
<STC_ROOT>\BUILD\Modules\eways\<Adapter Name>adapter\javadoc
```

Chapter 5

Sample eDK eWay Projects

This chapter describes the samples included in the eWay Development Kit, and the ICAN sample projects created from them.

What's in this Chapter

- **Importing eDK Samples** on page 65
- **Creating the edkfile Sample in the Build Tool** on page 66

5.1 Importing eDK Samples

Three eDK samples are included in the **eWayDevelopmentKit.sar** file. These samples are designed to provide a general understanding of how to build an eWay using the eWay Development Kit Build Tool.

eDK eWay samples include:

- File eWay inbound/outbound example, located under:

  ```
  ${env.STC_ROOT}\eways\edk\samples\edkfile
  ```

- TCP/IP outbound client example, located under:

  ```
  ${env.STC_ROOT}\eways\edk\samples\tcpipclient
  ```

- TCP/IP inbound server example, located under:

  ```
  ${env.STC_ROOT}\eways\edk\samples\tcpipserver
  ```

5.1.1 Importing a Sample into the eWay Development Kit Build Tool

To Import a sample eWay into the eWay Development Kit Build Tool:

1. Start the eWay Development Kit build tool.
2. Click the Open eWay icon, or choose **File** > **Open** from the file menu.
3. Browse to and select one of the eDK eWay samples under:

   ```
   ${env.STC_ROOT}\eways\edk\samples\<eDK sample>\def\<eDK
   sample>_sample.xdef
   ```

4. Click **Open**. The sample appears in the eWay Development Kit Build Tool.

5.2 Creating the edkfile Sample in the Build Tool

This section describes the settings and parameters used in the **edkfile** sample. For detailed instructions on entering information into the eWay Development Kit Build Tool are found in **"Using the eWay Development Kit" on page 15**.

5.2.1 Overview

The **edkfile** eWay sample provides a simple inbound/outbound scenario that is similar to the *File eWay Intelligent Adapter*, which is used to exchange data between an external file system and the eGate Integrator System.

As an inbound eWay, the **edkfile** eWay polls an input directory for files based on a file name matching a specified regular expression. When the eWay detects a matching file, it opens the file and publishes the data to a Collaboration or Business Process Service. The original data file is then renamed to the same file name with a **.~in** extension. For example, **input1.txt** is renamed to **input1.~in**.

As an outbound eWay, the **edkfile** writes processed data to a file in an output directory. The default file name is **output%d.dat**; for example, **output1.dat**, **output2.dat**, and so on.

The following steps outline the procedures required to create the **edkfile** eWay using the eDK.

Steps to build the edkfile eWay include:

1. Obtain a license for the new eWay.
2. Set the environment variables.
3. Start the eWay Development Kit Build Tool.
4. Create and specify details of the new eWay – such as the eWay Name, Description, Version, and so forth.
5. Enter the required eWay Interfaces – including any return types, parameter names, parameter types, exceptions thrown, and so forth.
6. Define the eWay configuration template.
7. Run the code generator to create the eWay shell code.
8. Modify the generated shell code in the eWay implementation environment and provide the required implementation.
9. Run the Apache Ant build tool to build the eWay **.sar** file.
10. Upload the new eWay **.sar** file to the ICAN Repository using Enterprise Manager.
11. Run the Enterprise Designer Update Center.
12. Create, build, and deploy a new Project using the new eDK based eWay.

5.2.2 Step 1: Acquire an eDK eWay License File

A valid license file is required to upload your new eWay into the ICAN Repository using Enterprise Manger. Refer to **"Step 1: Acquire an eDK eWay License File" on page 18** for information on acquiring an eDK eWay license file.

5.2.3 Step 2: Set the Environment Variables

An implementation environment is required to compile Java source files and to build an eWay **.sar** file.

To set the implementation environment variables:

1. Open a new command line window.
2. Locate the root directory of the extracted eDK folder (e.g. C:\eDK), and run the **env.bat** file.

5.2.4 Step 3: Start the eWay Development Kit Build Tool

To start the eWay Development Kit build tool:

1. From the same command window change directories to:

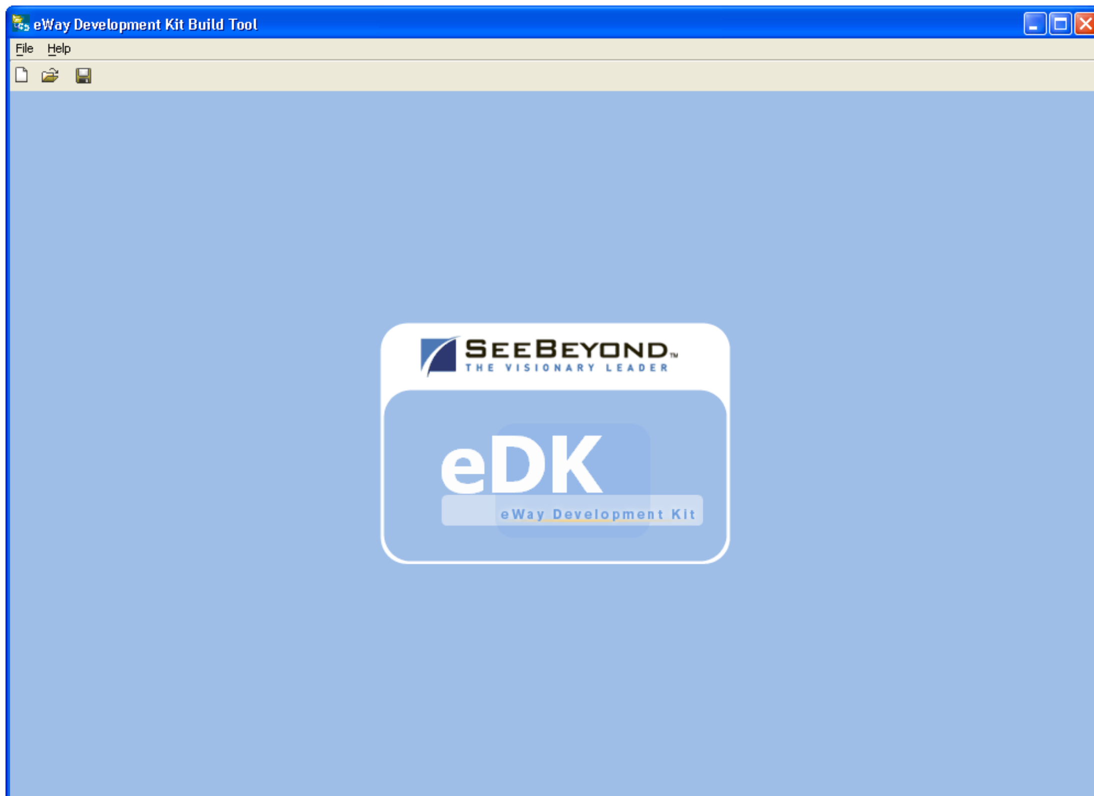
   ```
   <eDK root>\eways\edk\devtools
   ```

2. Run the following command:

   ```
   ant runedkgui
   ```

   The eWay Development Kit splash screen appears.

Figure 22 eWay Development Kit Build Tool Splash Screen



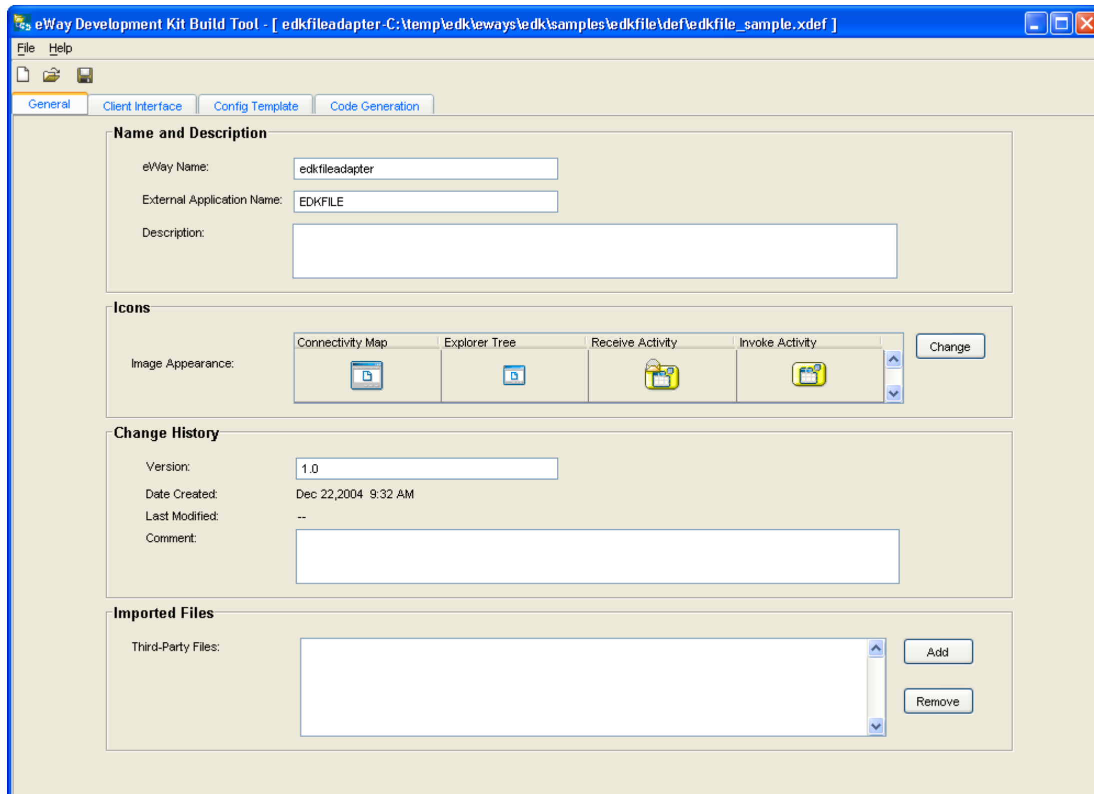
5.2.5 Step 4: Create and Specify the New eWay

Perform the following steps to create the new eWay:

1. From the file menu, select **File** then select **New eWay**, or click the **New eWay** icon.
2. Enter the following on the General tab.
   - eWay Name **=** “**edkfileadapter**”
   - External Application Name = “**EDKFILE**”
   - Version = “**1.0**”

   Figure 23 below displays the completed fields on the General tab.

Figure 23 General Tab of the eWay Development Kit Build Tool



5.2.6 Step 5: Enter the Required eWay Interfaces

Enter the following on the Client Interface tab.

1. Create a new Java outbound interface method and enter the following:
   - Method Name = “**write**”
   - Return Type = “**int**”
   - Throw Exception = “**EDKFileApplicationException**”
2. Create a new Java outbound interface method and enter the following:
   - Method Name = “**writeBytes**”
   - Return Type = “**int**”
   - Throw Exception = “**EDKFileApplicationException**”
3. Create a new Java attribute and enter the following:
   - Attribute Name = “**text**”
   - Type = “**java.lang.String**”
4. Create a new Java attribute and enter the following:
   - Attribute Name = “**byteArray**”
   - Type = “**byte[]**”

5 Create a new Java user defined class and enter the following:

- Name = "**EDKFileApplicationException**"
- Attribute List = "**errMsg**" (Name), "**java.lang.String**" (Type)

6 Create a new inbound BPEL interface method and enter the following:

- Method Name = "**read**"
- Parameter List = "**EDKFileTextMessage**"

7 Create a new BPEL user defined class and enter the following:

- Name = "**EDKFileWriteOutputMessage**"
- Attribute List = "**status**" (Name), "**boolean**" (Type)

8 Create a new BPEL user defined class and enter the following:

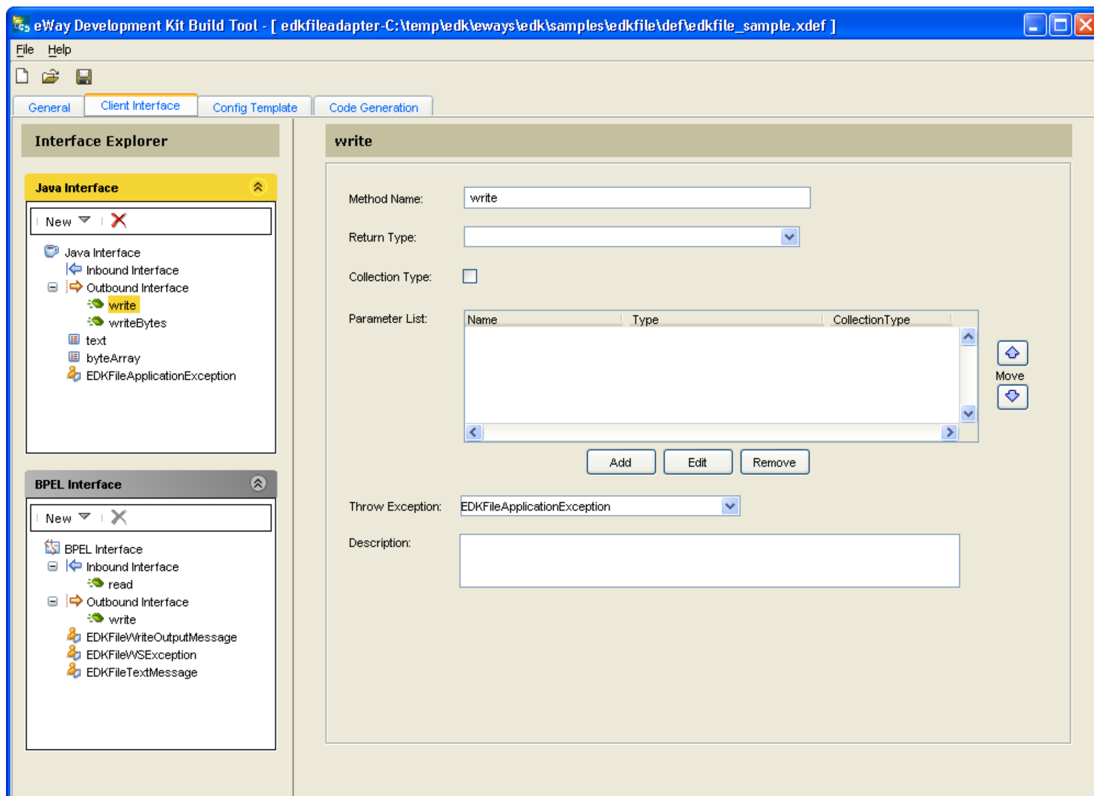
- Name = "**EDKFileWriteOutputMessage**"
- Attribute List = "**message**" (Name), "**java.lang.String**" (Type)

9 Create a new BPEL user defined class and enter the following:

- Name = "**EDKFileTextMessage**"
- Attribute List = "**edkfiletext**" (Name), "**java.lang.String**" (Type)

Figure 24 below displays the completed fields on the Client Interface tab.

Figure 24 Client Interface tab of the eWay Development Kit Build Tool



5.2.7 Step 6: Define the eWay Configuration Template

Enter the following properties and attribute values on the Config Template tab.

The eWay Config Template defines the following properties:

- inbound-configuration properties
- outbound-configuration properties

inbound-configuration Properties

inbound-configuration properties are listed under the following sections:

```
com.stc.connector.framework.jca.system.STCActivationSpec >
configuration > parameter-settings
```

Add the following:

1. Create a property called **InputType**, and define the following attribute values:
   - name = “InputType”
   - display name = “InputType”
   - description = “InputType”
   - default = “Bytes”
   - is choice* = “false”
   - is choice editable = “false”
   - is collection = “true”
   - type = “STRING”
2. Create a property called **RemoveEOL**, and define the following attribute values:
   - name = “RemoveEOL”
   - display name = “RemoveEOL”
   - description = “If multiple records per file is True, this is an option to exclude the terminating EOL character from the message sent to the subscriber.”
   - default = “false”
   - is choice* = “false”
   - is choice editable = “false”
   - is collection = “false”
   - type = “BOOLEAN”

3. Create a property called **MultipleRecordsPerFile**, and define the following attribute values:

   - name = “MultipleRecordsPerFile”
   - display name = “Multiple records per file”
   - description = “Specifies if multiple records will be obtained per file. Multiple records will be generated per line up to the maximum bytes per record.”
   - default = “false”
   - is choice* = “false”
   - is choice editable = “false”
   - is collection = “false”
   - type = “BOOLEAN”

4. Create a property called **MaxBytesPerRecord**, and define the following attribute values:

   - name = “MaxBytesPerRecord”
   - display name = “Maximum bytes per record”
   - description = “Maximum number of bytes per record sent to the subscriber.”
   - default = “4096”
   - is choice* = “false”
   - is choice editable = “false”
   - is collection = “false”
   - default min = “”
   - default max = “”
   - type = “NUMBER”

5 Create a property called **InputFileMask**, and define the following attribute values:
   - name = “InputFileMask”
   - display name = “Input file name”
   - description = “Input file name”
   - default = “input*.txt”
   - is choice* = “false”
   - is choice editable = “false”
   - is collection = “false”
   - is encrypted = “false”
   - type = “STRING”
6 Create a property called **Directory**, and define the following attribute values:
   - name = “Directory”
   - display name = “Directory”
   - description = “Directory”
   - default = “C:\temp”
   - is choice* = “false”
   - is choice editable = “false”
   - is collection = “false”
   - type = “Path”

outbound-configuration Properties

outbound-configuration properties are listed under the following sections:

```
com.stc.connector.framework.jca.system.STCManagedConnectionFactory >
configuration > parameter-settings
```

Add the following:

1 Create a property called **AddEOL**, and define the following attribute values:
   - name = “AddEOL”
   - display name = “Add EOL”
   - description = “Add EOL”
   - default = “true”
   - is choice* = “false”
   - is choice editable = “false”
   - is collection = “false”
   - type = “BOOLEAN”

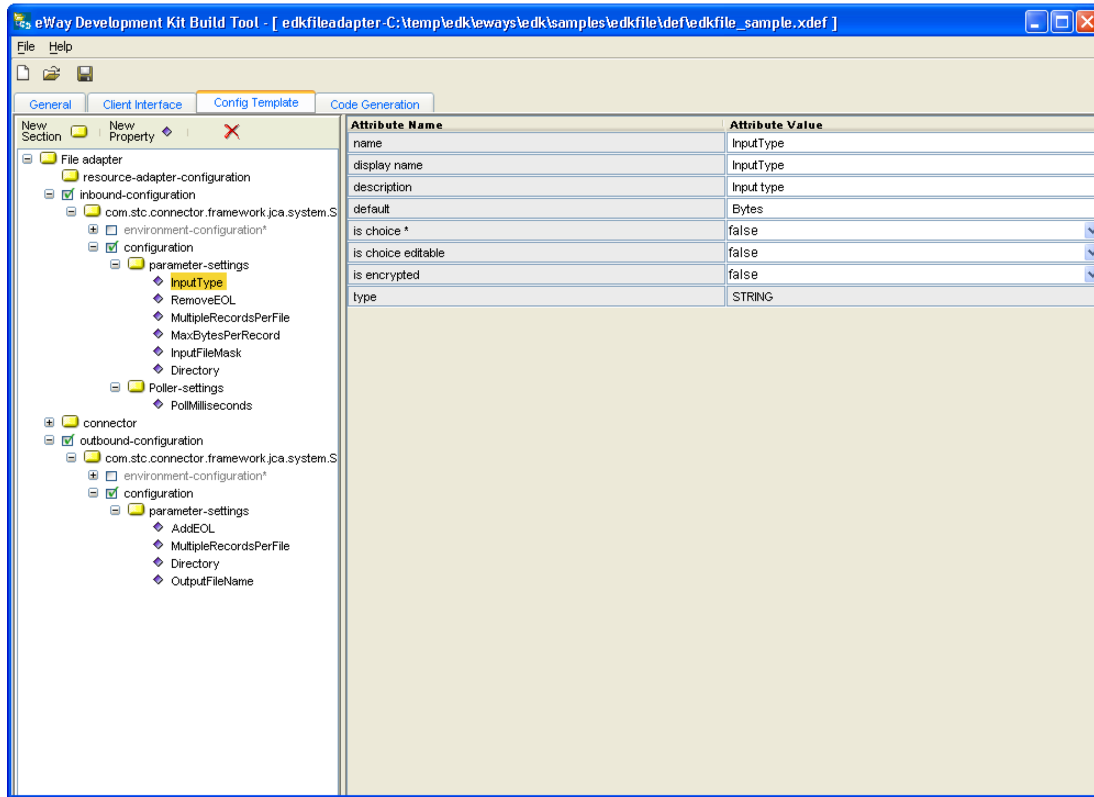
2 Create a property called **MultipleRecordsPerFile**, and define the following attribute values:

  - name = "MultipleRecordsPerFile"
  - display name = "Multiple records per file"
  - description = "Multiple records per file"
  - default = "true"
  - is choice* = "false"
  - is choice editable = "false"
  - is collection = "false"
  - type = "BOOLEAN"

- Create a property called **Directory**, and define the following attribute values:
  - name = "Directory"
  - display name = "Directory"
  - is readable = "true"
  - description = "Directory"
  - default = "C:\temp"
  - is choice* = "false"
  - is choice editable = "false"
  - is collection = "false"
  - type = "PATH"
- Create a property called **OutputFileName**, and define the following attribute values:
  - name = "OutputFileName"
  - display name = "Output file name"
  - description = "Output file name"
  - default = "output%d.dat"
  - is choice* = "false"
  - is choice editable = "false"
  - is collection = "false"
  - is encrypted = "false"
  - encryption key = "null"
  - type = "STRING"

Figure 25 below displays the completed properties on the Config Template tab.

Figure 25 Config Template tab of the eWay Development Kit Build Tool



5.2.8 Step 7: Run the Code Generator

Enter the following on the Code Generation tab.

1. Create the following directory in the Output directory field:

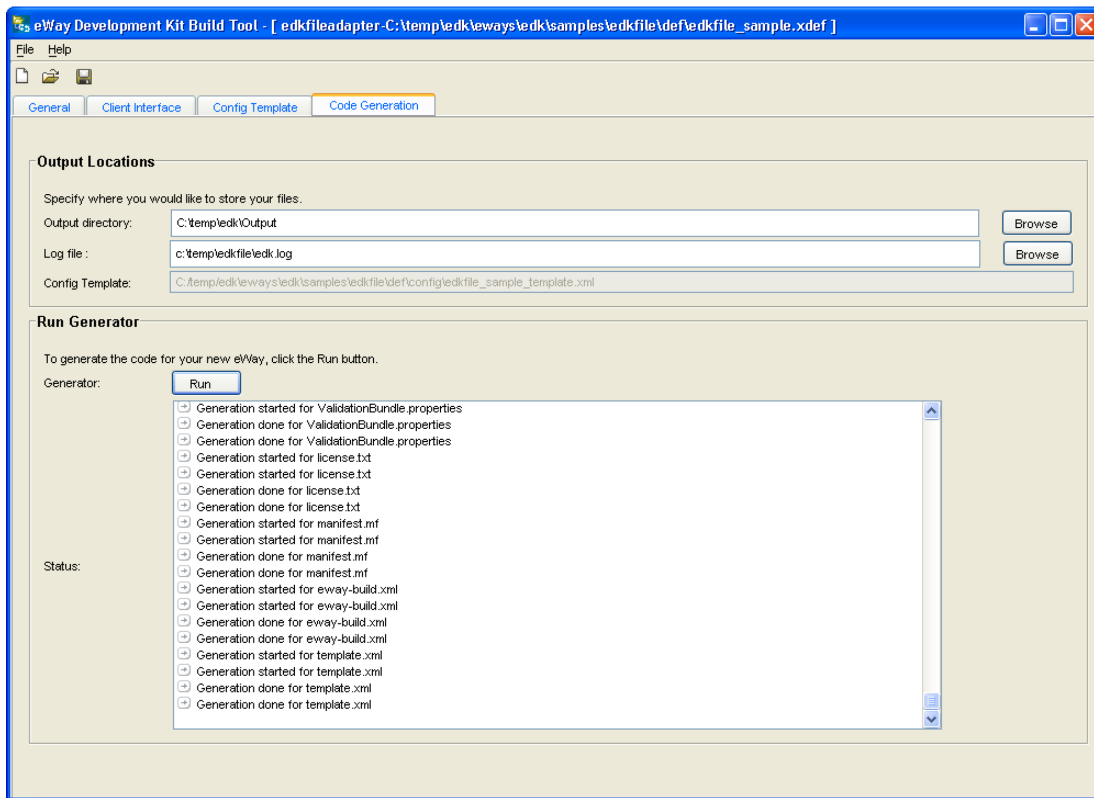
   `c:\temp\edkfile\pre`

2. Create the following directory in the Log file field:

   `c:\temp\edkfile\edk.log`

3. Click **Run** to run the code generator. The Status window displays a list of completed tasks, see Figure 26 below.

Figure 26 Code Generation tab of the eWay Development Kit Build Tool



The eWay Development Kit Build Tool generates two main folders in the specified output directory.

- **connectors folder** – contains all the J2EE connector code. The **.java** files contained in this folder require implementation.
- **eways folder** – contains the GUI code that plugs into the Enterprise Designer's code generation components. No implementation is required in this folder.

5.2.9 Step 8: Implement the eWay

Complete the following steps to modify and implement the generated shell code.

1. Browse to and copy **edkfileadapter**, located under the **connectors** folder in the output directory, to:

   ```
   ${env.STC_ROOT}\connectors\
   ```

2. Modify the **<external_application>EWayConnection** class. The following is an example of how this class is implemented.

```
/***************************************************************************
 *
 *          Copyright (c) 2004, SeeBeyond Technology Corporation,
 *          All Rights Reserved
 *
 *          This program, and all the routines referenced herein,
 *          are the proprietary properties and trade secrets of
 *          SEEBEYOND TECHNOLOGY CORPORATION.
 *
 *          Except as provided for by license agreement, this
 *          program shall not be duplicated, used, or disclosed
 *          without written consent signed by an officer of
 *          SEEBEYOND TECHNOLOGY CORPORATION.
 *
 ***************************************************************************/
package com.stc.connector.edkfileadapter.ewayconn;


import com.stc.connector.edkfileadapter.appconn.EDKFILEApplicationConnection;

import com.stc.connector.framework.client.AssociateableHandle;
import com.stc.connector.framework.eway.EwayConnection;
import com.stc.connector.framework.eway.ManagedConnectionCallback;
import com.stc.connector.framework.jca.system.STCPropertiesInfo;
import com.stc.connector.framework.jca.system.STCManagedConnectionMetaData;
import com.stc.connector.management.STCManagedSlave;
import com.stc.connector.management.jca.system.mbeans.STCManagedConnectionMonitorBean;

import javax.resource.ResourceException;
import javax.resource.NotSupportedException;

import javax.resource.spi.ConnectionRequestInfo;
import javax.resource.spi.LocalTransaction;
import javax.resource.spi.ManagedConnectionMetaData;

import javax.security.auth.Subject;

import javax.transaction.xa.XAResource;

import org.apache.log4j.Logger;

/**
 * This class implements the EwayConnection interface for EDKFILE.
 * It is required that it has a public constructor without arguments.
 *
 */
public class EDKFILEEwayConnection
    implements EwayConnection,  STCManagedSlave  {
    private ManagedConnectionCallback mcCallback;
    private Subject subject;
    private STCPropertiesInfo cri;
   private Logger mLogger = Logger.getLogger("STC.eWay.edkfile" + getClass().getName());
    private STCManagedConnectionMonitorBean mBean = null;

    /**
     * Creates a new instance of EDKFILEEwayConnection
     */
    public EDKFILEEwayConnection() {
        mLogger.debug("Instance of EDKFILEEwayConnection created...");
    }

    /**
     * Initialize the EwayConnection.  Establish the connection to the external
     * system (EIS).
     *
     * @param mcCallback The callback object for getting services from the
     *        associated ManagedConnection.
     * @param subject The Subject instance which holds the credentials for EIS
     *        signon.
     * @param cri The ConnectionRequestInfo instance which can hold both EIS
     *        signon information or general connection specific information.
     *        The ConnectionRequestInfo provided will contain the union of the
     *        properties from the client connection request properties, the
     *        ResourceAdapter properties, and the ManagedConnectionFactory
```

```
     *        properties.
     *
     * @throws ResourceException upon error.
     */
    public void initialize(ManagedConnectionCallback mcCallback,
        Subject subject, ConnectionRequestInfo cri)
        throws ResourceException {
        mLogger.debug ("Invoking initialize...");
        if ((mcCallback == null) ||
            (cri == null) || ((STCPropertiesInfo)cri).isEmpty()) {
           throw new ResourceException("Invalid parameters - ManagedConnectionCallback is [" +
mcCallback +
                                        "] ConnectionRequestInfo is [" + cri + "]");
        }
        this.mcCallback = mcCallback;
        this.subject = subject;
        this.cri = (STCPropertiesInfo) cri;

        // custom connection initialization code goes here (if any)...
        // <Start_User_Code>

        // <End_User_Code>


    }

    /**
     * Determines whether this instance of the EwayConnection matches the
     * the connection request with the connection information in Subject and/or
     * ConnectionRequestInfo.
     *
     * @param subject The Subject instance which holds the credentials for EIS
     *        signon.
     * @param cri The ConnectionRequestInfo instance which holds both EIS
     *        signon information or general connection specific information.
     *        The ConnectionRequestInfo provided will contain the union of the
     *        properties from the client connection request properties, the
     *        ResourceAdapter properties, and the ManagedConnectionFactory
     *        properties.
     *
     * @return  A boolean true if there is a connection match; false otherwise.
     */
    public boolean matchConnection(Subject subject, ConnectionRequestInfo cri) {
      boolean match = false;
      // determines the custom match criteria here...
      // <Start_User_Code>

      // <End_User_Code>

      return match;
    }

    /**
     * Get a new instance of the connection handle (application connection).
     *
     * @param subject The Subject instance which holds the credentials for EIS
     *        signon.
     * @param cri The ConnectionRequestInfo instance which holds both EIS EIS
     *        signon information or general connection specific information.
     *        The ConnectionRequestInfo provided will contain the union of the
     *        properties from the client connection request properties, the
     *        ResourceAdapter properties, and the ManagedConnectionFactory
     *        properties.
     *
     * @return A connection handle (application connection) instance which
     *         implements the Associateable interface.
     *
     * @throws ResourceException upon error.
     */
    public AssociateableHandle getConnection(Subject subject,
                                             ConnectionRequestInfo cri)
                                             throws ResourceException {
        // any custom authentication code goes here...
        // <Start_User_Code>

        // <End_User_Code>
        return new EDKFILEApplicationConnection(mcCallback, (STCPropertiesInfo)cri);
    }

    /**
     * Perform clean up of any resources or reset of any state held by the
     * instance of EwayConnection.
     *
     * @throws ResourceException upon error.
     */
    public void cleanup() throws ResourceException {
        // any custom cleanup code goes here...
        // <Start_User_Code>

        // <End_User_Code>
    }
```

```
/**
 * Release any resources prior to the destruction of the associated
 * ManagedConnection.
 *
 * @throws ResourceException upon error.
 */
public void destroy() throws ResourceException {
    // any custom resource release code goes here...
    // <Start_User_Code>

