SeeBeyond™ eBusiness Integration Suite

XML Toolkit

Release 4.5.3

SᴇᴇBᴇʏᴏɴᴅ™

Contents

List of Figures

List of Tables

Introduction

This chapter introduces you to this guide, its general purpose and scope, and its organization. It also provides sources of related documentation and information.

1.1 Document Purpose and Scope

This guide explains how to use the SeeBeyond Technology Corporation™ (SeeBeyond™) XML Toolkit with the SeeBeyond e*Gate™ Integrator system. This explanation includes:

- Product overviews for both the Java and Monk programming language XML Toolkit versions

- Installation

- Using the Document Type Definition (DTD) and Extensible Markup Language (XML) Schema Builders (including the Monk Converter versions)

- XML-to-Java and XML-to-Monk data mapping

- Implementation and examples

Important: *Any operation explanations given here are generic, for reference purposes only, and do not necessarily address the specifics of setting up and/or operating individual e*Gate systems.*

This document does not contain specific information on the e*Gate Integrator system. For information on this and related topics, see **"Supporting Documents" on page 14**.

1.2 Intended Audience

The reader of this document is presumed to be a developer or system administrator with the following prerequisites and/or skill sets:

- Familiarity with the e*Gate system and its associated Java and Monk Event Type Definition (ETD) Editors

- Thorough knowledge of Windows operations and administration

- Familiarity with Windows-style graphical user interface (GUI) operations
- Familiarity with XML, DTD, and XML Schema concepts and functions
- Familiarity with Java and/or Monk programming

1.3 Organization of Information

This document is organized topically as follows:

- **Chapter 1 "Introduction"** gives a general preview of this document, its purpose, scope, and organization.

- **Chapter 2 "XML Toolkit Summary"** provides a brief introduction to XML, DTDs, XML Schemas, and the XML Toolkit product; also shows a list of system and product requirements.

- **Chapter 3 "Java XML Toolkit Overview"** provides a brief description of the Java XML Toolkit; also gives some common Java mapping characteristics.

- **Chapter 4 "Java DTD Builder"** explains how to use the DTD Builder, including how to use the Builder, XML-to-Java data mapping, XML namespaces, and product capabilities.

- **Chapter 5 "Java XML Schema Builder"** explains how to use the XML Schema Builder, including how to do the conversion, World Wide Web Consortium (W3C) XML-to-Java data mapping, and a description of the results.

- **Chapter 6 "Java Conversion Examples"** provides before-and-after Java conversion examples of DTDs and XML Schemas, with comments.

- **Chapter 7 "Registry API for XML Schema Metadata"** explains how to use the e*Gate Registry application programming interface (API) for Java XML Schema metadata.

- **Chapter 8 "Monk DTD Converter"** explains how the DTD Converter creates ETD (**.ssc**) files from existing DTD files and how to use the Builder.

- **Chapter 9 "Monk XML Schema Converter"** discusses the XML Schema Converter, another tool in the XML Toolkit, which enables users to convert W3C XML Schema files (**.xsd**) to ETD files.

- **Chapter 10 "XSLT Collaboration Service"** describes how to create and use the XSLT Collaboration Service (XSLT stands for Extensible Stylesheet Language Transformations) and how to create its associated Collaboration Rules.

- **Chapter 11 "Monk DTD Generator"** provides a service that enables users to convert ETD files to XML files with a **.dtd** extension. This chapter explains how this functionality works with e*Gate Integrator.

- **Chapter 12 "$event->xml Monk Function"** provides the ability to transform non-XML messages into XML messages dynamically by taking a parsed representation of the non-XML event and generating an XML message; chapter explains how to use this function.

▪ **Chapter 13** **"Monk Capabilities and Troubleshooting"** explains the basic capabilities of some Monk XML Toolkit features and how to troubleshoot them.

1.4 Writing Conventions

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

Hypertext Links

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

For information on these and related topics, see **"Parameter, Function, and Command Names" on page 14**.

Command Line

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

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

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

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

Code and Samples

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

```
Configuration for BOB_Promotion
```

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

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

Notes and Cautions

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

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

User Input

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

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

File Names and Paths

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

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

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

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

Parameter, Function, and Command Names

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

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

The Monk function **iq-put** places an Event into an IQ.

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

1.5 Supporting Documents

The following SeeBeyond documents provide additional information about the XML Toolkit explained in this guide:

- *Creating an End-to-end Scenario with e*Gate Integrator*
- *e*Gate Integrator Alert and Log File Reference Guide*
- *e*Gate Integrator Collaboration Services Reference Guide*
- *e*Gate Integrator Installation Guide*
- *e*Gate Integrator Intelligent Queue Services Reference Guide*
- *e*Gate Integrator System Administration and Operations Guide*
- *e*Gate Integrator User's Guide*
- *SeeBeyond eBusiness Integration Suite Deployment Guide*
- *SeeBeyond eBusiness Integration Suite Primer*
- *SeeBeyond JMS Intelligent Queue User's Guide*

See the *SeeBeyond eBusiness Integration Suite Primer* for a complete list of SeeBeyond eBI Suite-related documentation. You can also refer to the appropriate Microsoft Windows or UNIX documents, if necessary.

Note: *For information on how to use a specific add-on product (for example, an e*Way™ Intelligent Adapter or IQ™), see the user's guide for that product.*

1.6 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

XML Toolkit Summary

This chapter provides an overview of the Extensible Markup Language (XML), Document Type Definitions (DTDs), and XML Schemas, including a general description of the e*Gate XML Toolkit and its components. It also gives system requirements and installation instructions for the XML Toolkit product.

2.1 Introduction to XML

XML was developed by a working group of the World Wide Web Consortium (W3C). The result is an international standard for electronic document exchange, a markup-design language that can be used to create other markup languages. Most applications of XML are:

- Documents
- Data exchange
- Database connectivity

Using XML to create documents is similar to creating HTML documents. Where HTML is a fixed, standard markup language, XML is more robust and flexible, and is far more extensible. XML users can design their own markup languages to suit individual requirements and continue to add additional features as desired. Of course, the derived markup languages and additional features must conform to XML rules.

For additional information, refer to any standard XML reference and/or user guide, including those available on the Internet.

Note: *For more information on W3C XML and DTD specifications, point your Web browser to:*
http://www.w3.org/TR/REC-xml

2.1.1 DTD Overview

The purpose of DTDs is to validate XML documents. DTDs contain specifications for the valid XML syntax, structure, and format set up by the user-defined markup elements. In other words, XML is a piece of structured data and DTD defines the structure. The markup language's DTD specifications are necessary to interpret data in this type of structured format.

A DTD is the blueprint of an XML document. It can be used to help with constructing and interpreting XML documents in the desired way. You can also use DTDs when constructing/parsing XML documents.

2.1.2 XML Schema Overview

The concept of the XML Schema is basically the same as that of the DTD. Developed later than the DTD, the XML Schema serves the same purpose, except that it has more flexibility, complexity, and a greater variety of different formats used to define data.

2.2 XML Toolkit Versions

In conjunction with e*Gate, the XML Toolkit creates e*Gate Event Type Definition (ETD) files from existing DTD (**.dtd**) and XML Schema (**.xsd**) files. You can create these files in either the Java or Monk programming language. These files can then be used to parse XML documents and are fully compatible with the e*Gate system.

The XML Toolkit is an add-on feature separate from core e*Gate. It creates the following types of ETD files:

Java Version

- With the extension **.xsc**.

Monk Version

- With the extension **.ssc**.

2.3 Supported Operating Systems

The CPU and RAM requirements for the XML Toolkit are the same as those for core e*Gate, including all Java-related requirements. See the current *e*Gate Integrator Installation Guide* for a list of these requirements.

The XML Toolkit is available on the following operating systems:

- Windows 2000, Windows 2000 SP1, and Windows 2000 SP2
- Windows NT 4.0 SP6a
- Japanese Windows 2000, Windows 2000 SP1, and Windows 2000 SP2
- Japanese Windows NT 4.0 SP6a
- Traditional Chinese Windows 2000, Windows 2000 SP1, and Windows 2000 SP2
- Traditional Chinese Windows NT 4.0 SP6a

2.4 System Requirements

To use the XML Toolkit, you need the following:

- An e*Gate Participating Host, version 4.5 or later
- A TCP/IP network connection
- 1.5 MB of disk space

2.5 External System Requirements

XML Schema Standards

Java Standards

The World Wide Web Consortium (W3C) XML Schema standards used by this Java XML Toolkit version are:

- October 2000
- May 2001

For more information, see:

"XML Schema Versions: Java" on page 34

Monk Standards

The W3C XML Schema standards used by this Monk XML Toolkit version are:

- April 2000

For more information, see:

"XML Schema Versions: Monk" on page 86

2.6 Windows Installation

This section explains how to install the XML Toolkit on a Windows NT or Windows 2000 system. For more information, including installation procedures and total e*Gate system/disk space requirements, see the *e*Gate Integrator Installation Guide*.

2.6.1 Pre-installation

1 Exit all Windows programs before running the setup program, including any anti-virus applications.

2 You must have Administrator privileges for both the workstation and the Registry Host to install the XML Toolkit.

2.6.2 ## Installation Procedure

To install the XML DTD Converter on a Windows system

1 Log in as an Administrator on the workstation on which you want to install the XML Toolkit.

2 Insert the e*Gate installation CD-ROM containing the XML Toolkit into the CD-ROM drive.

*Note: For information on how to tell which disk has the XML Toolkit, see the **e*Gate Integrator Installation Guide**.*

3 If you are installing this add-on in the SeeBeyond eBI Suite using the master install wizard, the installation for the add-on applications will launch automatically. Otherwise, do the following steps:

   • On the e*Gate CD-ROM, navigate to the **\Setup\addons** folder.

   • Double-click the **Setup** icon.

4 Follow the online prompts in the "InstallShield Wizard" to navigate though the introductory windows and to accept the license agreement.

5 When the **User Information** dialog box appears, type your name and company name.

6 Select the destination directory.

*Note: Be sure to install the XML Toolkit files in the suggested **\client** installation directory. The installation utility detects and suggests the appropriate installation directory. **Unless you are directed to do so by SeeBeyond support personnel, do not change the suggested "installation directory" setting.***

7 Select the component that you want to install, in this case, the **ETD Builders,** then click **Change**.

8 Select the **XML Toolkit** and click **Continue**, then **Next**.

9 Follow the on-screen prompts to select a program folder and confirm your selection.

10 You are prompted for the Registry Host to which this add-on is to be installed. Enter the Registry Host's name (if installing to a distributed Registry system, enter the primary Registry Host's name) and click **Next**.

11 You will be prompted for the Administrator name and your password. Enter the requested information and click **Next**.

12 Select the platform that the selected Registry Host(s) support and click **Next**.

   The installation utility installs add-on files and commits them to the e*Gate Registry. This process generally takes no more than a few minutes. On a Windows system, the installation program may open a series of DOS windows. If these windows open, close them, and the installation program proceeds normally.

13 Follow the on-screen prompts to complete the installation.

14 After the installation is complete, exit the install utility.

Java XML Toolkit Overview

This chapter provides an overview of the e*Gate Java XML Toolkit, its components, and Java mapping.

3.1 Java XML Toolkit Description

The Java XML Toolkit components are briefly described in this section. The Java version offers the utility and advantages of the Java programming language. Detailed information on each of the components can be found in their respective chapters in this guide.

3.1.1 DTD Builder

In conjunction with e*Gate, the DTD Builder creates Java Event Type Definition (ETD), files from existing DTD (.**dtd**) files. The resulting ETD file has the extension **.xsc**, is fully compatible with the e*Gate system, and can then be used to parse XML documents.

Access the DTD Builder via the Java ETD Editor in the e*Gate Enterprise Manager graphical user interface (GUI). This ETD Editor provides a convenient Wizard GUI to facilitate the conversion of the original DTD to Java. The DTD Builder maps the individual XML data elements in the original file from XML to corresponding Java elements in the new ETD file.

3.1.2 XML Schema Builder

The XML Schema Builder creates Java Event Type Definition (ETD), files from existing XML Schema (.**xsd**) files. The resulting e*Gate-compatible ETD file also has the extension **.xsc** and can then be used with XML documents, in the same way as files created by the DTD Builder.

Access and use the XML Schema Builder in the Enterprise Manager in same way as you do the DTD Builder. The XML Schema Builder also maps XML data from the original file to Java in the resulting file.

Note: *If there are any changes required to a DTD or XML Schema file, a new, corresponding ETD file must be re-generated to match the changes in the XML input.*

3.2   Java Mapping

When the DTD and XML Schema Builders convert XML **.dtd** and **.xsd** files to Java **.xsc** ETD files, they must map XML data types to Java data types. This section explains the following types of mapping common to both Builders:

- Identifier mapping
- Property mapping

For a detailed explanation of how each Builder maps additional data, see the respective chapter for that Builder.

3.2.1   Identifier Mapping

XML element tags and attribute names are mapped to Java identifiers. Since XML element tags and attribute names can contain characters that are not legal in Java identifiers or are in conflict with Java keywords, a mapping rule is necessary to resolve these cases.

This rule operates as follows:

- Any sequence of the characters ":", or "-", or "_" in the input are omitted.
- The next character following such a sequence, if any, is converted to uppercase.
- If the resulting identifier conflicts with a Java keyword, an underscore is appended.

3.2.2   Property Mapping

XML DTD element declarations that contain attribute declarations or content model particles other than simple (#PCDATA) are mapped to Java classes. Attribute declarations, and embedded content model particles are mapped to "properties" of the generated Java class.

These "properties" are an extension of the Java "bean" properties defined in the Java Beans standard as follows:

- Assuming a property named "x" with a type "T", if the property "x" is mandatory, the following getter/setter methods are generated:

```
T getX();
void setX(T value);
```

- If the property "x" is optional, the following additional methods are generated to test for the existence of and control the presence of "x":

```
boolean hasX();
void omitX();
```

- If the property "x" is repeating, the following methods are generated:

```
T[] getX();
```

Returns all of the "x" elements as an array.

```
void setX(T[] values);
```

Replaces the current "x" elements with values.

```
int countX();
```

Returns the current number of "x" elements.

```
T getX(int index);
```

Returns the "x" element at index.

```
void setX(int index, T value);
```

Replaces the "x" element at index with value.

```
void removeX(int index);
```

Removes the "x" element at index, moving all succeeding elements down.

```
void addX(T val);
```

Adds an "x" element at the end.

```
void addX(int index, T val);
```

Inserts an "x" element at the specified index, moving all succeeding elements.

```
void clearX();
```

Removes all "x" elements.

Java DTD Builder

This chapter explains the Java DTD Builder, including how to use it, a description of data mapping to Java, XML namespaces, and the Builder's capabilities

4.1 Using the DTD Builder

Access the Java DTD Builder using the Java Event Type Definition (ETD) Editor of the e*Gate Enterprise Manager graphical user interface (GUI).

Note: *For more information on how to use the Enterprise Manager, see the **e*Gate Integrator User's Guide**.*

To access the DTD Builder

1 Run the Enterprise Manager, log in, and select the desired schema.

The Enterprise Manager's Main window opens.

2 Be sure that the Java editors are selected as your default editors. To check this default, click on the Options menu and choose the **Default Editor** command.

3 To open the ETD Editor GUI, click the **ETD Editor** button on the Toolbar.

The ETD Editor Main window opens.

4 Click the **New** button on the Toolbar or click the File menu and choose the **New** command.

The New Event Type Definition dialog box opens.

5 Click the **DTD Wizard** icon then click **OK**.

These actions open the DTD Wizard's first window and begin the DTD Builder's operation.

To run the DTD Builder

1 Read the instructions in the DTD Wizard — Introduction (see Figure 1).

Figure 1 DTD Wizard — Introduction

Note: *Be sure to read all the Wizard's instructions carefully and follow the prompts that it gives you, for best results.*

2 Click **Next**.

The DTD Wizard — Step 1 appears (see Figure 2).