    // <End_User_Code>
}

/**
 * Get a LocalTransaction instance for local transaction demaracation.
 *
 * @return A LocalTransaction instance.
 *
 * @throws ResourceException upon error.
 */
public LocalTransaction getLocalTransaction() throws ResourceException {
  // any custom local transaction support code goes here,
    // <Start_User_Code>

    // <End_User_Code>

  // or un-comment the following throw clause if local transaction is not supported
    throw new ResourceException("Local transactions are not supported in EDKFILE eway");
}

/**
 * Get a new instance of the ManagedConnectionMetaData which contains
 * connection information for the currently established connection.
 * The ManagedConnectionMetaData interface provides information
 * about the underlying EIS instance associated with a
 * ManagedConnection instance.
 * An application server may use this information to
 * get runtime information about a connected EIS instance.
 * See <code>com.stc.connector.framework.jca.system.STCManagedConnectionMetaData</code>
 * for details.
 *
 * @return An instance of ManagedConnectionMetaData which contains
 *         information about the current established connection to the
 *         EIS.
 *
 * @throws ResourceException upon error.
 */
public ManagedConnectionMetaData getMetaData() throws ResourceException {
  // Currently, ICAN suite does NOT really use this metadata.
  // however, this could be used by other application servers
  // to gather information about the underlying EIS instance.
    // modify the following clause if necessary
    // <Start_User_Code>
    return new STCManagedConnectionMetaData(
        "EDKFILE", "UNKNOWN", 1, "UNKNOWN");
    // <End_User_Code>
}

/**
 * Get an XAResource instance for global transaction demaracation.
 *
 * @return A XAResource instance.
 *
 * @throws ResourceException upon error.
 */
public XAResource getXAResource() throws ResourceException {
  // any custom XA support code goes here,
    // <Start_User_Code>

    // <End_User_Code>

  // or un-comment the following throw clause if XA is not supported
    throw new ResourceException("XA transactions are not supported in EDKFILE eway");
}

/**
 * Implementing the STCManagedSlave interface to set up/register a callback so that
 * the connector framework can initialize the Mbean for this eway
 * @param Object the mbean to be set by the framework
 * @return none
 */
public void setMBean(Object mbean) {
  this.mBean = (STCManagedConnectionMonitorBean) mbean;
}

/**
 * Convenience method to get the initialized mBean
 * @param none
 * @return STCManagedConnectionMonitorBean the mbean
 */
public STCManagedConnectionMonitorBean getMBean() {
  return this.mBean;
```

```
    }
}
```

3 Implement the **<external_application>ClientApplicationImpl** class under the **appconn/appimpl** subfolder, and add new classes there if necessary. The following is an example of how this class is implemented.

```
/**************************************************************************
 *
 *          Copyright (c) 2004, SeeBeyond Technology Corporation,
 *          All Rights Reserved
 *
 *          This program, and all the routines referenced herein,
 *          are the proprietary properties and trade secrets of
 *          SEEBEYOND TECHNOLOGY CORPORATION.
 *
 *          Except as provided for by license agreement, this
 *          program shall not be duplicated, used, or disclosed
 *          without written consent signed by an officer of
 *          SEEBEYOND TECHNOLOGY CORPORATION.
 *
 ***************************************************************************/
package com.stc.connector.edkfileadapter.appconn.appimpl;

import com.stc.connector.edkfileadapter.appconn.EDKFILEApplicationConnection;
import com.stc.connector.edkfileadapter.appconn.EDKFILEClientApplication;
import com.stc.connector.edkfileadapter.appconn.EDKFILEConfiguration;

import org.apache.log4j.Logger;

//user added
import java.io.File;
import java.io.FileOutputStream;
import com.stc.connector.framework.util.SemiSema;
// end of user added

/**
 * Implements EDKFILEClientApplication which exposes
 * operations available for the client application.
 *
 */
public class EDKFILEClientApplicationImpl implements EDKFILEClientApplication {

    private EDKFILEApplicationConnection appConn = null;
    private EDKFILEConfiguration config = null;
    private Logger logger = Logger.getLogger("STC.eWay.edkfile"
                            + getClass().getName());

    //user added
    private static String FILE_SEPARATOR = System.getProperty(
            "file.separator");
    private String actualOutputFileName = null;
    private CounterManager countMgr = null;
    private SemiSema writeLock = null;
    private String directory = null;
    private String outputFileName = null;
    // end of user added

    /**
     * Constructor.
     *
     * @param appConn   The EDKFILEApplicationConnection instance.
     *
     */
    public EDKFILEClientApplicationImpl (EDKFILEApplicationConnection appConn) {
        this.appConn = appConn;
        this.config = appConn.getEwayConfiguration();

        // user added
        this.directory = config.getDirectory();
      this.outputFileName = config.getOutputFileName();
        this.countMgr = new CounterManager();
        this.writeLock = new SemiSema(1, false);
        // end of user added
    }

    /**
     * Returns the EDKFILEConfiguration object
     *
     * @return  An EDKFILEConfiguration object with EDKFILE
     *          connection information.
     *
     */
    public EDKFILEConfiguration getEDKFILEConfiguration() {
        return config;
    }
```

```
private java.lang.String text;

/**
 * Sets the text
 *
 * @param attrname
 */
public void setText(java.lang.String text) {
    this.text = text;
}

/**
 * Returns the text
 *
 * @return text
 */
public java.lang.String getText() {
    return this.text;
}
private byte[] byteArray;

/**
 * Sets the byteArray
 *
 * @param attrname
 */
public void setByteArray(byte[] byteArray) {
    this.byteArray = byteArray;
}

/**
 * Returns the byteArray
 *
 * @return byteArray
 */
public byte[] getByteArray() {
    return this.byteArray;
}

  public void write() throws EDKFileApplicationException {
      // <Start_User_Code>
      if (text == null)
          throw new EDKFileApplicationException("Error writing text: text is " + text);
      write(text.getBytes());
      // <End_User_Code>
  }
  public void writeBytes() throws EDKFileApplicationException {
      // <Start_User_Code>
      if (byteArray == null) {
        throw new EDKFileApplicationException("Error writing bytes: byteArray is:" +
byteArray);
      }
      write(byteArray);
      // <End_User_Code>
  }

  // user added from this point on
  public synchronized void write(byte[] payload) throws EDKFileApplicationException {
      try {
          File fileDir = new File(directory);
          int percentDIndex = outputFileName.indexOf("%d");

          if (percentDIndex == -1) {
              actualOutputFileName = outputFileName;
              writeContents(
                  payload,
                  fileDir + this.FILE_SEPARATOR + outputFileName);
          } else {
              int fileCount = this.countMgr.getNextCounterCount(getKey());
              String localOutputFileName = null;
              localOutputFileName = outputFileName.substring(
                      0,
                      percentDIndex)
                  + (new PrintfFormat("%d")).sprintf(fileCount)
                  + outputFileName.substring(
                      percentDIndex + 2,
                      outputFileName.length());
              actualOutputFileName = localOutputFileName;
              writeContents(
                  payload,
                  fileDir + this.FILE_SEPARATOR
                  + localOutputFileName);
          }
      } catch (Exception ex) {
          logger.error("Exception occurred in write", ex);
          throw new EDKFileApplicationException(ex.getMessage());
      }

  }

  public void writeContents(byte[] contents, String fullPathOutputFile)
```

```
            throws Exception {

            FileOutputStream os = null;
            try {
                // Block while there is another write in progress.
                if (logger.isDebugEnabled()) {
                    logger.debug("Attempting to acquire a write lock.");
                }
                if (writeLock.acquire()) {
                    if (logger.isDebugEnabled()) {
                        logger.debug("write lock acquired.");
                    }
                } else {
                    logger.error("Unable to acquire a write lock.");
                    throw new Exception("Unable to acquire a write lock.");
                }

                os = new FileOutputStream(fullPathOutputFile);
                if (contents != null) {
                    os.write(contents);
                }
                writeLock.release();
            } catch (Exception ex) {
                logger.error("Exception occurred in writeContents", ex);
                throw ex;
            } finally {
                if (os != null) {
                    os.close();
                }
            }
        }

        public String getKey() {
            return this.directory + this.outputFileName;
        }

    }
```

4 Implement the **<external_application>WebClientApplication** class under webservice sub-folder, and add new classes there if necessary. The following is an example of how this class is implemented.

```
/***************************************************************************
 *
 *          Copyright (c) 2004, SeeBeyond Technology Corporation,
 *          All Rights Reserved
 *
 *          This program, and all the routines referenced herein,
 *          are the proprietary properties and trade secrets of
 *          SEEBEYOND TECHNOLOGY CORPORATION.
 *
 *          Except as provided for by license agreement, this
 *          program shall not be duplicated, used, or disclosed
 *          without written consent signed by an officer of
 *          SEEBEYOND TECHNOLOGY CORPORATION.
 *
 ***************************************************************************/
package com.stc.connector.edkfileadapter.webservice;

/**
 * This class defines the operations made available as
 * webservices for EDKFILE.
 *
 */
import com.stc.connector.edkfileadapter.appconn.EDKFILEApplicationConnection;
import com.stc.connector.edkfileadapter.appconn.EDKFILEApplicationException;
import com.stc.connector.edkfileadapter.appconn.EDKFILEConfiguration;
import org.apache.log4j.Logger;

//user added
import java.io.File;
import java.io.FileOutputStream;
import com.stc.connector.framework.util.SemiSema;
// end of user added

/**
 * Implements EDKFILEClientApplication which exposes
 * operations available for the client application.
 *
 */
public class EDKFILEWebClientApplication {
    private EDKFILEApplicationConnection appConn = null;
    private EDKFILEConfiguration config = null;
    private Logger logger = Logger.getLogger("STC.eWay.edkfile"
                            + getClass().getName());
    // user added
    private static String FILE_SEPARATOR = System.getProperty(
            "file.separator");
    private String actualOutputFileName = null;
    private CounterManager countMgr = null;
```

```
    private SemiSema writeLock = null;
    private String directory = null;
    private String outputFileName = null;
    // end of user added

    /**
     * Constructor.
     *
     * @param appConn   The EDKFILEApplicationConnection instance.
     *
     */
    public EDKFILEWebClientApplication(EDKFILEApplicationConnection appConn) {
        this.appConn = appConn;
        this.config = appConn.getEwayConfiguration();

        // user added
        this.directory = config.getDirectory();
      this.outputFileName = config.getOutputFileName();
        this.countMgr = new CounterManager();
        this.writeLock = new SemiSema(1, false);
        // end of user added
    }