Figure 2   DTD Wizard — Step 1
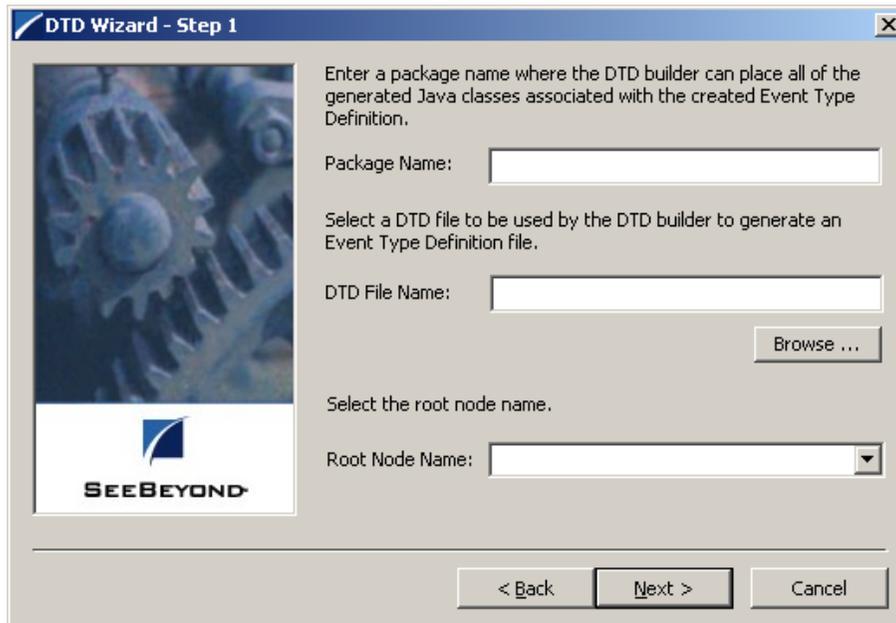
3   Enter the following information:

Caution: *When you are prompted for a package name or root name, be sure to supply a string composed of single-byte characters only. A compile error occurs if you try to compile an ETD whose root name contains Japanese, Korean, Chinese, or other multibyte characters. If the wizard picks up a double-byte string from the input source file and uses it for a root node name, you will need to alter the source and re-run the wizard.*

- **Java Package Name:** Type in the name you want to give the Java package, for example, **com.tes**. This name must conform to Java package name requirements. See the appropriate Java documentation for details.

- **DTD File Name:** Type in the name of the DTD file you want to convert. Click **Browse** to access an Open (file selection) dialog box, allowing you to choose the desired file.

- **Root Node Name:** This text box is a pull-down menu. Select the desired root node name from the menu. For more information on root nodes and ETDs, see the *e*Gate Integrator User's Guide.*

- **Allow whitespace in EMPTY segments:** Check the box to allow any whitespace characters (such as space, tab, or new line) in elements defined as EMPTY. For example, if this box is checked, the following elements are allowed:

  ```
  <chapter>  </chapter>
  ```

  Normally, no whitespace characters would be allowed in EMPTY elements.

- **Ignore #FIXED attributes:** Check the box to strip all #FIXED attributes from the DTD.

- **Ignore all attributes:** Check the box to strip *all* attributes from the DTD.

  ◆ **Include DOCType Reference**: Check the box to specify a **DOCType Reference**. For example, a **DOCType Reference** of **PUBLIC "file_name.dtd"** will generate the following value in the XML output:

```
<!DOCTYPE Result PUBLIC "file_name.dtd">
```

4   When you are finished, click **Next**.

The DTD Wizard — Step 2 appears (see Figure 3).

Figure 3   DTD Wizard — Step 2
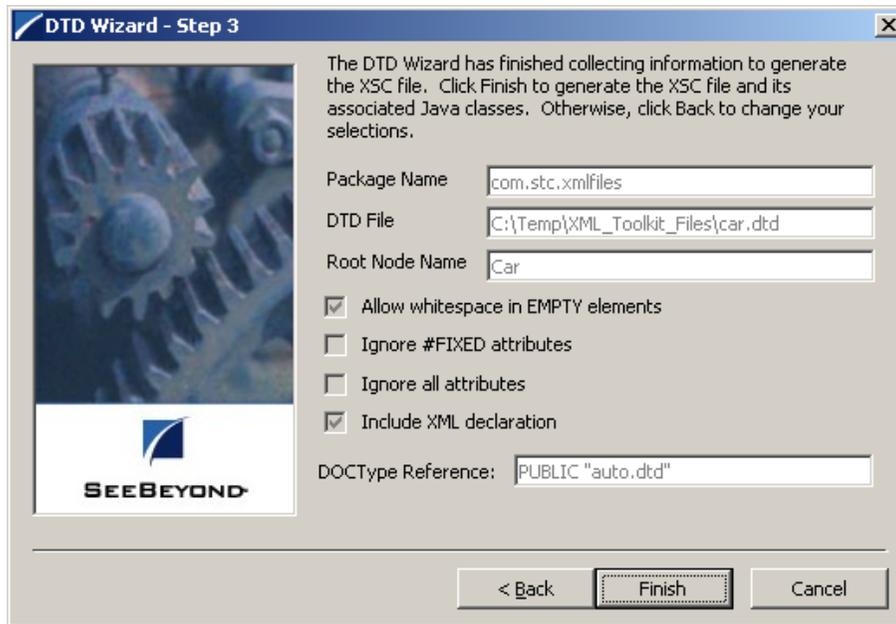
5   Review the information you have entered in the Wizard. If it is correct, click **Finish** to generate a Java ETD (.**xsc** file) from the original DTD file.

The Wizard closes, and the new ETD appears in the ETD Editor Main window. See the *e*Gate Integrator User's Guide* for details on how to use this editor, including an explanation of the information it shows.

6   To save the new ETD, click the **Save** button on the Toolbar or click the File menu and choose the **Save** command.

A Save dialog box appears.

7   Select the desired directory location, give the new ETD your desired name, and click **Save**.

The ETD Editor saves the new Java ETD.

8   You can continue to use the ETD Editor or click the File menu and choose the **Close** command to exit the GUI.

Note: *The ETD nodes created using the DTD Builder appear shaded in the ETD Editor, indicating that you cannot edit an ETD created by the Builder.*

4.2 DTD Builder Data Mapping

As described earlier, when the DTD Builder converts XML **.dtd** files to Java **.xsc** ETD files, the Builder must map XML data to Java data. This process uses the following types of mapping:

- Identifier mapping

- Property mapping

- Mapping of element declarations

- Mapping of attribute types

See **"Java Mapping" on page 21** for an explanation of how the DTD Builder maps identifiers and properties. This section explains how the DTD Builder maps each of the rest of the categories.

4.2.1 Mapping of Element Declarations

For each element declaration in the input DTD, a Java class with the same name is created. This process takes into account the identifier-mapping rule explained under **"Identifier Mapping" on page 21**.

The content of the class depends on the structure of the element declaration as follows:

- If the element contains a content model, the top-level particles of the content model are mapped as properties of the generated class.

- Each of the element's attribute declarations is mapped to a property in the generated class.

The following conditions also apply:

- If an element has a content model that is only character data (#PCDATA) and contains no attribute declarations, no class is generated. Instead, references to such an element in Java treat it as an instance of *java.lang.String*.

- If an element can only contain character data, but does have one or more attributes, a class is generated to represent it, and a special property called *$text* is generated to hold the element's character data.

- If an element's content model contains parenthesized sub expressions, each sub expression is mapped to a static inner class named *$<index>* where *<index>* is the 0-based index of the sub expression counting from the left of the element's content model, for example:

```
<!ELEMENT a  (b,(c,d)?)>
```

The previous sub expression would produce the following (simplified) Java class definition:

```
public class a {
      public b getB() {…}
      public void setB(b value) {…}
      static public class $1 {
              public c getC() {…}
              public void setC(c value) {…}
              public d getD() {…}
              public void setD(d value) {…}
      }
      public $1 get$1() {…}
      public void set$1($1 value) {…}
      public boolean has$1() {…}
      public void omit$1() {…}
}
```

4.2.2 Mapping of Attribute Types

XML attributes of type *CDATA, ID, IDREF, NMTOKEN, IDREFS*, and *NMTOKENS*, including enumerated types, are mapped to the Java type *java.lang.String*.

4.3 Builder Capabilities

This section describes the supported and unsupported DTD features of the DTD Builder.

4.3.1 Supported Features

Basically, the DTD Builder supports every DTD feature specified in the XML 1.0 standard found at the Web site given under **"Introduction to XML" on page 16**.

Namespaces

In addition, the DTD Builder supports namespaces even though namespace processing is not a part of a DTD itself.

The DTD Builder handles namespaces in the following ways:

- By processing instructions. For example:

```
<?xml:namespace ns="http://www.stc.com/ns1" prefix="stc"?>
<!ELEMENT stc:Book (stc:Chapter)+>
<!ELEMENT stc:Chapter (#PCDATA)>
<!ATTLIST stc:Chapter chapNo CDATA #REQUIRED>
```

The namespace processing instruction is the line starting with *<?xml:namespace*. A sample XML document is:

```
<sbyn:Book xmlns:sbyn="http://www.stc.com/ns1">
  <sbyn:Chapter chapNo="1">This is chapter 1</sbyn:Chapter>
</sbyn:Book>
```

Note: *The namespace is identified by its unique uniform resource identifier (URI)* **http://www/stc.com/ns1**. *Notice that, in the actual XML document, the original* **stc** *prefix in the DTD is replaced by* **sbyn**. *This is correct.*

- By special attributes beginning with *xmlns*. For example:

```
<!ELEMENT Book (Chapter)+>
<!ATTLIST Book xmlns CDATA #FIXED "http://www.stc.com/namespace">
<!ELEMENT Chapter (#PCDATA)>
<!ATTLIST Chapter chapNo CDATA #REQUIRED>
```

This example states that the URI for the default namespace (no prefix) is **http://www.stc.com/namespace**. The document instances must contain the default namespace specification. For example:

```
<Book xmlns="http://www.stc.com/namespace">
  <Chapter chapNo="1">This is chapter 1</Chapter>
</Book>
```

If your document is:

```
<Book>
  <Chapter chapNo="1">This is chapter 1</Chapter>
</Book>
```

You get an error because the Builder is expecting a namespace declaration.

Note: *You are not allowed to specify/override the reserved namespace prefix* **xml** *in the DTD.*

4.3.2 Limitations

This section explains the limitations of the DTD Builder.

Parent Nodes

The DTD Builder assumes that a DTD has at least one *ELEMENT* to generate a parent node in the resulting ETD, which has either child nodes or an attribute list. If the DTD contains no *ELEMENT* to generate child nodes or attribute lists, the **.xsc** (as well as **.jar**) files are not created.

For example, see the following DTD:

```
<!ELEMENT a (#PCDATA)>
<!ELEMENT b (#PCDATA)>
```

The DTD in the previous example cannot generate an **.xsc** file because *a* and *b* map to simple strings (*java.lang.String*), and no Java classes are generated for them. If you add another element that includes *a* and/or *b*, you have the following structure:

```
<!ELEMENT c (a,b)>
```

In this case, the DTD Builder can generate a Java ETD (**.xsc**) file.

Note: *For more information on node structures in Java ETDs, see the* **e*Gate Integrator User's Guide**.

Root Nodes

The DTD Builder only generates one ETD (**.xsc** file) from an input DTD, so there can only be one root *ELEMENT* in the DTD. ETDs can only have one root node. As a result, *ELEMENT* items in the DTD that are not referenced in the root *ELEMENT* tree are discarded. See the following example:

```
<!ELEMENT el1 (a,b)>
<!ELEMENT el2 (b+,c)>
<!ELEMENT a (#PCDATA)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (a|d)>
<!ELEMENT d (#PCDATA)>
```

In the previous example, if you select *el1* as the root *ELEMENT*, *el2*, *c* and *d* are discarded and do not show up in the resulting ETD, because *el1* only references *a* and *b*. However, if *el2* is selected as the root *ELEMENT*, *el1* and *b* are discarded. Because *el2* references *b* and *c*, and *c* references *a* and *d,* so *el2, b, c, a,* and *d* are all included in the resulting ETD.

Ignoring Empty Strings in PCDATA Elements

By default, the DTD Compiler does not ignore empty strings in PCDATA elements. To allow this behavior, you must set the **dtd.allowEmpty** value in your **.jcsrc** file to **true**.

To configure the .jcsrc file

1 Navigate to your **C:\Documents and Settings*<user>*** directory, where <user> is the user name you will use while compiling the DTDs.

2 Open the **.jcsrc** file with a text editor (or create one if one does not already exist).

3 Add the following entry to the file:

```
dtd.allowEmpty=true
```

4 Save the file.

Empty Element Tags

Although not necessarily a limitation with the DTD Builder, the XML parser does not currently support empty element tags. Elements using the shorthand notation with only an end tag cannot be parsed correctly.

For example, instead of using an empty element tag to show a city name:

```
</City>
```

The XML parser expects to see the element shown with a start and an end tag:

```
<City>data</City>
```

Java XML Schema Builder

This chapter explains the e*Gate Java XML Schema Builder, including how to use it, a description of data mapping to Java, XML namespaces, and the Builder's capabilities

5.1 Using the XML Schema Builder

The Java Event Type Definition (ETD) Editor's XML Schema Builder allows you to create an ETD file (with the extension **.xsc**) based on an input XML Schema file (**.xsd** file). The Builder's XSD Wizard allows you to build the ETD automatically. Access the XML Schema Builder using this ETD Editor in the e*Gate Enterprise Manager graphical user interface (GUI).

Note: *For more information on how to use the Enterprise Manager, see the **e*Gate Integrator User's Guide**.*

To access the XML Schema Builder

1   Run the Enterprise Manager, log in, and select the desired schema.

    The Enterprise Manager's Main window opens.

2   Be sure that the Java editors are selected as your default editors. To check this default, click on the Options menu and choose the **Default Editor** command.

3   To open the ETD Editor GUI, click the **ETD Editor** button on the Toolbar.

4   Click the **New** button on the Toolbar or click the File menu and choose the **New** command.

    The New Event Type Definition dialog box opens.

5   Click the **XSD Wizard** icon then click **OK**.

    These actions open the XSD Wizard's first window and begin the XML Schema Builder's operation.

To run the XML Schema Builder

1 Read the instructions in the XSD Wizard — Introduction (see Figure 4).

Figure 4 XSD Wizard — Introduction

Note: *Be sure to read all the Wizard's instructions carefully and follow the prompts that it gives you, for best results.*

2 Click **Next**.

The XSD Wizard — Step 1 appears (see Figure 5).

Figure 5 XSD Wizard — Step 1
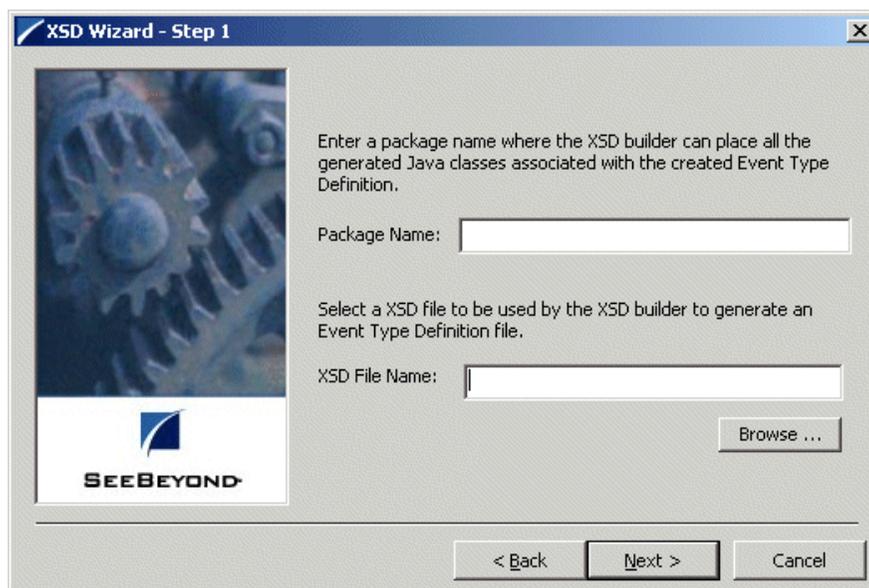
Enter the following information:

◆ **Java Package Name:** Type in the name you want to give the Java package, for example, **com.tes**. This name must conform to Java package name requirements. See the appropriate Java documentation for details.

◆ **XSD File Name:** Type in the name of the **.xsd** file you want to convert. Click **Browse** to access an Open (file selection) dialog box, allowing you to choose the desired file.

3   When you are finished, click **Next**.

The XSD Wizard — Step 2 appears (see Figure 6).

Figure 6 XSD Wizard — Step 2

4   Review the information you have entered in the Wizard. If it is correct, click **Finish** to generate a Java ETD (**.xsc** file) from the original XML Schema file. The following processes happen:

◆ The wizard parses the input XML Schema and places its contents into the new ETD.

◆ The Wizard closes, and the new ETD appears in the ETD Editor Main window. See the *e*Gate Integrator User's Guide* for details on how to use this editor, including an explanation of the information it shows.

5   To save the new ETD, click the **Save** button on the Toolbar or click the File menu and choose the **Save** command.

A Save dialog box appears.

6   Select the desired directory location, give the new ETD your desired name, and click **Save**.

The ETD Editor saves the new Java ETD.

7   You can continue to use the ETD Editor or click the File menu and choose the **Close** command to exit the GUI.

Note: *The ETD nodes created using the XML Schema Builder appear shaded in the ETD Editor, indicating that you cannot edit an ETD created by the Builder.*

After you name and save the ETD, you can then use it as the basis for new or existing e*Gate Event Types in this schema. The original XML Schema (**.xsd** file) is saved in the Java archive (as a **.jar** file) corresponding to the Event Type's ETD (**.xsc** file).

5.2   XML Schema Versions: Java

The Java XML Toolkit supports the World Wide Web Consortium (W3C) October 2000 and May 2001 XML Schema standards.

The following URLs authoritatively identify (via XML namespaces) the two versions of XML Schema that supported in the Java XML Toolkit:

http://www.w3.org/2000/10/XMLSchema

http://www.w3.org/2001/XMLSchema

Note: *These identifiers are XML namespace identifiers and do not identify actual Web sites.*

The XML Schema Builder checks and enforces that the input schemas conform to one of the two W3C versions listed previously. The latter (2001/5) identifier corresponds to the approved W3C recommendations defined by the following documents:

http://www.w3.org/TR/2001/REC-xmlschema-1-20010502

http://www.w3.org/TR/2001/REC-xmlschema-2-20010502

The former identifier (2000/10) corresponds to a widely used draft version that the Builder also supports (due to that wide use), which is defined by the following documents:

http://www.w3.org/TR/2000/CR-xmlschema-1-20001024

http://www.w3.org/TR/2000/CR-xmlschema-2-20001024

5.3 XML Schema Builder Data Mapping

As described earlier, when the XML Schema Builder converts XML **.xsd** files to Java **.xsc** ETD files, the Builder must map XML Schema data types to Java classes. This process uses the following types of mapping:

- Identifier mapping
- Property mapping
- Generated classes
- Java packages
- Extended Java Bean properties
- Mapping of *complexType* data types
- Mapping of *simpleType* data types (W3C 10/2000 and 2001 specifications)
- Mapping of elements

See **"Java Mapping" on page 21** for an explanation of how the XML Schema Builder maps identifiers and properties. This section explains how the Builder maps each of the rest of the types.

5.3.1 Generated Classes

The XML Schema Builder maps top-level elements that have complex types and top-level *complexType* classes to Java classes that contain extended Java Bean properties. This process allows the Builder to access Java Bean attributes and element content. In addition, the Builder generates marshalling and unmarshalling code to serialize and deserialize instances of the generated element classes to or from XML documents.

5.3.2 Java Packages

When you run the XML Schema Builder, you must select a target Java package to contain the generated classes. Since elements and *complexType* classes may have the same names, the generated classes are created in two sub packages of the target package, that is, *elements* and *types* respectively.

5.3.3 Mapping of complexType Data Types

For each top-level *complexType* data type defined in the input schema a corresponding Java class is created under the *types* package. For each attribute defined in the *complexType* data type, an extended Java bean property is generated.

If the *complexType* data type contains an optional or repeating content model group (*sequence, choice, all*) a static inner class is generated to represent the model group. Inner classes that represent content model groups are named *$<index>* where *<index>* is a 1-based integer index corresponding to the occurrence of the content model group, counting from top to bottom.

Examples

```
<complexType name = "SomeType">
  <sequence minOccurs="0" maxOccurs="unbounded">        $1
    <element name="a" type= "string">
    <choice>                                            $2
      <element name="b" type="string">
      <element name="c" type="string">
    </choice>
    <element name="d" type="string">
    <sequence minOccurs="1" maxOccurs="unbounded">      $3
      <element name="e" type="string">
      <element name="f" type="string">
    </sequence>
  </sequence>
</complexType>
```

This convention corresponds to that used in the Perl and JavaScript programming languages for back-references to parenthesized sub expressions of a corresponding regular expression, as in the previous example:

(a(b | c)d(ef)+)*

The following conditions apply:

- If the content model particle is mandatory, no inner class is generated, and the particles contained in the *<all>, <choice>,* or *<sequence>* are generated as immediate members of the *complexType* data type.

- For each element or model group particle contained in the *complexType* data type content model, an extended Java Bean property is generated to access that particle.

- The *complexType* data types that extend or restrict other *complexType* data types are mapped to Java classes that extend the base *complexType* class. The *complexType* data types that extend *simpleType* data types are mapped to Java classes that contain a member named "base" that holds the character data of the base *simpleType* class.

5.3.4 Mapping of simpleType Data Types (W3C 2001 Specifications)

This section explains the mapping of XML Schema *simpleType* data types, which describe the syntax and semantics of attribute values and character data content in XML documents.

The standard Java mapping in this section is defined by the W3C year 2001 specifications for XML Schemas (see **"XML Schema Versions: Java" on page 34**).

Note: *An alternative mapping is also supported for the October 2000 version of XML Schemas. The differences in that mapping compared to these specifications are explained under* **"Mapping of simpleType Data Types (W3C 2001 Specifications)" on page 36***.*

Standard Java Mapping

A list explaining how the *simpleType* data types are mapped to Java classes, according to the W3C year 2001 specifications for XML Schemas, follows:

Any SimpleType

A completely unconstrained *simpleType* is mapped to the Java type *java.lang.String*.

Boolean

The Boolean data type is mapped to the Java primitive type *boolean*.

Base64Binary and hexBinary

These data types represent binary data. They are mapped to the Java type *byte[]*.

Float

The *float* data type is mapped to the Java primitive type *float*.

Double

The *double* data type is mapped to the Java primitive type *double*.

AnyURI

The *anyURI* data type is mapped to the Java type *java.lang.String*.

QName

The *QName* data type is mapped to the Java type *java.lang.String*.

NOTATION

The *NOTATION* data type is mapped to the Java type *java.lang.String*.

String

The *string* data type and all types derived from it, namely *token, language, Name, NMTOKEN, NCName, ID, IDREF*, and *ENTITY* are mapped to the Java type *java.lang.String*.

Union

All instances of the *Union* data-type constructor are mapped to the Java type *java.lang.String*.

List

All instances of the *list* data-type constructor are mapped to a repeating Java Bean property with its *itemType* facet mapped according to the rules given under **"Mapping of simpleType Data Types (W3C 2001 Specifications)" on page 36**.

Numeric Types

The XML Schema numeric data types are mapped to one of the following Java numeric types: *byte, short, int, long, java.math.BigInteger*, or j*ava.math.BigDecimal*. The Java type is selected according to the facets of the XML Schema type. The mapping chooses the smallest Java numeric type that can represent the XML Schema type according to its facets.

The following list shows the mapping for unconstrained built-in XML Schema numeric data types:

- *decimal* is mapped to the Java type *java.math.BigDecimal*.

- *integer, nonNegativeInteger, nonPositiveInteger, negativeInteger, positiveInteger*, and *unsignedLong* are mapped to the Java type *java.math.BigInteger*.

- *long* and *unsignedInt* are mapped to the Java type *Long*.

- *int* and *unsignedShort* are mapped to the Java type *Int*.

- *short* and *unsignedByte* data types are mapped to the Java type *Short*.

- *byte* is mapped to the Java type *Byte*.

Additional Java Mapping

The following list explains additional Java mapping the Builder generates:

- *Duration* is mapped to a custom class *com.stc.jcsre.xml.xsd.datatypes.Duration*.
  Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class Duration {
    public Duration();
    public Duration(Duration copy);
    public void setYears(int years);
    public void setMonths(int months);
    public void setDays(int days);
    public void setHours(int hours);
    public void setMinutes(int minutes);
    public void setSeconds(int seconds);
    public void setMilliseconds(int millis);
    public int getYears();
    public int getMonths();
    public int getDays();
    public int getHours();
    public int getMinutes();
    public int getSeconds();
    public int getMilliseconds();
    public boolean isNegative();
    public void setNegative(boolean value);
    public String toString();
    public static Duration parse(String value);
}
```

- *dateTime* is mapped to a custom Java class *com.stc.jcsre.xml.xsd.datatypes.DateTime*. Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;

public class DateTime {
    public DateTime() {
    public DateTime(java.util.GregorianCalendar cal)
    public DateTime(int year, int month, int day)
    public DateTime(int year,
            int month,
            int day,
            int hour,
            int minute,
            int second,
            int millisecond);

    public boolean after(DateTime other);
    public boolean before(DateTime other);
    public void setCalendar(GregorianCalendar cal);
    void roll(Duration dur);
    void add(Duration dur);
    void subtract(Duration dur);
    public GregorianCalendar toCalendar();
```

```
        public int getYear();
        public int getMonth();
        public int getDay();
        public int getHours(int hours);
        public int getMinutes();
        public int getSeconds();
        public int getMilliseconds();
        public void setYear(int year);
        public void setMonth(int month);
        public void setDay(int day);
        public void setHours(int hours);
        public void setMinutes(int minutes);
        public void setSeconds(int seconds);
        public void setMilliseconds(int millis);
        public void rollYear(int years);
        public void rollMonth(int months);
        public void rollDay(int days);
        public void rollHours(int hours);
        public void rollMinutes(int minutes);
        public void rollSeconds(int seconds);
        public void rollMilliseconds(int millis);
        public void addYear(int years);
        public void addMonth(int months);
        public void addDay(int days);
        public void addHours(int hours);
        public void addMinutes(int minutes);
        public void addSeconds(int seconds);
        public void addMilliseconds(int millis);
        public String toString();
        public static DateTime parse(char[] value);
        public static DateTime parse(String value)
}
```

▪ *time* is mapped to a custom Java class *com.stc.jcsre.xml.xsd.datatypes.Time*.
Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class Time {
        public Time() ;
        public Time(int hours, int minutes,
            int seconds, int milliseconds, java.util.TimeZone tz);
        public int getHours();
        public int getMinutes();
        public int getSeconds();
        public int getMilliseconds();
        public void setHours(int hours);
        public void setMinutes(int minutes);
        public void setSeconds(int seconds);
        public void setMilliseconds(int millis);
        public TimeZone getTimeZone();
        public void setTimeZone(java.util.TimeZone tz);
        public String toString();
        static public Time parse(String value);
}
```

▪ *date* is mapped to a custom Java class *com.stc.jcsre.xml.xsd.datatypes.Date*.
Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class Date {
        public Date();
        public Date(int year, int month, int day, java.util.TimeZone tz);
        public int getYear();
        public void setYear(int value);
        public int getMonth(); // January == 1
```

```
        public void setMonth(int value);
        public int getDay();
        public void setDay(int value);
        public java.util.TimeZone getTimeZone();
        public void setTimeZone(java.util.TimeZone tz);
        public String toString();
        static public Date parse(String value);
}
```

▪ *gYearMonth* is mapped to a custom Java class
  com.stc.jcsre.xml.xsd.datatypes.gYearMonth. Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class GYearMonth {
        public GYearMonth();
        public GYearMonth(int year, int month, java.util.TimeZone tz);
        public int getYear();
        public void setYear(int value);
        public int getMonth(); // January == 1
        public void setMonth(int value);
        public java.util.TimeZone getTimeZone();
        public void setTimeZone(java.util.TimeZone tz);
        public String toString();
        public static GYearMonth parse(String value);
}
```

▪ *gYear* is mapped to a custom Java class *com.stc.jcsre.xml.xsd.datatypes.GYear*.
  Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class GYear {
        public GYear();
        public GYear(int year, java.util.TimeZone tz);
        public int getYear();
        public void setYear(int value);
        public java.util.TimeZone getTimeZone();
        public void setTimeZone(java.util.TimeZone tz);
        public String toString();
        public static GYear parse(String value);
}
```

▪ *gMonthDay* is mapped to a custom Java class
  com.stc.jcsre.xml.xsd.datatypes.gMonthDay. Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class GMonthDay {
        public GMonthDay();
        public GMonthDay(int month, int day, TimeZone tz);
        public int getMonth() { // January == 1
        public void setMonth(int value);
        public int getDay();
        public void setDay(int value);
        public TimeZone getTimeZone();
        public void setTimeZone(java.util.TimeZone tz);
        public String toString();
        public static GMonthDay parse(String value);
}
```

▪ *gDay* is mapped to a custom Java class *com.stc.jcsre.xml.xsd.datatypes.gDay*.
  Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class GDay {
        public GDay();
        public GDay(int day, TimeZone tz);
```

```
        public int getDay();
        public void setDay(int value);
        public java.util.TimeZone getTimeZone();
        public void setTimeZone(java.util.TimeZone tz);
        public String toString();
        public static GDay parse(String value);
    }
```

- ▪ *gMonth* is mapped to a custom Java class *com.stc.jcsre.xml.xsd.datatypes.gMonth*.
  Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class GMonth {
    public GMonth();
    public GMonth(int month, TimeZone tz);
    public int getMonth();
    public void setMonth(int value); // 1 == January
    public java.util.TimeZone getTimeZone();
    public void setTimeZone(java.util.TimeZone tz);
    public String toString();
    public static GMonth parse(String value);
}
```

Mapping of simpleType Data Types (W3C 2000 Specifications)

This section explains the mapping of XML Schema *simpleType* data types. The mapping in this section is defined by the W3C October 2000 specifications for XML Schemas. For more details on these specifications, point your Web browser to:

http://www.w3.org/TR/2000/CR-xmlschema-2-20001024

Standard Java Classes

A list explaining how the *simpleType* data types are mapped to Java (W3C, 10/2000) follows:

Any SimpleType

A completely unconstrained *simpleType* is mapped to the Java type *java.lang.String*.

Boolean

The *boolean* data type is mapped to the Java primitive type *boolean*.

Binary

The *binary* data type represents binary data and is mapped to the Java type *byte[]*.

Float

The *float* data type is mapped to the Java primitive type *float*.

Double

The *double* data type is mapped to the Java primitive type *double*.

uriReference

The *uriReference* data type is mapped to the Java type *java.lang.String*.

QName

The *QName* data type is mapped to the Java type *java.lang.String*.

NOTATION

The *NOTATION* data type is mapped to the Java type *java.lang.String*.

CDATA

The *CDATA* data type is mapped to the Java type *java.lang.String*.

String

The *string* data type and all types derived from it, namely *token, language, Name, NMTOKEN, NCName, ID, IDREF,* and *ENTITY* are mapped to the Java type *java.lang.String*.

Union

All instances of the *Union* data-type constructor are mapped to the Java type *java.lang.String*.

List

All instances of the *list* data-type constructor are mapped to a repeating Java Bean property with its *itemType* facet mapped according to the rules given under **"Mapping of simpleType Data Types (W3C 2001 Specifications)" on page 36**.

Numeric Types

The XML Schema *numeric* data types are mapped to one of the following Java numeric types: *byte, short, int, long, java.math.BigInteger,* and *java.math.BigDecimal*. The Java type is selected according to the facets of the XML Schema type. The mapping chooses the smallest Java numeric type that can represent the XML Schema type according to its facets.

The following list shows the mapping for unconstrained built-in XML Schema numeric data types:

- *decimal* is mapped to the Java type *java.math.BigDecimal*.

- *integer, nonNegativeInteger, nonPositiveInteger, negativeInteger, positiveInteger,* and *unsignedLong* are mapped to the Java type *java.math.BigInteger*.

- *long* and *unsignedInt* are mapped to the Java type *long*.

- *int* and *unsignedShort* are mapped to the Java type *int*.

- *short* and *unsignedByte* are mapped to the Java type *short*.

- *byte* is mapped to the Java type *byte*.

Additional Java Mapping

The following list explains additional Java mapping the Builder generates:

- *recurringDuration* is mapped to a custom class *com.stc.jcsre.xml.xsd.datatypes.RecurringDuration*. Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class RecurringDuration
    public RecurringDuration();
```

```
        public RecurringDuration(TimeDuration duration, TimeDuration
            period);
        public RecurringDuration(char[] duration, char[] period);
        public void setPeriod(TimeDuration period);
        public void setPeriod(char[] period);
        public void setDuration(TimeDuration duration);
        public void setDuration(char[] duration);
        public void setCentury(int century);
        public void setYear(int year);
        public void setMonth(int month);
        public void setDay(int day);
        public boolean isLeap();
        public void setHour(int hour);
        public void setMinute(int minute);
        public void setSecond(int second, int millsecond, int
            millidigits);
        public void setZone(int hour, int minute);
        public void setNegative();
        public void setZoneNegative();
        public void setUTC();
        public TimeDuration getPeriod();
        public TimeDuration getDuration();
        public int getCentury();
        public int getYear();
        public int getMonth();
        public int getDay();
        public int getHour();
        public int getMinute();
        public int getSeconds();
        public int getMilli();
        public int getMilliDigits();
        public int getZoneHour();
        public int getZoneMinute();
        public boolean isUTC();
        public boolean isNegative();
        public boolean isZoneNegative();
        public static RecurringDuration parse(char[] chars);
        public String toString()
        public TimePeriod()
        public TimePeriod(char[] duration)
        public static TimePeriod parseTimePeriod(char[] value)
    }
```