    /**
     * Get the EDKFILEConfiguration object for setting EDKFILE connection
     * information.
     *
     * @return  An EDKFILEConfiguration object for populating EDKFILE
     *          connection information.
     *
     * @throws  EDKFILEApplicationException upon error.
     */
    public EDKFILEConfiguration getEDKFILEConfiguration()
        throws EDKFILEApplicationException {
        return config;
    }

    public EDKFileWriteOutputMessage write(EDKFileTextMessage edkfiletextmessage) throws
EDKFileWSException {
        EDKFileWriteOutputMessage output = new EDKFileWriteOutputMessage();
        output.setStatus(false);
        if (edkfiletextmessage.getEdkfiletext() == null) {
          throw new EDKFileWSException("Error writing: input message is not populated propertly");
      }
        try {
            write(edkfiletextmessage.getEdkfiletext().getBytes());
            output.setStatus(true);
        } catch (Exception e) {
            e.printStackTrace();
            new EDKFileWSException(e.getMessage());
        }

        return output;
    }

    // user added code from here on
    public synchronized void write(byte[] payload) throws Exception {
        try {
            File fileDir = new File(directory);
            int percentDIndex = outputFileName.indexOf("%d");

            if (percentDIndex == -1) {
                actualOutputFileName = outputFileName;
                writeContents(
                    payload,
                    fileDir + this.FILE_SEPARATOR + outputFileName);
            } else {
                int fileCount = this.countMgr.getNextCounterCount(getKey());
                String localOutputFileName = null;
                localOutputFileName = outputFileName.substring(
                        0,
                        percentDIndex)
                    + (new PrintfFormat("%d")).sprintf(fileCount)
                    + outputFileName.substring(
                        percentDIndex + 2,
                        outputFileName.length());
                actualOutputFileName = localOutputFileName;
                writeContents(
                    payload,
                    fileDir + this.FILE_SEPARATOR
                    + localOutputFileName);
            }
        } catch (Exception ex) {
            logger.error("Exception occurred in write", ex);
            throw new Exception(ex.getMessage());
        }

    }

    public void writeContents(byte[] contents, String fullPathOutputFile)
        throws Exception {
```

```
        FileOutputStream os = null;
        try {
            // Block while there is another write in progress.
            if (logger.isDebugEnabled()) {
                logger.debug("Attempting to acquire a write lock.");
            }
            if (writeLock.acquire()) {
                if (logger.isDebugEnabled()) {
                    logger.debug("write lock acquired.");
                }
            } else {
                logger.error("Unable to acquire a write lock.");
                throw new Exception("Unable to acquire a write lock.");
            }

            os = new FileOutputStream(fullPathOutputFile);
            if (contents != null) {
                os.write(contents);
            }
            writeLock.release();
        } catch (Exception ex) {
            logger.error("Exception occurred in writeContents", ex);
            throw ex;
        } finally {
            if (os != null) {
                os.close();
            }
        }
    }

    public String getKey() {
        return this.directory + this.outputFileName;
    }

}
```

5. Browse to the ${**env.STC_ROOT}\connectors\edkfileadapter** folder, and run the following:

   ```
   ant clean install -f connector-build.xml
   ```

   This should build the **edkfile.rar** file and all the other required jar files at the following locations:

   ```
   <STC_ROOT>\BUILD\Modules\connectors\lib\edkfile.rar
   <STC_ROOT>\BUILD\Modules\connectors\lib\edkfile_jca10.rar
   ```

5.2.10 Step 9: Build the .sar File

Run the Apache Ant Build Tool (included in the eDK install) to create the new **edkfileadapter.sar** file.

1. Browse to eWay working directory that you defined in **“Choosing a Working Directory” on page 40**, then run the following:

   ```
   ant clean install -f eway-build.xml
   ```

   This creates the new **edkfileadapter.sar** file in the following location:

   ```
   <STC_ROOT>\build\images\products\edkfileadapter.sar
   ```

5.2.11 Step 10: Upload the New eWay to the ICAN Repository

Perform the following steps to upload the new eWay to the ICAN Repository.

1. Click the **Admin** tab in Enterprise Manager.
2. Click **Browse** and open the **edkfileadapter.sar** file from the following location:

   ```
   <STC_ROOT>\build\images\products\edkfileadapter.sar
   ```

3. Click **Browse** and upload the **license.sar** file.

5.2.12 Step 11: Run the Enterprise Designer Update Center

After you start Enterprise Designer for the first time, you must install all the modules required to run the program.

For detailed information on running the Enterprise Designer Update Center, see the *Seebeyond ICAN Suite Installation Guide*.

5.2.13 Step 12: Creating, Building, and Deploying Sample Projects

The creation and deployment of new sample projects created using an eDK based eWay is beyond the scope of this user's guide; however, detailed information, including examples of how to create and deploy JCE based sample projects are found in the *eGate Integrator Tutorial*.

Chapter 6

Using eDK-Based eWay Java Methods

This chapter provides an overview of the Java classes and methods contained in eDK-based eWays. These methods are used to extend the functionality of the eWay.

What's in this Chapter

- **eWay Development Kit Javadoc** on page 86

6.1 eWay Development Kit Javadoc

The Javadoc is uploaded eDK file (**eWayDevelopmentKit.sar**) and downloaded from the Documentation tab of the Enterprise Manager. To access the full Javadoc:

1. Extract the Javadoc to an accessible folder
2. Double click the **index.html** file.

6.1.1 eDK-Based eWay Classes and Methods

Java methods allow you to set and get information in the eWay Object Type Definitions (OTDs).

The nature of this data transfer depends on the configuration parameters you created in the eDK and set in eGate Enterprise Designer.

eDK-Based eWay Classes

Java methods are organized into related classes. The methods for the eDK-based eWays are organized into the following Java classes:

- AlertCodeMap
- AlertCodes
- AssociateableHandle
- Base64
- Base64DecodingException
- Base64Utils

- BaseMonitorMBean
- ConfigurationHelper
- DirUtils
- EwayActivationSpec
- EwayConfigModelUtil
- EwayConnection
- EwayResourceAdapter
- EwayResourceAdapterImpl
- FileHelper
- JndiJCAObject
- MBeanInfoConfigModelHelper
- MBeansRegistrar
- ManagedConnectionCallback
- MessageManager
- STCActivationSpec
- STCActivationSpecMonitor
- STCActivationSpecMonitorBean
- STCAdapterConfigurationMonitor
- STCApplicationConnectionFactory
- STCConnectionDisabledException
- STCConnectionManager
- STCCreateMBeanException
- STCMBeanNameUtil
- STCMCFMonitor
- STCMCFMonitorBean
- STCManagedConnection
- STCManagedConnectionFactory
- STCManagedConnectionMetaData
- STCManagedConnectionMonitor
- STCManagedConnectionMonitorBean
- STCManagedMaster
- STCManagedSlave
- STCPropertiesInfo
- STCRAMonitor

- STCRAMonitorBean
- STCResourceAdapter
- SemiSema
- StringHelper
- ZIPUtils

Chapter 7

Adding and Sending Custom Alert Messages

This chapter describes how to add and send custom alerts using the Resource Adapter framework.

What's in this Chapter

- **eDK Alerts** on page 89
- **Adding eWay Specific Alert Message Codes** on page 90
- **Sending eWay Specific Alerts** on page 91

7.1 eDK Alerts

In the ICAN Suite, an Alert is triggered when a specified condition occurs in a Project component. The condition might be some type of problem that must be corrected. For example, an Alert might indicate that a SeeBeyond Integration Server is no longer running.

Enterprise Manager enables you to monitor Alerts. In the ICAN Monitor component of Enterprise Manager, you can view detailed information about the Alerts and mark them as observed or resolved. The *eGate Integrator System Administration Guide* describes how to access and use the ICAN Monitor.

Note: *The Alert Agent User's Guide describes how the Alert Agent can monitor both predefined Alerts and custom Alerts. The "Collaboration Definitions (Java)" chapter in the eGate Integrator User's Guide describes how to create custom Alerts at design time.*

eDK Alerts are managed through the <new eWay>AlertCodes.java file, which is located under:

```
<Output-Location>\connectors\<new eWay>\src\java\com\stc\
connector\<new eWay>\alerts
```

Implementation of the <new eWay>AlertCodes class is required to display your eWay specific Alerts. Alert codes used in the eDK include:

```
<new eWay>_EWAY_STARTED
<new eWay>_EWAY_RUNNING
<new eWay>_EWAY_STOPPING
<new eWay>_EWAY_STOPPED
<new eWay>_EWAY_SUSPENDING
<new eWay>_EWAY_SUSPENDED
```

7.2 Adding eWay Specific Alert Message Codes

This section describes how to add custom alerts using the Resource Adapter framework.

7.2.1 Java Code Changes

Use the following method on the monitor mbean object to send alerts:

```
public void sendAlert(String alertMsgCode,
                      String[] alertMsgCodeArgs,
                      String alertMsg,
                      int severity);
```

The user can get an instance of the monitor mbean object by invoking the appropriate getter method in: <external_application_name>EwayConnection.java

7.2.2 What to Pass for alertMsgCode and alertMsgCodeArgs

You need to create a properties file using the localization resource bundle naming convention.

As an example:

```
FILE_en_US.properties (for US English),
FILE_fr_FR.properties (French)
FILE_fr_CA.properties (Canadian French)
```

Note that the file naming convention specifies a string corresponding to your eway name. The same prefix string (e.g. FILE) must match the prefix of the alert message code variables contained in the Properties file. The contents of the Properties file will be your alert code message variable and the alert message.

As an example:

```
FILE-REN000001="Unable to rename file {0}"
FILE-WRT000001="Unable to write output file {0}."
```

Note that the placeholders in the message above are (convention {0}...{1}). These are specified in the alertMsgCodeArgs argument to sendAlert. So, a sample send alert call using this would look like:

```
String [] args =  { "myfilename" };
String alertMsg = "Unable to rename file " + ... + "Skipping ..." ;
sendAlert("FILE-REN000001",
          args,
```

```
            alertMsg,

com.stc.eventmanagement.NotificationEvent.SEVERITY_TYPE_CRITICAL);
```

To isolate where you update the alert codes (first argument above), you can create a class containing the constants corresponding to them.

As an example:

```
public class {EwayName}AlertCodes {

    /**
     * File rename alert
     * <code>Unable to rename file {0}</code>
     * Params:
     *      {0} file name
     */
    public static final String FILE_ASRENAMEFAILED = "FILE-
ASRENAMEFAILED000001";