- *century* is mapped to a custom class *com.stc.jcsre.xml.xsd.datatypes.Century*.
  Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class Century extends TimePeriod {
    public Century();
    public static Century parseCentury(char[] value)
    public String toString();
}
```

- *date* is mapped to a custom class *com.stc.jcsre.xml.xsd.datatypes.Date*. Accessor
  functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class Date extends RecurringDuration {
    public Date()
    public static Date parseDate(char[] value)
    public String toString();
}
```

- *month* is mapped to a custom class *com.stc.jcsre.xml.xsd.datatypes.Month*.
  Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class Month extends TimePeriod {
    public Month();
    public static Month parseMonth(char[] value);
    public String toString();
}
```

- *recurringDate* is mapped to a custom class
  com.stc.jcsre.xml.xsd.datatypes.RecurringDate. Accessor functions for this class
  are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class RecurringDate extends RecurringDuration {
    public RecurringDate();
    public String toString();
    public static RecurringDate parseRecurringDate(char[] value);
}
```

- *recurringDay* is mapped to a custom class
  com.stc.jcsre.xml.xsd.datatypes.RecurringDay. Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class RecurringDay extends RecurringDuration {
    public RecurringDay();
    public String toString();
    public static RecurringDay parseRecurringDay(char[] value);
}
```

- *time* is mapped to a custom class *com.stc.jcsre.xml.xsd.datatypes.Time*. Accessor
  functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class Time extends RecurringDuration {
    public Time();
    public String toString();
    public static Time parseTime(char[] chars);
}
```

- *timeDuration* is mapped to a custom class
  com.stc.jcsre.xml.xsd.datatypes.TimeDuration. Accessor functions for this class are:

```
package com.stc.jcsre.xml.xsd.datatypes;
public class TimeDuration {
    public TimeDuration();
    public void setYear(int year);
    public void setMonth(int month);
    public void setDay(int day);
    public void setHour(int hour);
    public void setMinute(int minute);
    public void setSeconds(int second, int millisecond, int
        millidigits);
    public void setNegative();
    public void setValue(int year, int month, int day,
                         int hour, int minute, int second, int
                             millisecond, int millidigits);
    public int getYear();
    public int getMonth();
    public int getDay();
    public int getHour();
    public int getMinute();
    public int getSeconds();
    public int getMilliseconds();
    public int getMilliDigits();
    public boolean isNegative();
    public String toString();
```

```
                    public static TimeDuration parse(char[] str);
            }
```

- *timeInstant* is mapped to a custom class
  com.stc.jcsre.xml.xsd.datatypes.TimeInstant. Accessor functions for this class are:

```
    package com.stc.jcsre.xml.xsd.datatypes;
    public class TimeInstant extends RecurringDuration {
        public TimeInstant();
        public static TimeInstant parseTimeInstant(char[] value);
    }
```

- *year* is mapped to a custom class *com.stc.jcsre.xml.xsd.datatypes.Year*. Accessor functions for this class are:

```
    package com.stc.jcsre.xml.xsd.datatypes;
    public class Year extends TimePeriod {
        public Year();
        public static Year parseYear(char[] value);
        public String toString();
    }
```

5.3.5 Mapping of Elements

Elements that have complex content are mapped to Java classes. If the element's *type* attribute refers to a top-level *complexType*, the generated class extends the class generated for the referenced *complexType*. If the element contains an in-line *complexType*, the element class body is generated according to the same rules given above for *complexType* and no separate class is generated to represent its type.

Java classes are not generated for elements that have simple content. Instead *complexType* classes that reference such elements contain a Java Bean property to access the element and code to marshal/unmarshal the element as part of the enclosing class.

5.4   Builder Capabilities

This section describes the supported and unsupported XML Schema features of the XML Schema Builder.

5.4.1 Supported Features

The XML Schema Builder supports the following features:

- **Namespaces**: This release of the XML Schema Builder fully supports XML namespaces.

- *Xsi:type*: This release of the XML Schema Builder supports the *xsi:type* feature for dynamic type selection among *complexType* extensions and restrictions.

5.4.2 Unsupported Features

The XML Schema Builder does *not* support the following features:

- **Validation:** This release of the XML Schema Builder is not a full-fledged XML Schema validator. It does basic syntax checking of input schemas but does not rigorously enforce semantic constraints. Furthermore, the generated Java classes do not enforce XML Schema data type constraints beyond those required to map character data to the corresponding Java data types.

- **Wildcards:** This release of the XML Schema Builder does not support the *<any>* or *<anyAttribute>* constructs of XML Schemas.

- **Mixed content:** This release of the XML Schema Builder does not support mixed content.

- **Substitution groups:** This release of the XML Schema Builder does not support substitution groups.

- *Xsi:null*: This release of the XML Schema Builder does not support *xsi:null*.

Java Conversion Examples

This chapter provides before-and-after Java conversion examples of DTDs and XML Schemas, with explanations.

6.1 DTD Examples

This section provides examples of DTDs converted by the e*Gate Java DTD Builder. The first examples show the XML/DTD content before the conversion, and the second shows how the DTD looks after using the e*Gate DTD Builder, in the Event Type Definition (ETD) Editor's Main window. This window shows the node structure of the generated Java ETD (**.xsc**) file.

6.1.1 Book Sample

This section provides examples of a DTD book before and after conversion.

DTD File Before Using the Builder

The following example shows the XML contents of a DTD book file:

```
<!ELEMENT Book (Title, Chapter+)>
<!ATTLIST Book Author CDATA #REQUIRED>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Chapter (#PCDATA)>
<!ATTLIST Chapter id ID #REQUIRED>
```

Converted File in the ETD Editor Window

The following example shows the DTD book file after using the Builder, as it appears in the ETD Editor window.

Figure 7 Book DTD in ETD Editor Window
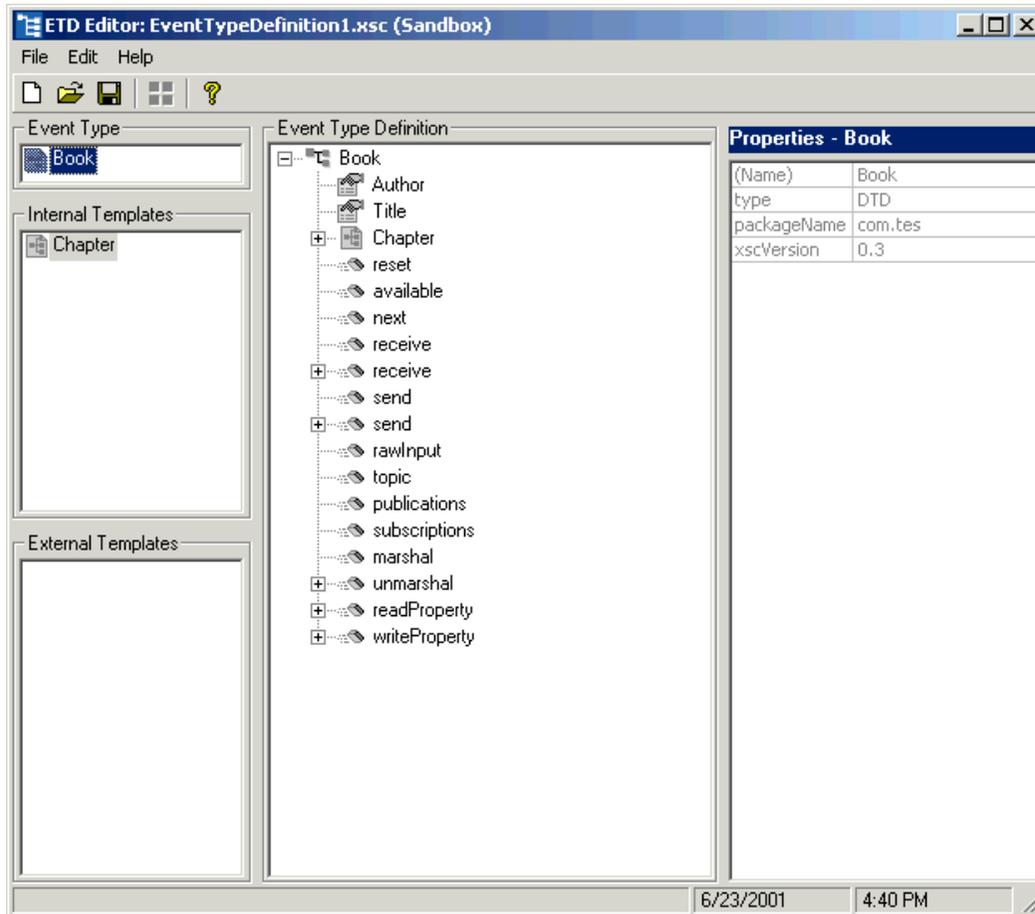
Note: *The Event Type Definition pane in the ETD Editor's Main window shows the node structure of the generated ETD. The operation of this pane is similar to that of the Microsoft Explorer window. Click on the + mark (if present) in front of a node icon to see the child node structure under that parent node. A - mark means there are no child nodes under that node. For details on this pane's operation, see the **e*Gate Integrator User's Guide**.*

6.1.2 Personnel Record Sample

This section provides examples of a DTD personnel record before and after conversion.

DTD File Before Using the Builder

The following example shows the XML contents of a DTD personnel record file:

```
<?xml encoding="UTF-8"?>
<!ELEMENT personnel (person+)>

<!ELEMENT person (name,email*,url*,link?)>
<!ATTLIST person id ID #REQUIRED>
<!ATTLIST person note CDATA #IMPLIED>
<!ATTLIST person contr (true|false) 'false'>
<!ATTLIST person salary CDATA #IMPLIED>

<!ELEMENT name ((family,given)|(given,family))>

<!ELEMENT family (#PCDATA)>

<!ELEMENT given (#PCDATA)>

<!ELEMENT email (#PCDATA)>

<!ELEMENT url EMPTY>
<!ATTLIST url href CDATA 'http://'>

<!ELEMENT link EMPTY>
<!ATTLIST link manager IDREF #IMPLIED>
<!ATTLIST link subordinates IDREFS #IMPLIED>

<!NOTATION gif PUBLIC '-//APP/Photoshop/4.0' 'photoshop.exe'>
```

Converted File in the ETD Editor Window

The following example shows the DTD personnel record file after using the Builder, as it appears in the ETD Editor window.

Figure 8   Personnel Record DTD in ETD Editor Window
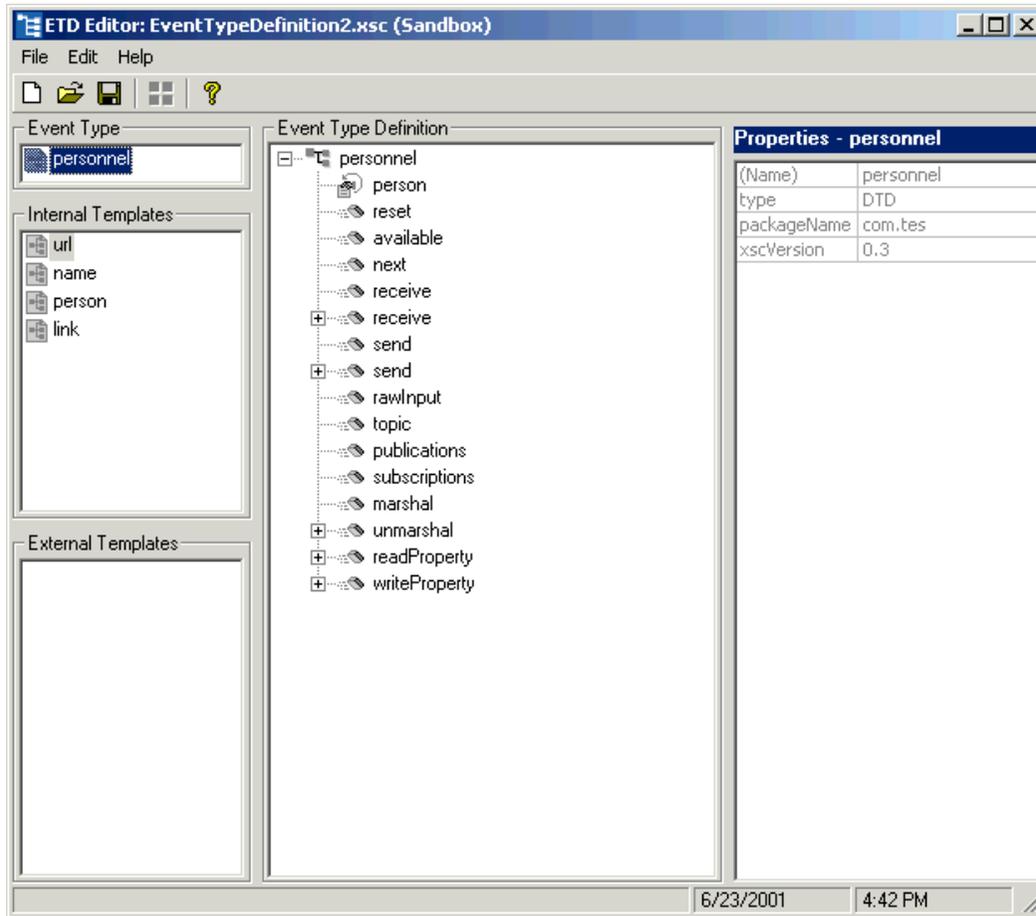
6.1.3 Namespace Sample

This section provides examples of a DTD document before and after conversion.

DTD File Before Using the Builder

The following example shows the XML contents of a DTD namespace file:

```
<?xml:namespace ns="http://ns.name.org" prefix="name"?>
<?xml:namespace ns="http://ns.any.org" prefix="any"?>
<!ELEMENT name:foo (name:bar+)>
<!ATTLIST name:foo xml:lang TOKEN #REQUIRED>
<!ATTLIST name:foo any:foo TOKEN #REQUIRED>
<!ELEMENT name:bar (#PCDATA)>
```

Converted File in the ETD Editor Window

The following example shows the DTD namespace file after using the Builder, as it appears in the ETD Editor window.