}
```

The sendAlert call appears as follows:

```
sendAlert(MessageCodes.FILE_REN_01,
            args,
            alertMsg,

com.stc.eventmanagement.NotificationEvent.SEVERITY_TYPE_CRITICAL);
```

7.2.3 Installing Alert Code Properties Files (install.xml changes)

Custom Alert codes are installed automatically when the eWay is uploaded to the ICAN Repository. The following file must be edited to add custom Alert codes:

```
<output_folder>\eways\<eway_adapter_name>\install\alertcodes\<exte
rnal_application_name>.properties
```

7.3 Sending eWay Specific Alerts

This section describes how to send custom alerts using the Resource Adapter framework.

For alerts that are associated with your managed connection, make your EwayConnection class implement the following:

```
com.stc.connector.management.STCManagedSlave
```

This interface includes the **setMBean()** method, which is called from the associated STCManagedConnection. This provides access to the MBean class, which encapsulates the **sendAlert()** methods. Use the following:

```
public void sendAlert(String alertMsgCode,
                      String[] alertMsgCodeArgs,
                      String alertMsg,
                      int severity);
```

Valid values for severity are:

```
com.stc.eventmanagement.NotificationEvent.SEVERITY_TYPE_CRITICAL
com.stc.eventmanagement.NotificationEvent.SEVERITY_TYPE_INFO
com.stc.eventmanagement.NotificationEvent.SEVERITY_TYPE_MAJOR
com.stc.eventmanagement.NotificationEvent.SEVERITY_TYPE_MINOR
com.stc.eventmanagement.NotificationEvent.SEVERITY_TYPE_WARNING
```

The EventManagement API is called by the above method. See theEventManagement module for details on this API.

For alerts associated with your ActivationSpec, do the same as above for your EwayActivationSpec class (implement STCManagedSlave).

Chapter 8

Appendix A

What's in this Chapter

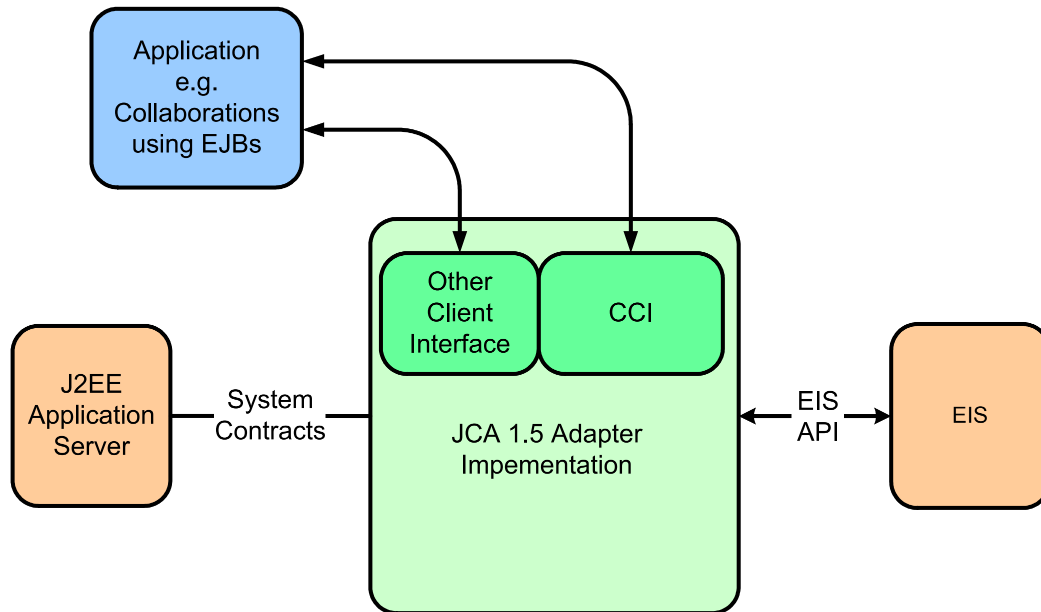
- **J2EE Connector Architecture Overview** on page 93
- **RA Framework Class Diagram** on page 94
- **RA Framework Sequence Diagram** on page 96
- **Client Application Sequence Diagram** on page 97
- **Application Connection Interfaces** on page 98
- **eWay Connection Interfaces** on page 99

8.1 J2EE Connector Architecture Overview

The J2EE Connector Architecture specifies how to develop resource adapters that are used to interact with Enterprise Information Systems (EIS). It describes the interfaces between the J2EE Application Server and the resource adapter which provide for transaction management, connection management, security management, work management, and life cycle management. These Application server/resource adapter interfaces are also referred to as System Contracts, (see **Figure 27 on page 94**).

The Connector architecture also describes the client interface to the resource adapters. Figure 27 illustrates allowed client interfaces. The resource adapter client (normally an EJB) is shown interacting with the resource adapter either through the Client Connection Interface (CCI) or the SeeBeyond Application Connection (AppConn) interface.

Figure 27 J2EE Connector Architecture Overview



8.2 RA Framework Class Diagram

The Resource Adapter (RA) framework is best specified in a UML class and sequence diagram.

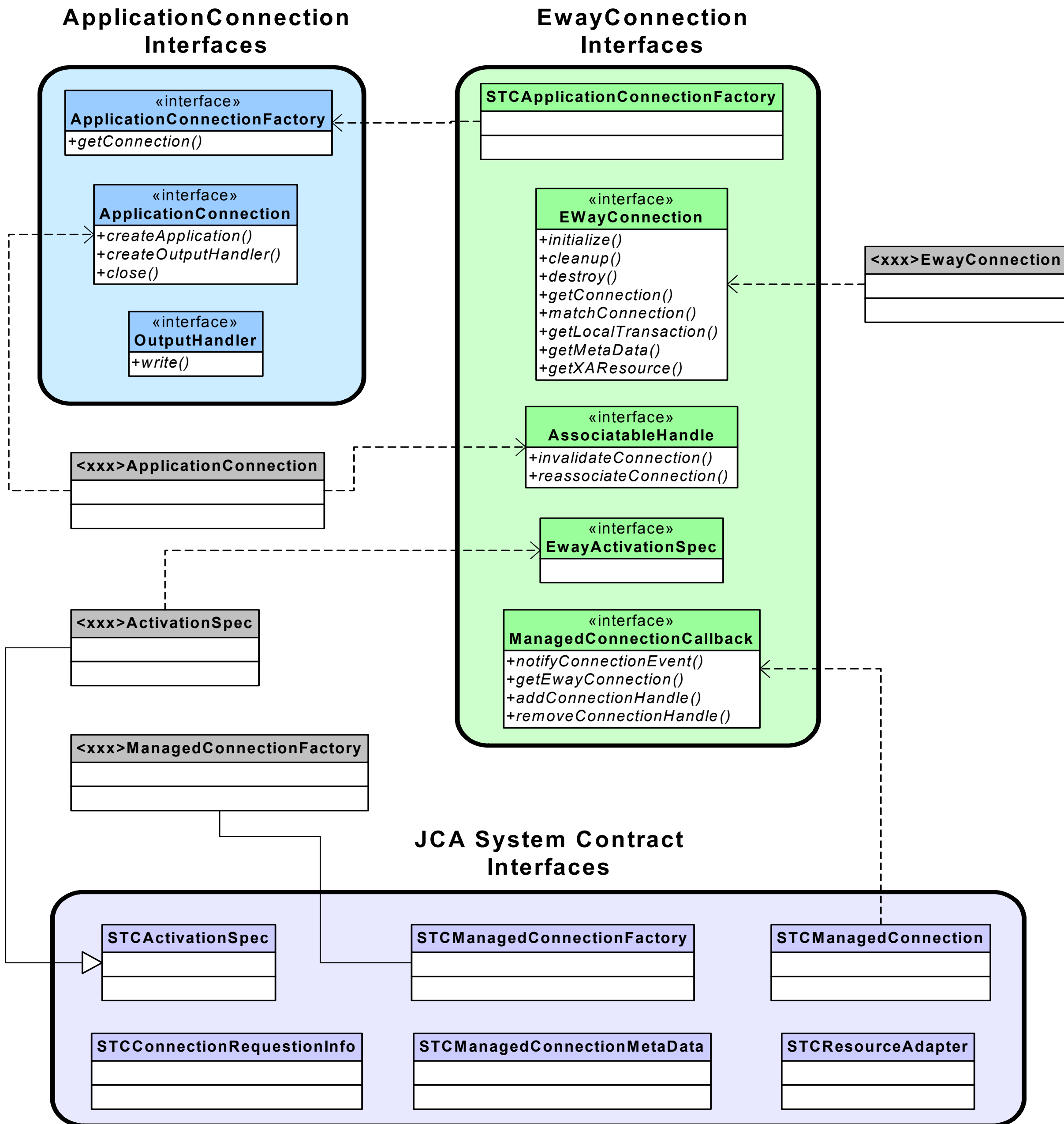
The following interfaces represent the core eDK tool.

- ApplicationConnection Interfaces
- EwayConnection Interfaces
- JCA System Contract Interfaces

The following class files contain methods to be implemented in the eWay.

- <xxx>ApplicationConnection
- <xxx>ApplicationSpec
- <xxx>EwayConnection
- <xxx>ManagedConnectionFactory
- <xxx>Application

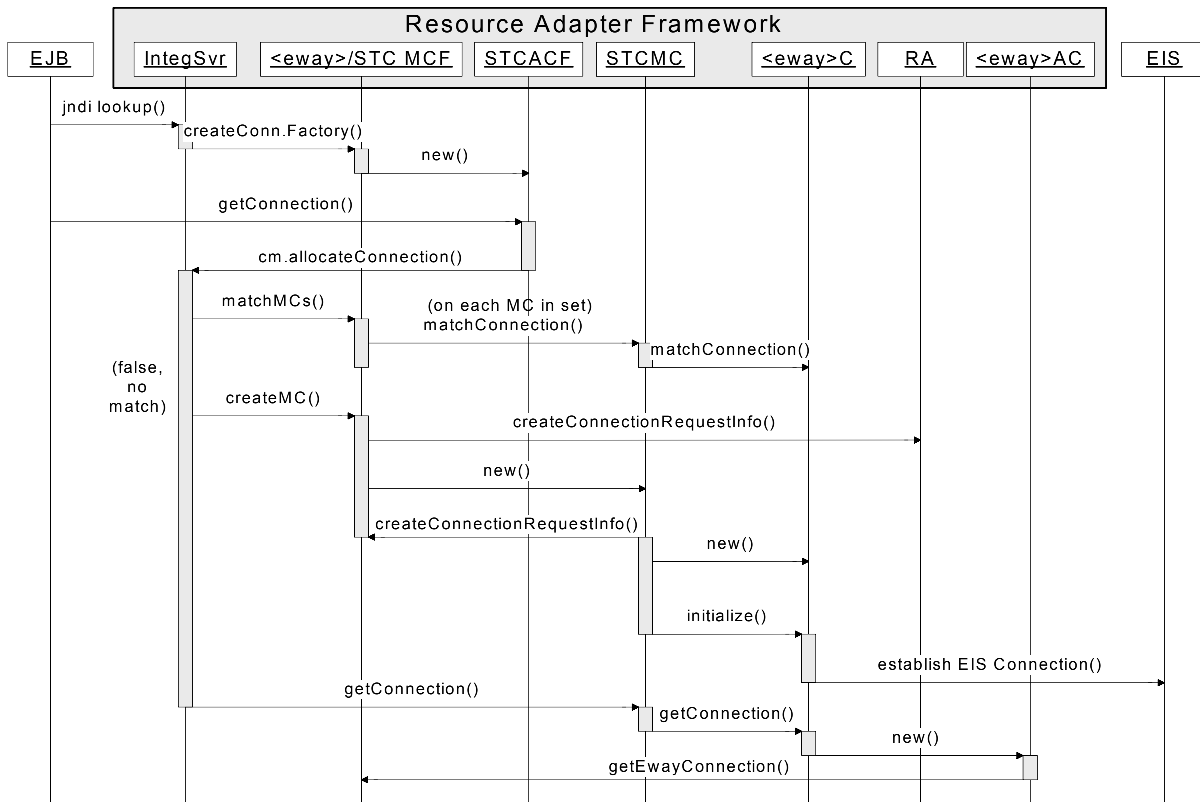
Figure 28 RA Framework Class Diagram



8.3 RA Framework Sequence Diagram

The following diagram describes the client interaction with a resource adapter using the RA framework.

Figure 29 RA Framework Sequence



The RA framework sequence occurs as follows:

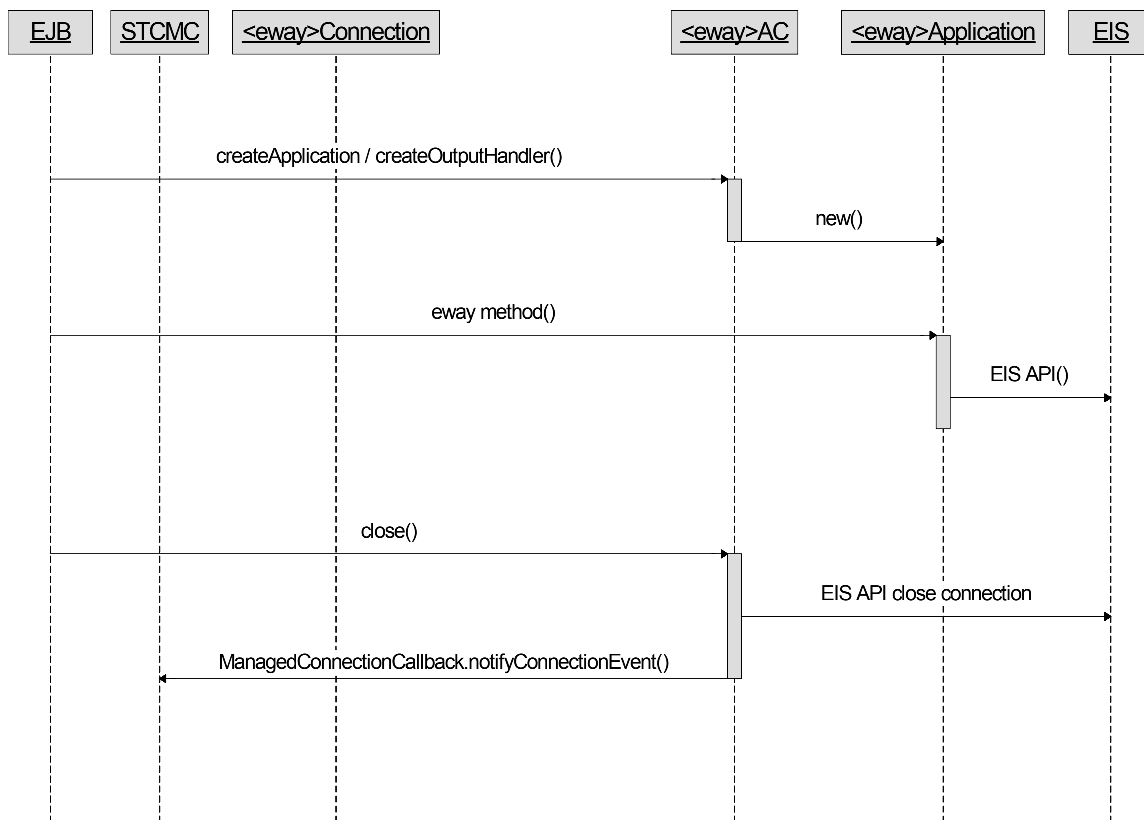
1. The EJB client performs a JNDI lookup to obtain a client connection factory (**ApplicationConnectionFactory**).
2. The Application Server (Integration server) uses the RA framework managed connection factory to create an **ApplicationConnectionFactory** which is passed back to the EJB client's JNDI lookup.
3. The EJB client obtains an ApplicationConnection by calling the **getConnection()** method on the **ApplicationConnectionFactory**.
4. The Integration server and RA framework classes interact by first trying to obtain a matching connection from the connection pool. If a matching connection is not found, a new connection is requested by the Integration server from the resource adapter. Note that a **ConnectionRequestInfo** object containing the connection configuration properties is used when checking for a match.

5 The **EwayConnection** interface implementation is used when establishing a physical connection to the EIS; the initialize method is normally where this is performed. It is also used in connection matching via the **matchConnection()** method. The **getConnection()** method is used to return the established connection.

8.4 Client Application Sequence Diagram

The following diagram shows the interaction after the EJB client has obtained a resource adapter connection.

Figure 30 JCA Framework Sequence



The JCA sequence occurs as follows:

1 The **createApplication()** method is called—once an ApplicationConnection is obtained—to obtain the eWay-specific Application object which exposes the methods interacting with the EIS.

2 The EJB client calls the various methods in the eWay's Application object

3 The EJB's resource adapter connection is closed via the close method in the ApplicationConnection after completing execution of the method calls.

4. The Connector architecture provides a notification mechanism between the Application server and the resource adapter for connection events. These are hidden in the RA framework.

8.5 Application Connection Interfaces

SeeBeyond's common client interface (CCI), includes the following interfaces:

ApplicationConnectionFactory