Figure 9 Namespace DTD in ETD Editor Window
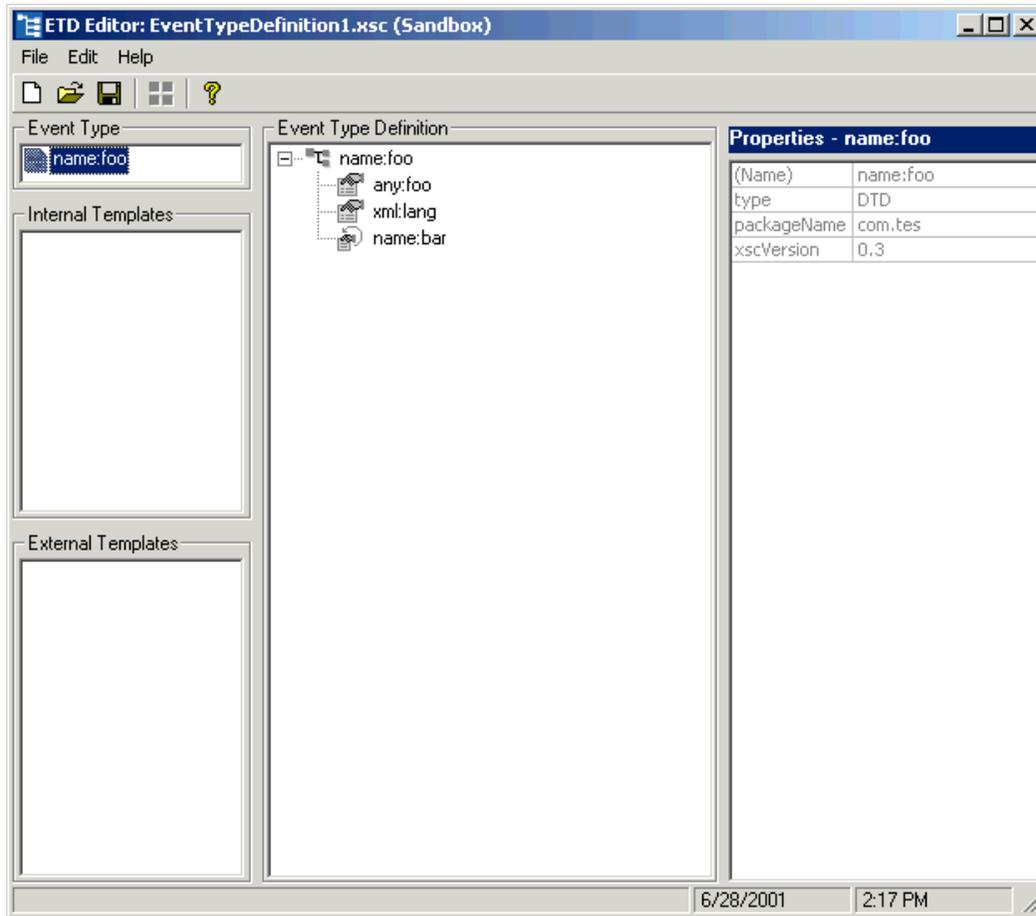
6.1.4 Mixed Sample

This section provides examples of a mixed DTD before and after conversion.

DTD File Before Using the Builder

The following example shows the XML contents of a mixed DTD file:

```
<!ELEMENT foo (#PCDATA|bar)*>
<!ATTLIST foo text CDATA #IMPLIED>
<!ELEMENT bar (a,b)>
<!ELEMENT a (#PCDATA)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT any ANY>
```

Converted File in the ETD Editor Window

The following example shows the mixed DTD file after using the Builder, as it appears in the ETD Editor window.

Figure 10   Mixed DTD in ETD Editor Window
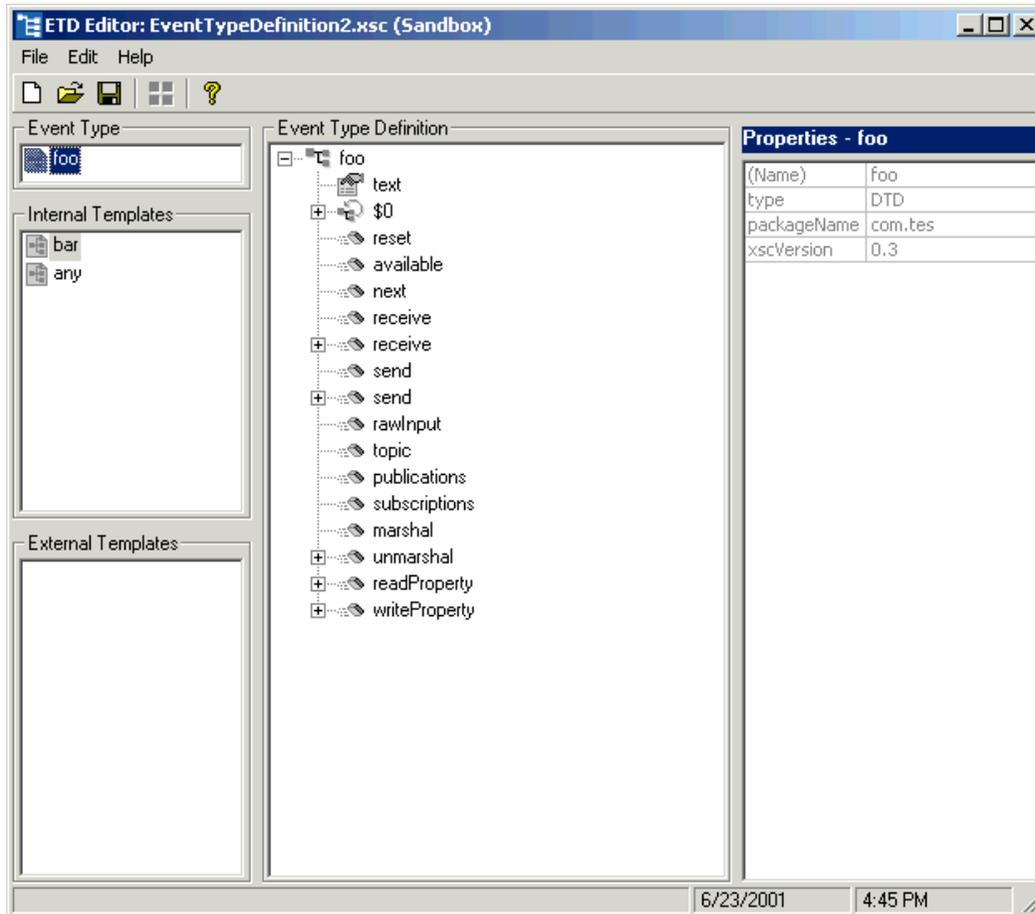
6.1.5 Document Sample

This section provides examples of a DTD document before and after conversion.

DTD File Before Using the Builder

The following example shows the XML contents of a DTD document file:

```
<!ELEMENT doc       (title, (para|listing|indexterm)+)>

<!ELEMENT para
(#PCDATA|emphasis|cite|xref|footnote|indexterm)*>

<!ELEMENT emphasis (#PCDATA|footnote)*>
<!ELEMENT cite      (#PCDATA)>
<!ATTLIST cite
    type(book|article|other)"book"
```

```
>

<!ELEMENT footnote (#PCDATA|para)*>

<!ELEMENT title    (#PCDATA|emphasis)*>

<!ELEMENT listing  (#PCDATA)>
<!ATTLIST listing
    id       ID  #IMPLIED
    colwidthCDATA"80"
>

<!-- startref points to rangestart -->
<!ELEMENT indexterm (prim?, sec?)>
<!ATTLIST indexterm
    id       ID  #IMPLIED
    type     (rangestart|rangeend|singular) "singular"
    startrefIDREF#IMPLIED
>

<!ELEMENT prim     (#PCDATA)>
<!ELEMENT sec      (#PCDATA)>

<!ELEMENT xref     EMPTY>
<!ATTLIST xref
    linkend  IDREF#REQUIRED
>

<!ENTITY ldquo"&#x201C;">
<!ENTITY rdquo"&#x201D;">
```

Converted File in the ETD Editor Window

The following example shows the DTD document file after using the Builder, as it appears in the ETD Editor window.

Figure 11   Document DTD in ETD Editor Window
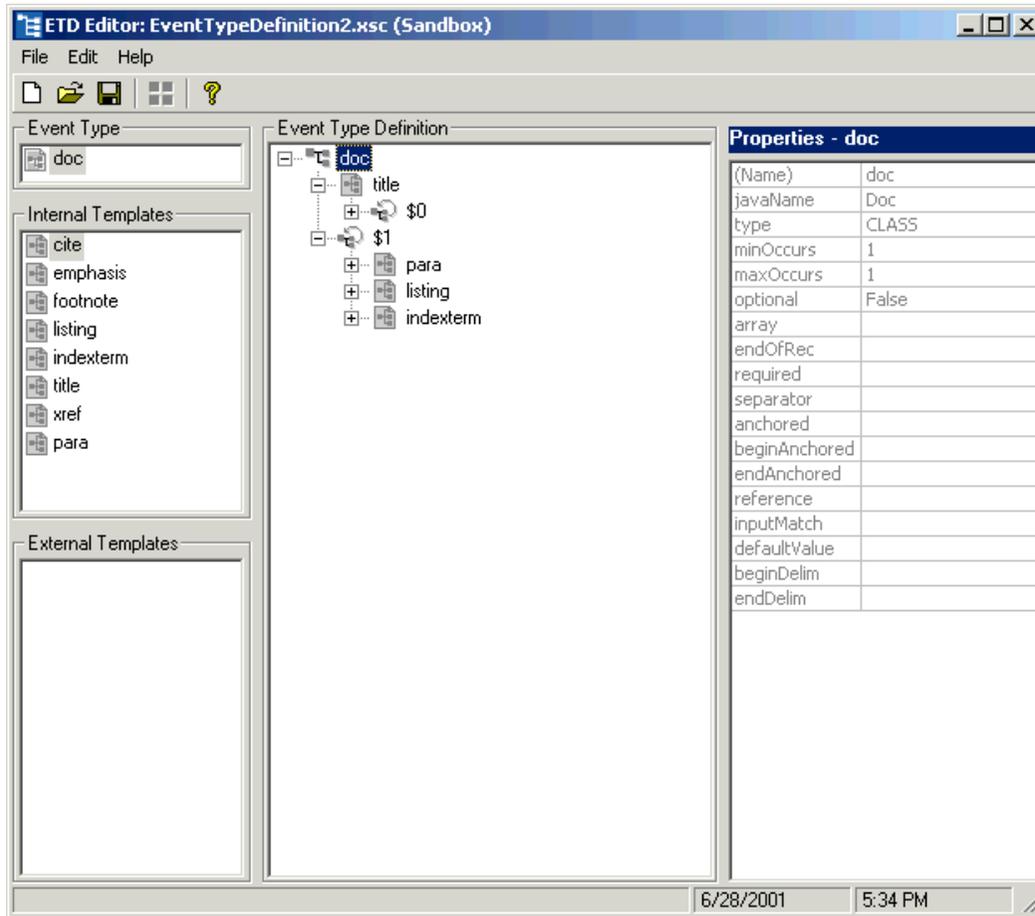
6.2   XML Schema Example

This section provides examples of XML Schemas converted by the e*Gate Java XML Schema Builder. The first examples show the XML Schema content before the conversion, and the second shows how the XML Schema looks after using the e*Gate XML Schema Builder, in the Event Type Definition (ETD) Editor's Main window. This window shows the node structure of the generated Java ETD (**.xsc**) file.

This section provides examples of an XML Schema purchase order file before and after conversion.

6.2.1 XML Schema File Before Using the Builder

The following example shows the contents of an XML Schema purchase order file:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.purchase_order/sample"
xmlns:ipo="http://wwww.purchase_order/sample/schema"
    <annotation>
        <documentation>
    Sample Purchase Order schema
        </documentation>
    </annotation>
    <!-- include address constructs -->
    <include schemaLocation="address.xsd"/>
    <element name="purchaseOrder" type="ipo:PurchaseOrderType"/>
    <element name="comment" type="string"/>
    <complexType name="PurchaseOrderType">
        <sequence>
            <element name="shipTo" type="ipo:Address"/>
            <element name="billTo" type="ipo:Address"/>
            <element ref="ipo:comment" minOccurs="0"/>
            <element name="Items" type="ipo:Items"/>
        </sequence>
        <attribute name="orderDate" type="date"/>
    </complexType>
    <complexType name="Items">
        <sequence>
            <element name="item" minOccurs="0"
maxOccurs="unbounded">
                <complexType>
                    <sequence>
                        <element name="productName"
type="string"/>
                        <element name="quantity">
                            <simpleType>
                                <restriction
base="positiveInteger">
                                    <maxExclusive value="100"/>
                                </restriction>
                            </simpleType>
                        </element>
                        <element name="price" type="decimal"/>
                        <element ref="ipo:comment" minOccurs="0"/
>
                        <element name="shipDate" type="date"
minOccurs="0"/>
                    </sequence>
                    <attribute name="partNum" type="ipo:Sku"/>
                </complexType>
            </element>
        </sequence>
    </complexType>
    <simpleType name="Sku">
        <restriction base="string">
            <pattern value="\d{3}-[A-Z]{2}"/>
        </restriction>
    </simpleType>
</schema>
```

6.2.2 Converted File in the ETD Editor Window

The following example shows the XML Schema purchase order file after using the Builder, as it appears in the ETD Editor window.

Figure 12 XML Schema Purchase Order File in ETD Editor Window
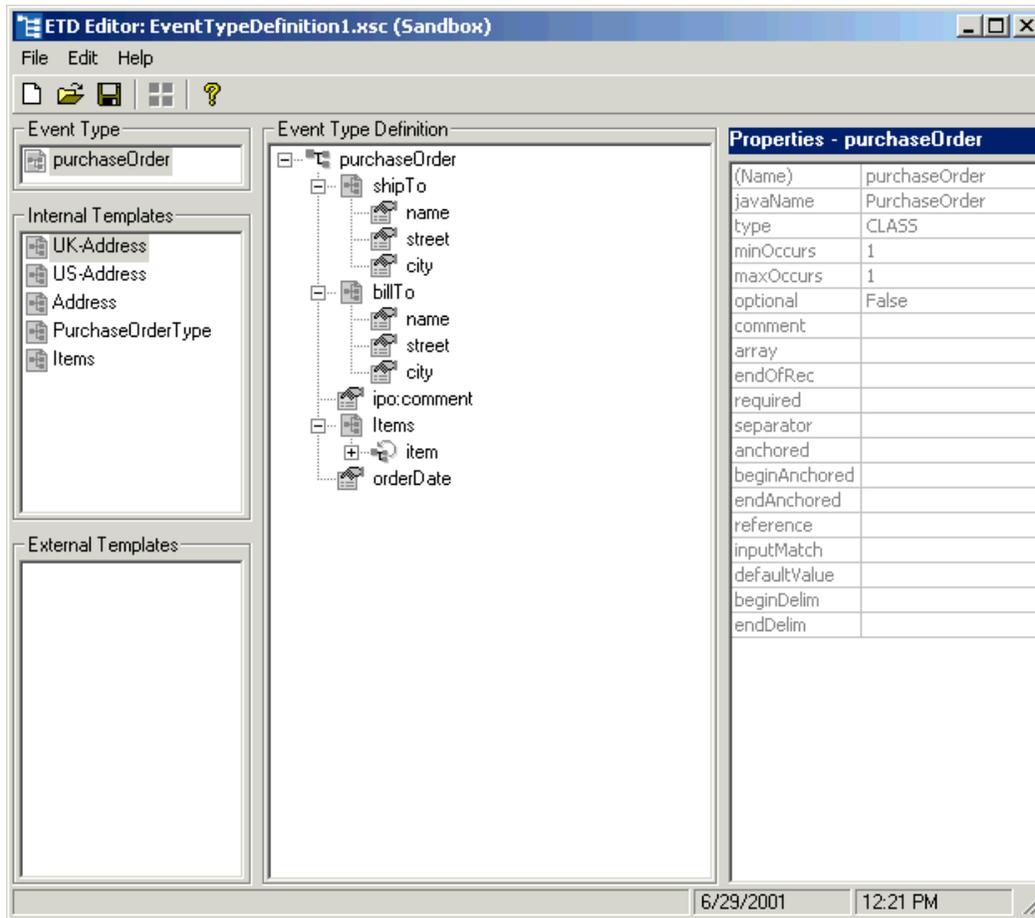
Registry API for XML Schema Metadata

This chapter explains how to use the e*Gate Registry Application Programming Interface (API) for Java XML Schema Metadata.

7.1 Registry API for XML Schemas: Overview

The Registry API for XML Schema Metadata allows you to do the following tasks:

- List the names of all e*Gate schemas in a given Registry
- List the names of all Event Types in a given e*Gate schema
- List the names of all XML Schema files (**.xsd** files) in the **.jar** file associated with a given Event Type
- Retrieve the schema data contained in a given **.xsd** file

Note: *When querying a Registry, the API preferentially returns content in the Sandbox of the current user. It returns Run-time content only when there is no un-promoted Sandbox content for the current user.*

This chapter includes:

- Overview of the package and brief description of each API method
- Sample code implementations

Platforms and Prerequisites

The e*Gate Registry API for XML Schema Metadata is supported on all platforms supported by the current version of e*Gate. The package is installed on top of an existing installation of e*Gate, but it runs outside of e*Gate and makes no use of the e*Gate graphical user interfaces (GUIs).

Note: *You can use the API to list and retrieve information on any connected host, not just the host where you install this package.*

XML Schema Builder

For a complete explanation of how to use the e*Gate ETD Editor XML Schema Builder, see **"Using the XML Schema Builder" on page 31**.

7.2 Package Contents, Setup, and APIs

This section includes the following information:

- Lists the files comprising the package for e*Gate Registry API for XML Schema Metadata

- Provides instructions for system preparation and setup

- Explains how to use the APIs

7.2.1 Contents

The package for e*Gate Registry API for XML Schema Metadata consists of three **.jar** files (see Table 1).

Table 1 Files for This Package

File Name	Comments
stcjcs.jar	Replacement for the **stcjcs.jar** file shipped with core e*Gate
jcscomp.jar	Replacement for the **jcscomp.jar** file shipped with core e*Gate
EgateXMLSchemaRegistry.jar	New file for this package

7.2.2 System Preparation

Before using this package, you must have already installed e*Gate. Before installing this package, back up your existing **stcjcs.jar** and **jcscomp.jar** files. For example:

```
cd \eGate\client\classes
rename stcjcs.jar stcjcs.jar.bak
cd \eGate\Server\registry\repository\default\classes
rename stcjcs.jar stcjcs.jar.bak
cd \eGate\client\bin\java
rename jcscomp.jar jcscomp.jar.bak
cd \eGate\Server\registry\repository\default\bin\java
rename jcscomp.jar jcscomp.jar.bak
```

7.2.3 System Setup

Use the following steps to install the package and set up your environment:

1 Copy the three **.jar** files to your client classes and default repository classes directories. For example, if your CD-ROM drive is F and if e*Gate is installed on your C drive in the **\eGate** directory, enter the following commands:

```
cd \eGate\client\classes
copy F:stcjcs.jar .
copy F:EgateXMLSchemaRegistry.jar .
cd \eGate\Server\registry\repository\default\classes
copy F:stcjcs.jar .
copy F:EgateXMLSchemaRegistry.jar .
cd \eGate\client\bin\java
copy F:jcscomp.jar .
cd \eGate\Server\registry\repository\default\bin\java
copy F:jcscomp.jar .
```

2 Be sure your classpath (%classpath% on Windows, or $CLASSPATH on UNIX) includes the following **.jar** files:

> **C:\eGate\client\classes\stcjcs.jar**
> **C:\eGate\client\classes\egate.jar**
> **C:\eGate\client\classes\swingall.jar**
> **C:\eGate\client\classes\EgateXMLSchemaRegistry.jar**

If your e*Gate installation is located in a path other than **C:\eGate**, such as **/home/*user-name*/eGate/** on UNIX, make the appropriate substitution.

3 Verify that your path (%path% on Windows, or $PATH on UNIX) includes the following directory:

> **C:\jdk1.3.1\bin**

If your installation of Java 2 SDK v1.3 is located in a path other than **C:\jdk1.3.1\bin**, such as **/jdk1.3/bin**, make the appropriate substitution.

Note: *If you run an API in this package and encounter **classnotfound** errors, add your current directory to your classpath.*

7.2.4 Using the APIs

This section lists and describes the APIs in the package for e*Gate Registry API for XML Schema Metadata.

The API consists of two classes that include a total of seven methods. All class files are packaged inside the **EgateXMLSchemaRegistry.jar** file.

For the class **EgateXMLSchemaRegistry**, there are the following APIs:

- **connect()** on page 60
- **listEgateSchemas()** on page 60
- **listEgateEventTypes()** on page 61
- **listXMLSchemaFiles()** on page 61
- **close()** on page 62

For the class **XMLSchemaFileReader**, there are the following APIs:

- **getXMLSchemaData()** on page 62
- **getXMLSchemaFileName()** on page 63

connect()

Syntax

```
static EgateXMLSchemaRegistry connect(String host, long port,
String username, String password)
```

Description

connect() connects to the Registry using the given argument values.

Parameters

Name	Type	Description
host	String	Host name (for example, **localhost**)
port	long	Port number (for example, **23001**)
username	String	User name (for example, **Administrator**)
password	String	Password for this username (for example, **STC**)

Return Values

Returns an instance of the **EgateXMLSchemaRegistry** object.

Throws

None.

listEgateSchemas()

Syntax

```
public Iterator listEgateSchemas()
```

Description

listEgateSchemas() queries all available e*Gate schemas in the currently connected Registry.

Parameters

None.

Return Values

Returns a list of their names (such as **MySchema**) in Iterator.

Throws

java.lang.Exception

listEgateEventTypes()

Syntax

```
public Iterator listEgateEventTypes(String egateSchemaName)
```

Description

listEgateEventTypes() lists all available Event Types for a given e*Gate schema name.

Parameters

Name	Type	Description
egateSchemaName	String	Name of an e*Gate schema (for example, **MySchema**)

Return Values

Returns the names of the Event Types.

Throws

java.lang.Exception

Example

A typical output resembles the following example:

```
00000000
GenericIn
GenericOut
Notification
```

Note: *The Iterator does not include the path or the file extension.*

listXMLSchemaFiles()

Syntax

```
public Iterator listXMLSchemaFiles(String egateEventName)
```

Description

listXMLSchemaFiles(), after **listEgateEventTypes()** has been called, lists all the available XML Schemas (**.xsd** files) in the **.jar** file associated with the given Event Type.

Note: *Do not call the **listXMLSchemaFiles()** method until after you have first called the **listEgateEventTypes()** method.*

Parameter

Name	Type	Description
egateEventName	String	Name of an e*Gate Event Type (for example, **adm**)

Return Values

Returns the names of the XML Schemas in the **.jar** file.

Throws

None.

Example

For example, for an Event Type named **adm**, you would set *egateEventName* to **adm** and this method would return a list of the **XMLSchemaFileReader** objects corresponding to the **.xsd** files in the **.jar** file associated with the **adm** Event Type.

close()

Syntax

```
public void close()
```

Description

close() closes the socket connection to the Registry and releases all resources appropriately.

Parameters

None.

Return Values

None.

Throws

None.

getXMLSchemaData()

Syntax

```
public byte[] getXMLSchemaData()
```

Description

getXMLSchemaData() queries the contents of the XML Schema (**.xsd**) file.

Parameters

None.

Return Values

Returns the contents of the file as a byte array.

Throws

None.

getXMLSchemaFileName()

Syntax

```
public String getXMLSchemaFileName()
```

Description

Queries the file name of the XML Schema (the **.xsd** file, including extension).

Parameters

None.

Return Values

Returns the file name.

Throws

None.

7.3 Sample Implementations

This section provides listings of sample programs that show the source code for several examples, including retrieving:

- Names of e*Gate schemas on a given Registry Host

- Names of Events in a given e*Gate schema

- File names and contents of the XML Schema associated with a given Event

The following table shows the parameter names used throughout the three sample programs provided in this section:

Table 2 Parameter Names in Examples

Name	Type	Value
host	String	"**localhost**" (all three implementations)
port	Long	**com.stc.common.registry.Registry.DEFAULT_PORT** (all three implementations)
username	String	"**Administrator**" (all three implementations)
password	String	"**STC**" (all three implementations)
egateSchemaName	String	"**RegInsAPITest**" (EventsRetrieve and GetXMLSchemaFile)
egateEventName	String	"**adm**" (GetXMLSchemaFile only)

7.3.1 SchemaListRetrieve.java

The **SchemaListRetrieve.java** program opens a connection to a given Registry, retrieves the names of all schemas in the Registry, and closes the socket. The following example shows a use of this program:

```java
import com.stc.eGateRegistryAPI.*;
import java.util.*;
import java.io.*;
public class SchemaListRetrieve
{
  public static void main(String arg[])
  {
    // this is the place you put your own information
    String host = "localhost";
    String schema = "RegInsAPITest";
    String username = "Administrator";
    String password = "STC";
    long port = com.stc.common.registry.Registry.DEFAULT_PORT;
    try
    {
      // get EgateXMLSchemaRegistry Object
      // by calling static method connect
      EgateXMLSchemaRegistry eGateXML=
      EgateXMLSchemaRegistry.connect(host,port,username,password);
      //************ Show available Schemas **********
      System.out.println("****** All Schemas *********");
      Iterator sIter = eGateXML.listEgateSchemas();

      while (sIter.hasNext())
      {
        String schema_name = (String)sIter.next();
        System.out.println("schema-> " + schema_name);
      }
      eGateXML.close();
    }
    catch(Exception e)
    {
    }
  }
}
```

7.3.2 EventsRetrieve.java

The **EventsRetrieve.java** program opens a connection to a given Registry, prints out a list of all Events in a given schema, and closes the socket. The following example shows a use of this program:

```java
import com.stc.eGateRegistryAPI.*;
import java.util.*;
import java.io.*;
public class EventsRetrieve
{
  public static void main(String arg[])
  {
    // this is the place you put your own information
    String host = "localhost";
    String schema = "RegInsAPITest";
    String username = "Administrator";
    String password = "STC";
    long port = com.stc.common.registry.Registry.DEFAULT_PORT;
```

```
        try
        {
          // get EgateXMLSchemaRegistry Object
          // by calling static method connect
          EgateXMLSchemaRegistry eGateXML=
          EgateXMLSchemaRegistry.connect(host,port,username,password);

          //************* Show Egate Events ***************
          System.out.println("**** Egate Events ****");
          Iterator iter=eGateXML.listEgateEventTypes(schema);

          while (iter.hasNext())
          {
            String event = (String)iter.next();
            System.out.println("event is " + event);
          }
          // ***********************************************
        // call close to close the connection to registry
        eGateXML.close();
        }
        catch(Exception e)
        {
        }
      }
    }
```

7.3.3 GetXMLSchemaFile.java

The **GetXMLSchemaFile.java** program opens a connection to a given Registry, retrieves the names of all XML Schema (**.xsd**) files associated with a given Event Type in a given schema, and closes the socket. The following example shows a use of this program:

```
import com.stc.eGateRegistryAPI.*;
import java.util.*;
import java.io.*;

public class GetXMLSchemaFile
{
  public static void main(String arg[])
  {
    // this is the place you put your own information
    String host = "localhost";
    String schema = "RegInsAPITest";
    String username = "Administrator";
    String password = "STC";
    long port = com.stc.common.registry.Registry.DEFAULT_PORT;

    try
    {

      // get EgateXMLSchemaRegistry Object
      // by calling static method connect
      EgateXMLSchemaRegistry eGateXML=
      EgateXMLSchemaRegistry.connect(host,port,username,password);

      //************* Show Egate Events ***************
      System.out.println("**** Egate Events ****");

      Iterator iter=eGateXML.listEgateEventTypes(schema);

      while (iter.hasNext())
```

```
        {
          String event = (String)iter.next();
          //System.out.println("event is " + event);
        }
        // Returns iterator to traverse  xmlSchemaReaders
        /**
         * You will need to put some xsd file in
         * your eventName.jar on the server side
         * The size you see should be the size of your xsd file
         */

        // change the Event Type to one from your own schemas
        Iterator iter2 =eGateXML.listXMLSchemaFiles("adm");

        while(iter2.hasNext())
        {
          XMLSchemaFileReader reader =
            (XMLSchemaFileReader)iter2.next();
          byte b[] = reader.getXMLSchemaData();
          String outputName = reader.getXMLSchemaFileName();
          File output= new File("c:\\testdir\\"+outputName);
          System.out.println("creating-> " + output.getName());
          output.createNewFile();
          FileOutputStream fout = new FileOutputStream(output);
          fout.write(b);
          System.out.println("in reader-> schema data
            size is " + b.length);
        }


        // call close to close the connection to registry
        eGateXML.close();