```
public interface ApplicationConnectionFactory
    extends Referenceable, Serializable
{
    public abstract ApplicationConnection getConnection()
        throws ApplicationConnectionException;
    public abstract ApplicationConnection getConnection(Properties 
properties)
        throws ApplicationConnectionException;
}
```

ApplicationConnection

```
public interface ApplicationConnection
{
    public abstract void close()
        throws ApplicationConnectionException;
    public abstract Object createApplication(String s)
        throws ApplicationException;
    public abstract OutputHandler createOutputHandler()
        throws ApplicationException;
}
```

OutputHandler

```
public interface OutputHandler
{
    public abstract void write(byte abyte0[])
        throws ApplicationException;
    public abstract void write(OutputStream outputstream)
        throws ApplicationException;
}
```

ApplicationConnectionException

```
public class ApplicationConnectionException extends Exception
{
    public ApplicationConnectionException()
    {
    }
    public ApplicationConnectionException(String msg)
    {
        super(msg);
    }
}
```

ApplicationException

```
public class ApplicationException extends Exception
{
    public ApplicationException()
    {
```

```
        }
    public ApplicationException(String msg)
        {
            super(msg);
        }
    }
```

8.6 eWay Connection Interfaces

The eWay Connection Interfaces are located in the following package:

```
package com.stc.connector.framework.client;
```

AssociatableHandle

```
public interface AssociateableHandle
{
    public abstract void invalidateConnection()
        throws ResourceException;
    public abstract void
reassociateConnection(ManagedConnectionCallback
managedconnectioncallback)
        throws ResourceException;
}
package com.stc.connector.framework.eway;
```

EwayConnection

```
public interface EwayConnection
{
    public abstract void initialize(ManagedConnectionCallback
managedconnectioncallback, Subject subject, ConnectionRequestInfo
connectionrequestinfo)
        throws ResourceException;
    public abstract void cleanup()
        throws ResourceException;
    public abstract void destroy()
        throws ResourceException;
    public abstract AssociateableHandle getConnection(Subject
subject, ConnectionRequestInfo connectionrequestinfo)
        throws ResourceException;
    public abstract boolean matchConnection(Subject subject,
ConnectionRequestInfo connectionrequestinfo);
    public abstract LocalTransaction getLocalTransaction()
        throws ResourceException;
    public abstract ManagedConnectionMetaData getMetaData()
        throws ResourceException;
    public abstract XAResource getXAResource()
        throws ResourceException;
}
package com.stc.connector.framework.eway;
```

ManagedConnectionCallback

```
public interface ManagedConnectionCallback
{
    public abstract void notifyConnectionEvent(ConnectionEvent
connectionevent);
    public abstract EwayConnection getEwayConnection();
```

```
    public abstract void addConnectionHandle(AssociateableHandle
associateablehandle);
    public abstract void removeConnectionHandle(AssociateableHandle
associateablehandle);
}
```

EwayActivationSpec

```
public interface EwayActivationSpec
{
    public abstract void initialize(BootstrapContext
bootstrapcontext);
    public abstract void validate(STCPropertiesInfo
stcpropertiesinfo)
        throws InvalidPropertyException;
    public abstract void endpointActivation(MessageEndpointFactory
messageendpointfactory, STCPropertiesInfo stcpropertiesinfo)
        throws NotSupportedException;
    public abstract void endpointDeactivation(MessageEndpointFactory
messageendpointfactory);
    public abstract XAResource getXAResource()
        throws ResourceException;
}
```

Chapter 9

Appendix B

What's in this Chapter

- **Generating eDK Code by Command Line** on page 101
- **eDK Definition File** on page 102

9.1 Generating eDK Code by Command Line

In addition to using the eWay Development Kit Build Tool to generate code, you can also create eWays via the command line.

To Generate Code by Command Line:

1. Run the **env.bat** file located at the root level of the extracted eDK folder to set up the implementation environment.
2. Create the eDK definition **<adapter_name>.xdef** file using the eWay Development Kit Build Tool. See **"Steps Required to Build an eWay" on page 18**. Alternately, you can also create a definition file manually. See **"eDK Definition File" on page 102**.

   An eDK definition file contains all required eWay metadata, including eWay name, external application name, third-party jar files, operations to be exposed in Java and BPEL Collaborations, etc.
3. Create a configuration template **<adapter_name_template.xml** file using the eWay Development Kit Build Tool.
4. Run the following command from the **${env.STC_ROOT}\eways\edk\devtools** folder.

```
ant run -Dxmlfilename=<location_of_xdef_file>
```

   This generates the "connectors" and "eways" folder under the output directory (as specified in the definition file).

Choosing Your Working Directory

As an optional step, you can choose to either work from the generated output folder, or choose a new working directory by copying the following folders:

- Copy the **<newEway>adapter** folder, located under the **connectors** folder in the output directory (as specified in the definition file) to:

```
${env.STC_ROOT}\connectors\
```

- Copy the **<newEway>adapter** folder, located under the **eways** folder in the output directory (as specified in the definition file) to:

```
${env.STC_ROOT}\eways\
```

9.2 eDK Definition File

The eDK definition file is an **.xdef** file that stores metadata information about an eWay. In order for the eWay Development Kit Build Tool to generate eWay shell code properly, information about the eWay needs to be gathered. This can be achieved by either using the eWay Development Kit Build Tool, or by manually creating the eDK definition file. This appendix discusses how to create an eDK definition file manually.

All information about the eDK definition itself, such as eDK user, version number, revision number, date created, date modified, can be stored as attributes in the MetaData element. Note that the eWay user does not need to populate the date created/modified fields; this information is automatically generated in the proper date format by the eWay Development Kit Build Tool.

The Properties element contains the following basic information needed to build an eWay:

- **Eway_adapter_name (required):** name of the eWay. This is used for the directory name under connectors and eways folder.
- **External_application_name (required):** name of the external application
- **Output_directory (required):** output folder for the generated eWay shell code
- **Log_File_Name (optional):** When specified, all eWay generation loggings will be saved to this file
- **Resource_locations (required):** This element allows the user to specify resource file locations, namely configuration template location, and optional icon file locations.

Resource file locations include:

- **Config_template_location:** file location of the configuration template.
- **Icon_file_location:** This element allows the user to specify icon files to be used for the eWay. Other eWay icons not specified in this element, created with the eWay Development Kit Build Tool include:
  - **External_app_icon:** file location of the External System icon to be displayed on eDesigner Environment and Project Explorer.
  - **Connectivity_map_icon:** file location of the External System icon to be displayed on eDesigner Connectivity Map.
  - **BPEL_invoke_icon:** file location of the Business Activity icon to be displayed on eInsight Business Process canvas. This icon is used for all outbound business activities.
  - **BPEL_receive_icon:** file location of the Business Invoke Activity icon to be displayed on eInsigh Business Process canvas. This icon will be used for all inbound business activities.

The **JCE_client_interface** describes all inbound and/or outbound operations to be exposed in SeeBeyond ICAN Java Collaboration Editor. The **JCE_client_interface** allows three elements to be defined.

The **Operation** element defines the operation to be exposed in Java Collaboration Editor. For each operation, it is required to define the operation name and mode. The value of the mode attribute can be either "inbound" or "outbound". Users can also define the description of the operation using the "description" attribute. Operation elements allow the user to specify input, output, and exception, with following sub-elements:

- **Input element:** has the required name and type attributes. In case of Java Collaboration operations, any number of inputs can be added to the operation.
- **Output element:** has the required type attribute.
- **Exception element:** has the required type attribute.

The **Attributes** element allows the user to define all attributes for a Java Collaboration OTD. This element can hold as many attributes as the user needs to define for a JCE OTD. Each Attribute element requires a name and type (**.xml**) attribute.

The **Types** element contains all user defined data types and/or user defined class definitions. This element can hold any number of user defined types.

- **User_defined_data_container elements:** allows users to specify only data elements. It has the required "Element" (**.xml**) sub-element, which in turn has the required "name" and "type" attributes. This definition allows the eWay Development Kit Build Tool to generate java bean classes for each "Element" defined. The "type" attribute can also refer to the name of another "User_defined_data_container" element.
- **User_defined_class elements:** allows users to specify a java class definition. It allows the "Element" sub-element, "Attribute" subelement and "Method" subelement. "Method" element behaves very much like the "Operation" element. The reason for User_defined_class elements is to allow the user to specify the "inbound OTD" type for JCE inbound operations. It is ONLY intended to be used for an "input" type definition of a JCE inbound operation.

The **BPEL_client_interface** describes all inbound and/or outbound operations to be exposed in SeeBeyond ICAN Business Process Editor. It contains very similar elements as in **JCE_client_interface**. A notable difference is that only one "input" element is allowed for any BPEL operation. Also, no "User_defined_class" element is allowed in the "Types" element.

Index

Symbols

.sar file
building **40**
<adapter_name>.xdef **39**
<adapter_name>_template.xml **39**

A

About **17**
additional files created during installation **13**
Alerts
adding eWay specific message codes **90**
Enterprise Manager **89**
installing alert code property files **91**
sending **91**
triggering **89**
what to pass **90**
Apache Ant build tool **17**
AppConn client interface **49**
Application class **50**
ApplicationConnection class **50**
ApplicationConnectionFactory class **50**
attributes
creating **30**

B

BPEL versus JCE user defined class files **29**
Building a .sar file **40**
building an eWay
creating and specifying the eWay **24**
e-mail format conventions **19**
licensing process **19**
naming restrictions **25**
obtaining a valid license **18**
requesting a new license **19**
requesting a reissued license **21**
setting environment variables **23**
starting the build tool **23**
steps required **18**

C

choosing a working directory **40**
class files created
EwayActivationSpec **47**
ApplicationConnection **46**
ClientApplication **46**
EwayConnection **46**
WebClientApplication **46**
Classes
eDK-based **86**
cleanup() method **55**, **56**
Client Application
sequence diagram **97**
Client Connection Interface (CCI) **93**
Client Interface **69**
ClientApplicationImpl class **80**
code generation
folders created after **41**
Code Generation components called during deployment **46**
Code Generation tab
running the **38**
saving work **39**
code generator
running **75**
Codelets **46**
command line
generating code by **101**
Config Template tab
deleting sections and properties **37**
disabling and enabling sections **37**
configuration properties
inbound **71**
outbound **73**
configuration template **39**
defining **71**
connectors filder **41**
connectors folder **16**, **76**
alerts folder **41**
appconn folder **41**
ewayconn folder **42**
webservice folder **42**
connectors folder (src_jca15)
appconn folder **42**
ewayconn folder **42**
conventions, document **9**
creating attributes **30**
creating methods (JCE) **28**
creating operations **30**
creating user defined class files **29**

D

date created field **26**
deploying a project **85**
description of eWay **25**

destroy() method **56**
document conventions **9**

E

eDK definition file **39**
.xdef file **102**
understanding the **102**
EIS connections
automatic mode **57**
dynamic connection **57**
establishing **57**
overriding configurations **59**
env.bat file **17**
Environment Variables
setting **67**
ESR
74717 **12**
78987 **14**
eWay
creating **68**
implementing **77**
eWay comments field **26**
eWay components **47**
eWay creation date field **26**
eWay description **25**
eWay Development Kit build tool
starting **67**
eWay implementation environment **15**
eWay interfaces
attribute **28**
defining the BPEL interface **27**
defining the Java interface **26**
methods **28**
naming restrictions **28**
operation **28**
user defined **28**
eWay Name **25**
eWay name **25**
eWay version number **26**
EWayConnection class **77**
EwayConnection class **58**
EwayConnection interface **55**
eways folder **16, 41, 76**
codegen **43**
config **43**
egategui **43**
install **43**
module **43**
Thirdpartylib **43**
external application name **25**
external system sections and properties **37**

G

General tab **24**
change history **26**
description **25**
eWay Name **25**
extenal applicaton name **25**
Icons used in the eDK eWay **25**
imported files **26**
maximum icon size **25**
name and description **25**
getConnection() method **55**
GUI code for plugging into the Enterprise Designer **45**

I

ICAN versions - supported **12**
icons
maximum size **25**
implementation environment **17**
setting up the **17**
implementing and building shell code **40**
Implementing XA **56**
getXAResource() method **56**
importing a project **65**
initialize() method **55**
Insallation
installing ESR 78987 **14**
installing sample Projects **14**
Installation
directories created after **13**
installation **11–14**
additional files created during **13**
interfaces
Application Connection **98**
eWay Connection **99**
eWay connection **98**

J

J2EE
connector architecture **93**
J2EE connector architecture **93**
J2EE Connector Architecture Resource Adapter **45**
J2EE resource adapter **46**
Java Collaboration Wizard **45**
Java naming restrictions **28**
Javadoc
eWay Development Kit **86**
Javadocs **14**
generating **63**
JCA (J2EE Connector Architecture) **7**
JNI code

suggested conventions for writing 47

L

last modified field 26
LD_LIBRARY_PATH variable (JNI code) 48
license file 67

M

Managed Connection Factory
implementation example 51
Managed Connection Factory class 55
ManagedConnection.cleanup() method 56
matchConnection() method 55
Message-driven Bean (MDB) 46
methods
eDK-based 86
Javadoc 86
minimum monitor resolution 11
monitor resolution - minimum 11

N

naming restrictions
alphabetic letters 28
alphbetic letters 25
digit 28
digits 25

O

opening saved work 39
operating systems - supported 11
operations
creating 30

P

PATH variable (JNI code) 48
properties
inbound 71
outbound 73

R

RA framework 17
about the 17
Resource Adapter
about the 17
class diagram 95
framework 94
sequence diagram 95, 96
running the env.bat file 17

S

sample projects 65
deploying 85
importing 65
locations 65
overview 66
SAR file
building 84
SeeBeyond Application Connection (AppConn)
interface 93
shell code 40
source control
applying 62
Specifying Configuration Properties 59
Connectivity Map eWay properties 60
external system properties 61
state
maintaining and persisting 63
supported ICAN versions 12
eGate version 5.0.5, 5.0.4 12
Supported Operating Systems 11
supported operating systems 12

T

third-party .jar
wrapping 62
third-party files 26

U

Update Center
running 85

V

version 26

W

WebClientApplication class 82
writing JNI code 47