      }
      catch(Exception e)
      {
      }
    }
  }
```

Monk DTD Converter

This chapter provides an overview of the Monk DTD Converter's functionality and how it works with e*Gate Integrator. It also includes descriptions of the terms used throughout this chapter and provides sample files.

8.1 Monk XML Toolkit: Introduction

The Monk DTD Converter is a plug-in to the Event Type Definition (ETD) Editor in the Enterprise Manager. This section explains the feature's basic operation.

8.1.1 Using the Monk DTD Converter

The Monk ETD Editor is where the .**dtd** file is selected, which is then converted into an e*Gate ETD file with an .**ssc** extension. The resulting .**ssc** file follows the rules in the given DTD file and can map the input XML data file into its nodes. The mapped data can then be processed through the e*Gate system via the Collaboration process.

Figure 13 on page 68 shows how the Monk DTD Converter is used to convert a DTD file into a Monk ETD (**.ssc** file).

Figure 13   DTD-to-ETD Conversion Process
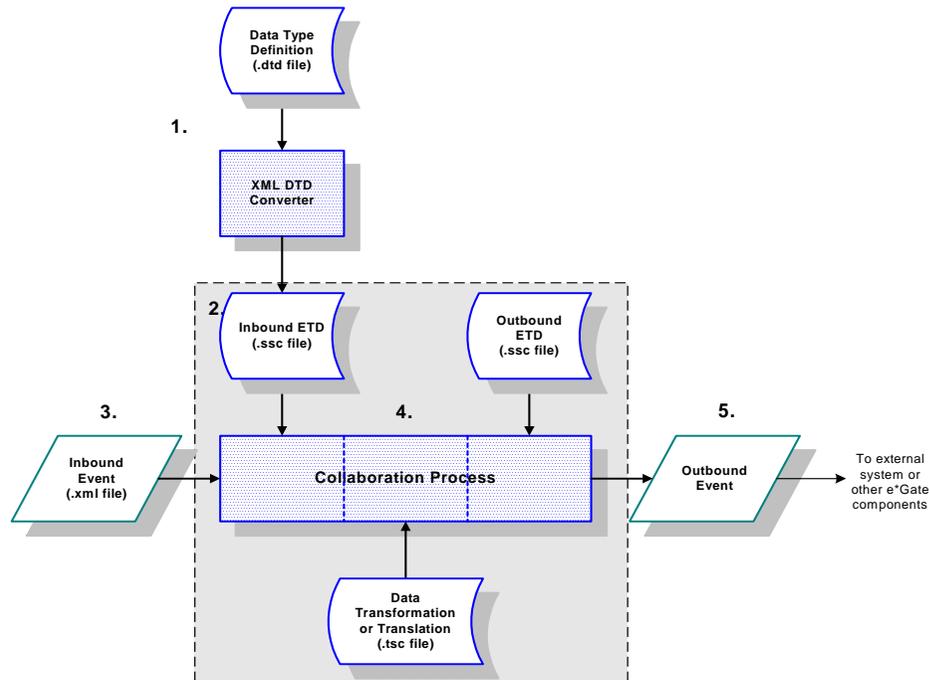
In general, this conversion process happens as follows:

1 The user opens the XML DTD Converter and directs it to read a DTD file as input.

2 The Converter creates an ETD file.

3 At runtime, data in XML format is sent into the e*Gate system.

4 The ETD created in Step 2 above is used to parse the data within a Collaboration.

5 The parsed Event data is then published for use by another external system or other e*Gate components.

The DTD file is converted into an ETD file only once when the e*Gate configuration is initially established. The generated ETD file successfully parses all data that complies with its source DTD file. If there are any changes required to the DTD file, the ETD file must be re-generated to match the changes in the XML input.

8.1.2 Operational Overview

The following diagram illustrates an example of the XML Toolkit components in the e*Gate Integrator environment.

Figure 14 XML Toolkit Components in Sample Configuration
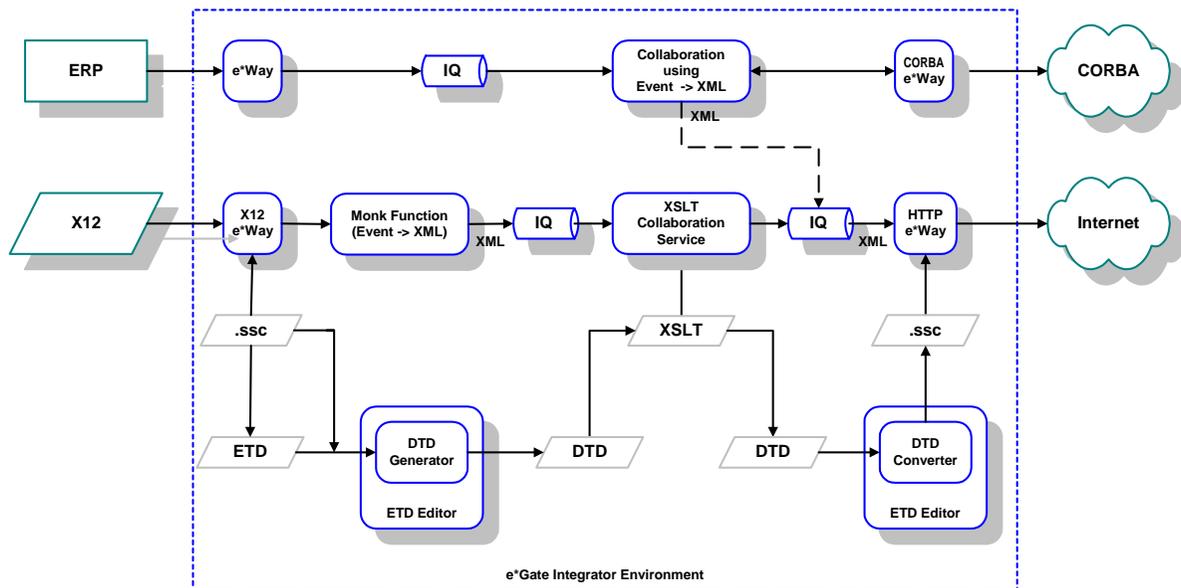
8.2 Feature Summary

The following table provides a summary of the Monk DTD Converter features.

Table 3 Monk DTD Converter Feature Summary

Feature	Explanation
Support for DOCTYPE/ SYSTEM/DTD in the output	Any document generated by the XML Toolkit must set the **DOCTYPE** parameter to point to the originating DTD or XML Schema.
Internal/external parameter entity	This refers to support entities, which are analogous to macros in C programming.

Table 3 Monk DTD Converter Feature Summary (Continued)

Feature	Explanation
Element occurrence indicators	The ssc generated by the Converter allows the Monk parser to recognize optional and repeating content for nodes based on indicators in the DTD or XML Schema. For example: \<ELEMENT alchoholic_drink (#PCDATA) \<ELEMENT vegetable (#PCDATA)> \<ELEMENT meat (#PCDATA)> \<ELEMENT starch (#PCDATA)> \<ELEMENT appetizer (#PCDATA)> \<ELEMENT dessert (#PCDATA)> \<ELEMENT dinner (alchoholic_drink+ |(appetizer?, meat, starch+, vegetable*, dessert*))> This example shows the three significant occurrence indicators in a DTD: + - one or more of ? - zero or one of * - zero or more of XML Schemas use a different notation that the three above qualifiers equate to supporting the minOccurs and maxOccurs qualifiers. These are similarly supported by the XML Toolkit.
Element choice	The "or" operator is supported. In XML DTDs, this is encoded with the "|" character. In XML Schemas, this is accomplished using the "choice" qualifier. The previous example demonstrates this feature, allowing a dinner to be either one or more "alchoholic_drinks" or the sequence of appetizers, meats, etc.
Element sequence	The sequence operator is supported as shown in the "dinner" element of the previous example. XML DTDs encode this with the comma operator. XML Schemas encode this with the "sequence" qualifier.
Default attributes	this supports the XML Schema capacity for setting default values in documents. This is documented in Sec. 2.2 "Complex Type Definitions, Element & Attribute Declaration" of the XML Schema Specification.
General entity	This refers to support entities. General entities apply to XML documents themselves (as opposed to parameter entities, which apply to DTDs).
Prefixed tag names	The Converter allows for other name space qualifier in the tag names.
Enumerated attributes	This refers to supported restricted values, which are defined in the XML Schema using the "restriction" key word.

8.3   Implementation

This section describes how to implement the XML DTD Converter in the ETD (Event Type Definition) Editor. Additional command line arguments are defined and node mapping is explained. Sample files and a sample conversion are included.

8.3.1 Using the XML DTD Converter

The Converter is accessed in the ETD Editor.

To access the XML DTD Converter using the Build tool

1 From the e*Gate Enterprise Manager screen, click the ETD Editor menu button ⊞ to launch the Event Type Definition Editor.

2 On the ETD Editor's Toolbar, click **Build**.

The Build an Event Type Definition dialog box appears (see the following figure).

Figure 15 Build an Event Type Definition Dialog Box — 1
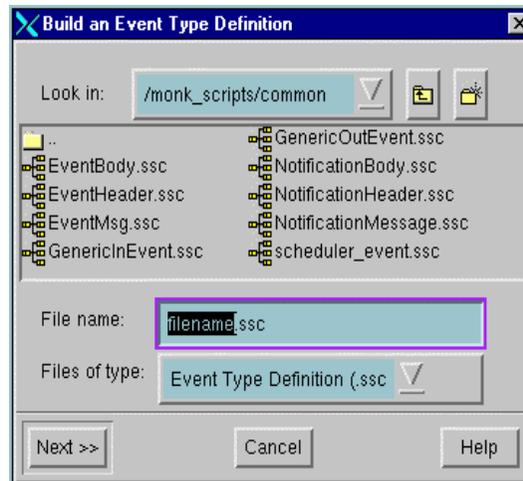
3 In the **File Name** box, type the name of the output file you wish to build. *Do not specify any file extension*. The Editor supplies the .**ssc** extension for you.

4   Click **Next**. Another Build an Event Type Definition dialog box appears (see the following figure),

Figure 16   Build an Event Type Definition Dialog Box — 2
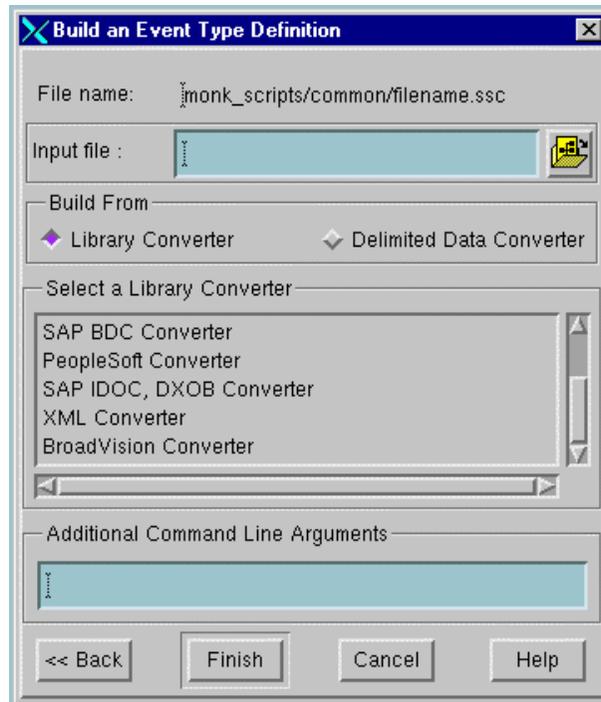
5   In the **Input file** box, type the name of the DTD input file.

6   Under **Build From**, select **Library Converter**.

7   Under **Select a Library Converter**, select **XML Converter**.

8   In the **Additional Command Line Arguments** box, specify the command line argument. Refer to the following section for a description of these command line arguments.

9   Click **Finish**.

Note: *The nodes are now generated in a collapsed GUI format by default. To generate them in an expanded format, use the Additional command-line argument **-expanded**.*

8.3.2 Command-line Arguments

The following switches allow you to control how the XML Converter generates the **.ssc** files:

abb
The XML DTD Converter uses abbreviated names when generating fixed message structure nodes.

depth <depth_level>

This is the maximum depth (of the tree) that the XML DTD Converter will generate in a message structure. This is useful for self-referential documents, i.e., rewind.

expanded

The default is not expanded. The message structure editor will not display all the nodes after the conversion. This speeds up the loading of the converted **.ssc** file.

noattlist

This tells the XML DTD Converter not to generate a message structure to handle the attribute list in XML. If you are sure your input XML files won't contain useful data in the attribute list, you can turn this on to speed up the parsing process of your input files.

nocdata (default)

The Converter will not generate the CDATA and SDATA nodes under the Data node. CDATA node is a node for mapping the CDATA section in XML. SDATA node is a node for mapping the data between the begin tag and the end tag, and is not CDATA section data.

root <root_element_name>

Specify the root directly as the XML DTD Converter may not be able to find the root element in the DTD file if the root element is not declared as the first element in the file.

treedepth <tree_depth_level>

Forces the Converter to stop generating the specified level. When running the XML DTD Converter, using the 'treedepth n' parameter restricts the depth of the generated ETD. You may need to experiment with different values for 'n' to find the optimal ETD size.

For example, the full ETD for a complicated DTD or XML Schema may be too large and take too long to generate to be practical. It may be necessary to restrict the size and complexity of the ETD, striking a balance between having enough nodes to do your **MonkID** or **MonkCollaboration**, but not having too many nodes to make the size of the **.ssc** file (and its memory requirements) too large.

xcomment

Extra nodes that map comments between sequence elements.

8.3.3 Understanding the ETD Structure

The first step to using the ETD is to understand the structure of the nodes in the context of the XML message being created. Each level is structured in the same way, so once you understand how the structure works you will be able to find the nodes you need to populate in your Collaboration Rules files.

The ETD contains a number of nodes that do not explicitly correlate to the XML DTD but are required by the Monk engine to parse the XML data correctly. These *facilitator* nodes are listed in **Table 4 on page 74**.

Table 4   Facilitator Nodes in the ETD

Name	Abbreviation	Description
Container	**CT**	A container node for an XML element. This node allows the short and long forms of XML tags to coexist in the structure.
Data Section	**DSN**	Identifies a data section within an XML element. This is the long form of the XML tag.
Data Set	**DS**	Identifies a data set within an XML element. The subelements within a data set can occur in any order.
Empty	**Empty**	This is the short form of the corresponding DSN node XML tag.
Comment	**CM**	XML comment.
Data	**Data**	This node holds the data for the element.
Attribute Set	**AS**	Identifies an XML attribute set within an XML element.
Equal Sign	**EQ**	The equal sign ("=") within an XML attribute.
Value	**Value**	This node holds the value for the XML attribute.
CDATA	CDATA	This node holds the CDATA Sections. (Used by system.)
SDATA	SDATA	This node holds the non-CDATA Sections. (Used by system.)

The facilitator nodes always occur in a set order and define the structure of the XML message. In the e*Xchange ETD, the facilitator nodes define the following types of branches:

- XML element without sub-elements
- XML element with sub-elements
- XML attribute

Figure 17 on page 75 illustrates the ETD structure for a basic XML sample:

Figure 17   XML Example
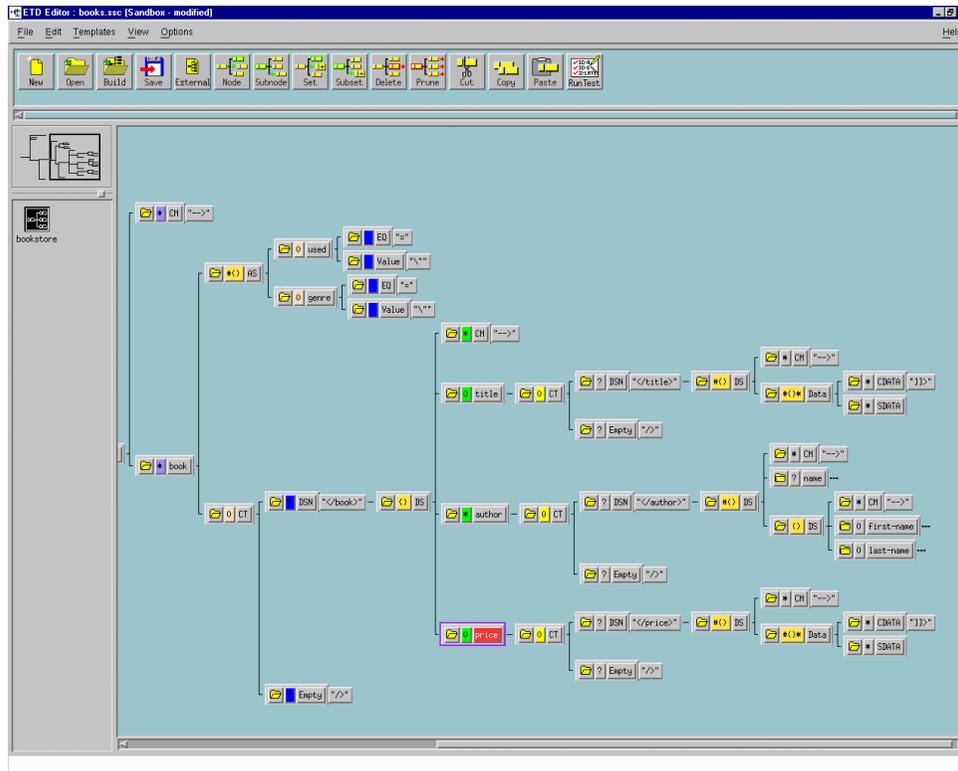
XML Element without Sub-elements

Figure 18 illustrates the ETD structure for an XML element that does not have sub-elements. Each XML element contains one child node, **CT**. **CT** identifies the parent node as an XML element. The **CT** node contains two child nodes: **DSN** and **Empty**. **DSN** maps the long form of the XML tag (*</tag>*) and **Empty** maps the short form (*</>*).

The **DSN** and **DS** nodes always occur as parent-child pairs. In this type of branch, **DS** is the parent node for two types of child nodes:

- **CM**, which holds XML comments for the element
- **Data**, which holds the data for the element

Figure 18   XML Element without Sub-elements
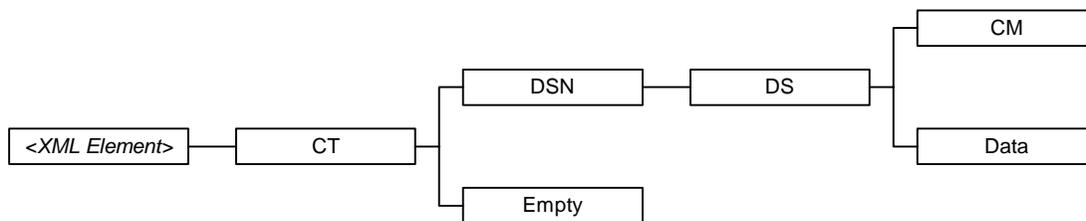
The **Data** node contains the actual data for the XML element that is defined. When you are creating your Collaboration Rules scripts, you must map your XML element data to the **Data** nodes at the terminal end of the element's branch.

XML Element with Sub-elements

The following figure illustrates the ETD structure for an XML element that has sub-elements.

Figure 19 XML Element with Sub-elements

Notice that the only difference between this diagram and the previous diagram are the **<Sub-element>** child nodes in place of the **Data** child node shown in Figure 1. The **DSN** and **DS** nodes always occur as parent-child pairs. In this type of branch, **DS** is the parent node for two types of child nodes:

- **CM**, which holds XML comments for the element

- **<Sub-element>**, the name of a sub-element of the parent element

The **DS** node will always contain a **CM** child node to hold XML comments. Each **<Sub-element>** node will then contain an ETD structure of its own, with the **<Sub-element>** node as the parent node for the branch.

XML Attribute

The following figure illustrates the ETD structure for an XML attribute.

Figure 20 XML Attribute
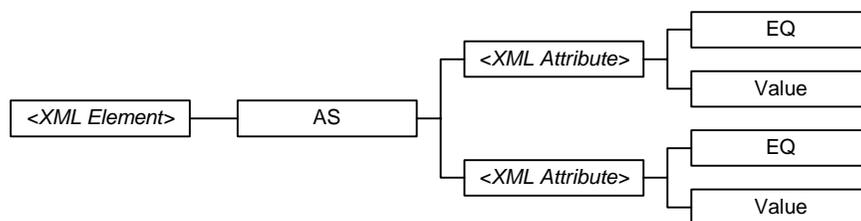
In this case, the XML element contains one child node, **AS**, which identifies the branch as XML attributes of the parent element. The **AS** node contains the **<XML Attribute>** nodes as child nodes.

Each **<XML Attribute>** node has two child nodes: **EQ** to represent the equal sign (=) in the attribute and **Value** which holds the actual value for the attribute. When you are

creating your Collaboration Rules scripts, you must map your XML attribute value to the **Value** nodes at the terminal end of the attribute's branch.

8.3.4 Using the ETD Editor

This section describes an additional feature that has been added to the ETD Editor window to provide support for XML.

Node Properties Dialog Box

The Node Properties dialog box is now reorganized into the following four tabs:

- **General** covers the general features, for example, file name and path (see Figure 21).

- **Content** covers tag characters, default characters, and scavenger data.

- **Repetition** allows you to enter order, group repetition, and matching characteristics, including the **N of N** attribute.

- **Delimiters** allows you to enter an expanded array of delimiter attributes, including a begin delimiter for a fixed-ETD node.

The following figure shows an example of the Node Properties dialog box.

Figure 21 Node Properties Dialog Box
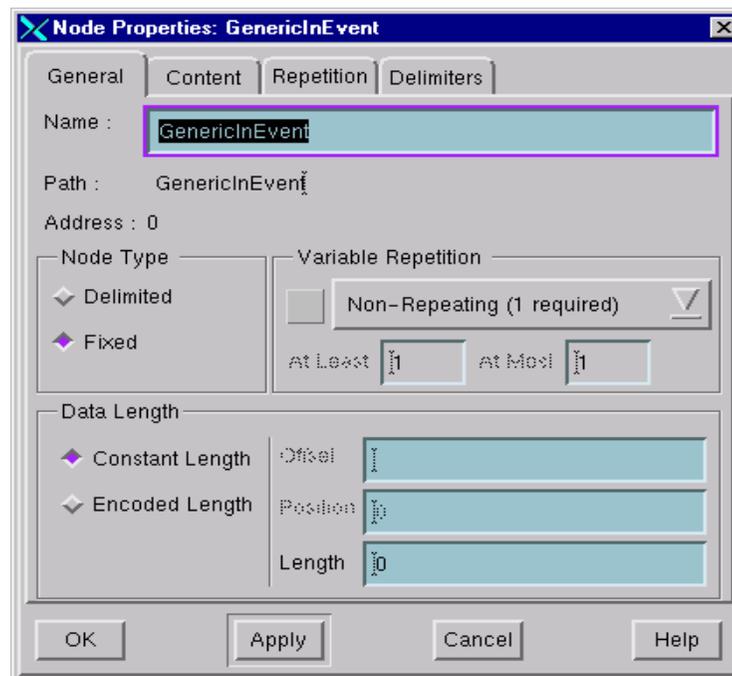
8.3.5 **Mapping**

This section explains how the Converter handles the XML-to-Monk data mapping.

Mapping for Elements

The generated ETD file is based on the elements defined in a DTD file. Each element is generated in the same pattern as shown in the following table.

Table 5 DTD Element Mapping Pattern

Line No.	Description
Line1	Comment of the ETD file showing the start of the element to be generated.
Line2	This line starts the block of ETD structure for handling the element.
Line3	"CT" (Container) is used for splitting the long form and short form of the XML element. The NofN (1 1) is used here to match exactly minimum 1 and maximum 1 case only (XOR logic).
Line4	"DSN" (DataSection) is used to map the data between begin tag and end tag for the XML long form. This node uses a begin delimiter and an end delimiter to match whatever is between those tags.
Line5	"DS" (DataSet) is used to map several kinds of information between a begin tag and an end tag such as #PCDATA, sub-elements and comments.
Line6	"CM" is used to map the XML comments.
Line7	"Data" is used to map the data portion of this element. In compliance with the XML specification, a regular expression is used here to stop the mapping whenever a "<" character is detected.
Line8	"CDATA" is used only when the data contains any "<" characters, and to map the CDATA Section data. Syntax: <![CDATA[Your data contains < character puts here]]>. This node won't be generated if optional argument -nocdata is specified. (For system use.)
Line9	"SDATA" is used to map non-CDATA Section data. This node won't be generated if optional argument -nocdata is specified. (For system use)
Line10	The right parenthesis for line 7.
Line11	The right parenthesis for line 5.
Line12	The right parenthesis for line 4.
Line13	"Empty" is used to map the short form of XML element. Example: <A /> (no end tag or data included)
Line14	The right parenthesis for line 3.
Line15	The right parenthesis for line 2.

The following example illustrates the pattern listed in the previous table:

```
1   ;: <=========== Name Begin Tag ===========>
2   (((ScN "\n \t\r") (Bd "<Name") required)
     Name OF 1 1 und und und und
3     (((NofN (1 1))) CT OF 1 1 und und und 0
4       (((ScN " \n\t\r") (">" "</Name>") beginanchored required )
         DSN ON 0 1 und und und 0
5        ((Gr) DS AS 1 1 und und und und
6          (((Sc "\n \t\r") ("<!--" "-->") beginanchored required )
            CM ON 0 INF und und und 0 )

7          ((Gr) Data AS 0 INF und und und 0
8            (( ("<![CDATA[" "]]>") beginanchored required )
              CDATA ON 0 INF und und und 0 )
9            (((ScN "\n \t\r")) SDATA OF 0 INF "\[^<\]\*" und und 0)
10           )
11         )        ;: End Data Set
12       ) ;: End Data Section
13      (((ScN " \n\t\r")(Ed "/>") endanchored required )
          Empty ON 0 1 und und und und)
14     )    ;: End Container
15  ) ;:= {0:Y}   ;: End Begin Tag Name
```

Mapping for Sub-elements

The child elements will be generated as in the "Mapping for Elements", the generated code will be inserted between Line10 and Line11 in "Mapping for Elements".

Mapping for Attributes

The section is generated between Line2 and Line3 in "Mapping for Elements" only if there is/are attribute definition(s) for the element (see the following table).

Table 6 Mapping for Attributes

Line No.	Description
Line1	This node called "AttSet" is for holding all the attributes for the element.
Line2	This node called "LastName" is used to map the name of the attribute "LastName" from the input XML data.
Line3	This node called "EQUAL_SIGN" is used to map the "=" character right after the attribute "LastName". The Sc "\n \t\r" node property is used to bypass the White Space from the input before the "=" from input.
Line4	This is the most important node called "Value" which maps whatever information between two double quotes (attribute value).
Line5	This is another definition of attribute "First Name", similar to Line2.
Line6	Same function as in Line3.
Line7	Same function as in Line4.
Line8	The right parenthesis for node "AttSet" which is the end of generation of the attribute section for a particular element.

The following example illustrates the pattern listed in the previous table:

```
1    ( AttSet AS 0 1 und und und und
2      (((Sc "\n \t\r") ( Bd "LastName" ) beginanchored ) LastName OF
1 1 und  und und und
3        (((Sc "\n \t\r") ( Ed "=" ) endanchored required ) EQUAL_SIGN
ON 1 1 und und und und)
4        (((Sc "\n \t\r") ("\"" "\"" ) anchored required ) Value ON 1
1 und und und und))
5      (((Sc "\n \t\r") ( Bd "FirstName" ) beginanchored ) FirstName OF
1 1 und und und und
6        (((Sc "\n \t\r") ( Ed "=" ) endanchored required ) EQUAL_SIGN
ON 1 1 und und und und)
7        (((Sc "\n \t\r") ("\"" "\"" ) anchored required ) Value ON 1
1 und und und und))
8    )    ;: AttSet
```

Mapping for Occurrence

The following table shows how XML symbols are mapped in the converted ETDs.

Table 7 Symbol Mapping

Symbols in XML	Mapping in ETD
Nothing after an sub-element definition.	1 Required
"?" after an sub-element definition.	0 or 1 occurrence
"*"after an sub-element definition.	0 or Infinite occurrence
"+"after an sub-element definition.	1 or Infinite occurrence

8.3.6 Sample Conversion

Here is an example of a DTD file that has been converted to an **.ssc** file (Event Type Definition).

Given DTD Example

```
<!ELEMENT bookstore (book)*>
<!ELEMENT book (title,author*,price)>
<!ATTLIST book genre CDATA #REQUIRED>
<!ATTLIST book used CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name | (first-name,last-name))>
<!ELEMENT price (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT first-name (#PCDATA)>
<!ELEMENT last-name (#PCDATA)>
```

Converted ETD from Example

```
;:- STC MsgStruct Version 3.1
;:- MsgStructure Header
;:- MsgStructure "bookstore"
;:- UserComment "  "
;:- Version "DataGate Version 3.1"
;:- FormatOption DELIMITED
;:- RepSeparator "Special Delimiter " ""
;:- Escape "Escape Character Delimiter " ""
;:- DefaultDelimiters "OTHER"
```

```
;:- Build 4.1.2.2502
;:- End MsgStructure Header

;:- Delimiter Structure
(define bookstore-delm '(
))

;:- Global Template Reference
;:- End Global Template Reference

;:- Local Template Definition
;:- End Local Template Definition

;:- MsgStructure Definition
(define bookstore-struct (message-convert (quote

((Ex) bookstore OF 1 1 und und und 0
( DS AS 1 1 und und und 0

  (((Sc "\n \t\r") ("<?" "?>") beginanchored required )
    PI ON 0 INF und und und 0
    (((Bd "xml") beginanchored ) XML_TAG OF 0 1 und und und 0
      (XML_DECL OF 0 1 und "" und und)
    )
  )

  (((Sc "\n \t\r") ("<!DOCTYPE" ">") anchored required )
    DOCTYPE ON 0 1 und " bookstore SYSTEM \"file:///d:/egate/client/
books.dtd\" " und und)

  (((Sc "\n \t\r") ("<!--" "-->") beginanchored required )
    CM ON 0 INF und und und 0 )


  ;: <=========== bookstore Begin Tag ===========>
  (((ScN "\n \t\r") (Bd "<bookstore") required)
    bookstore OF 1 1 und und und und
    (((NofN (1 1))) CT OF 1 1 und und und 0
      (((ScN " \n\t\r") (">" "</bookstore>") beginanchored required )
        DSN ON 0 1 und und und 0
        ((Gr) DS AS 0 1 und und und und
          (((Sc "\n \t\r") ("<!--" "-->") beginanchored required )
            CM ON 0 INF und und und 0 )

          ;: <=========== book Begin Tag ===========>
          (((Sc "\n \t\r") (Bd "<book") beginanchored)
            book OF 0 INF und und und und

            ( AS AS 1 1 und und und und
             (((Sc "\n \t\r") ( Bd "used" ) beginanchored ) used OF 1
1 und und und und
               (((ScN "\n \t\r") ( Ed "=" ) endanchored required ) EQ
ON 1 1 und und und und)
               (((ScN "\n \t\r") ("\"" "\"" ) anchored required )
Value ON 1 1 und und und und))
             (((Sc "\n \t\r") ( Bd "genre" ) beginanchored ) genre OF
1 1 und und und und
               (((ScN "\n \t\r") ( Ed "=" ) endanchored required ) EQ
ON 1 1 und und und und)
               (((ScN "\n \t\r") ("\"" "\"" ) anchored required )
Value ON 1 1 und und und und))
             );: AS
             (((NofN (1 1))) CT OF 1 1 und und und 0
```

```
                    (((ScN " \n\t\r") (">" "</book>") beginanchored required
)
                    DSN ON 0 1 und und und 0

                    (DS OS 1 1 und und und und
                       (((Sc "\n \t\r") ("<!--" "-->") beginanchored
required )
                       CM ON 0 INF und und und 0 )

                       ;: <=========== title Begin Tag ===========>
                    (((Sc "\n \t\r") (Bd "<title") beginanchored)
                       title OF 1 1 und und und und
                       (((NofN (1 1))) CT OF 1 1 und und und 0
                       (((ScN " \n\t\r") (">" "</title>") beginanchored
required )
                          DSN ON 0 1 und und und 0
                          ((Gr) DS AS 1 1 und und und und
                          (((Sc "\n \t\r") ("<!--" "-->") beginanchored
required )
                             CM ON 0 INF und und und 0 )

                             ((Gr) Data AS 0 INF und und und 0
                                (( ("<![CDATA[" "]]>") beginanchored
required )
                                   CDATA ON 0 INF und und und 0 )
                                (((ScN "\n \t\r")) SDATA OF 0 INF "\[^<\]\*"
und und 0)
                                )
                             );: End Data Set
                          );: End Data Section
                       (((ScN " \n\t\r")(Ed "/>") endanchored required )
                          Empty ON 0 1 und und und und)
                       );: End Container
                    ) ;:= {0:Y};: End Begin Tag title

                       ;: <=========== author Begin Tag ===========>
                    (((Sc "\n \t\r") (Bd "<author") beginanchored)
                       author OF 0 INF und und und und
                       (((NofN (1 1))) CT OF 1 1 und und und 0
                       (((ScN " \n\t\r") (">" "</author>") beginanchored
required )
                          DSN ON 0 1 und und und 0

                          ((Gr) DS AS 1 1 und und und und
                          (((Sc "\n \t\r") ("<!--" "-->") beginanchored
required )
                             CM ON 0 INF und und und 0 )

                             ;: <=========== name Begin Tag ===========>
                          (((Sc "\n \t\r") (Bd "<name") beginanchored)
                             name OF 0 1 und und und und
                             (((NofN (1 1))) CT OF 1 1 und und und 0
                                (((ScN " \n\t\r") (">" "</name>")
beginanchored required )
                                   DSN ON 0 1 und und und 0
                                   ((Gr) DS AS 1 1 und und und und
                                      (((Sc "\n \t\r") ("<!--" "-->")
beginanchored required )
                                         CM ON 0 INF und und und 0 )

                                      ((Gr) Data AS 0 INF und und und 0
                                      (( ("<![CDATA[" "]]>") beginanchored
required )
                                            CDATA ON 0 INF und und und 0 )
```

```
                                                 (((ScN "\n \t\r")) SDATA OF 0 INF
"\[^<\]\*" und und 0)
                                                   )
                                                 );: End Data Set
                                               );: End Data Section
                                               (((ScN " \n\t\r")(Ed "/>") endanchored
required )
                                                   Empty ON 0 1 und und und und)
                                             );: End Container
                                           ) ;:= {0:Y};: End Begin Tag name

                                           (DS_1 OS 0 INF und und und und
                                             (((Sc "\n \t\r") ("<!--" "-->")
beginanchored required )
                                                 CM ON 0 INF und und und 0 )

                                             ;: <=========== first-name Begin Tag
===========>
                                             (((Sc "\n \t\r") (Bd "<first-name")
beginanchored)
                                               first-name OF 1 1 und und und und
                                               (((NofN (1 1))) CT OF 1 1 und und und 0
                                               (((ScN " \n\t\r") (">" "</first-name>")
beginanchored required )
                                                   DSN ON 0 1 und und und 0
                                                   ((Gr) DS AS 1 1 und und und und
                                                     (((Sc "\n \t\r") ("<!--" "-->")
beginanchored required )
                                                       CM ON 0 INF und und und 0 )

                                                     ((Gr) Data AS 0 INF und und und 0
                                                       (( ("<![CDATA[" "]]>")
beginanchored required )
                                                         CDATA ON 0 INF und und und 0 )
                                                     (((ScN "\n \t\r")) SDATA OF 0 INF
"\[^<\]\*" und und 0)
                                                       )
                                                     );: End Data Set
                                                   );: End Data Section
                                                 (((ScN " \n\t\r")(Ed "/>") endanchored
required )
                                                     Empty ON 0 1 und und und und)
                                                 );: End Container
                                           ) ;:= {0:Y};: End Begin Tag first-name

                                             ;: <=========== last-name Begin Tag
===========>
                                             (((Sc "\n \t\r") (Bd "<last-name")
beginanchored)
                                               last-name OF 1 1 und und und und
                                               (((NofN (1 1))) CT OF 1 1 und und und 0
                                               (((ScN " \n\t\r") (">" "</last-name>")
beginanchored required )
                                                   DSN ON 0 1 und und und 0
                                                   ((Gr) DS AS 1 1 und und und und
                                                     (((Sc "\n \t\r") ("<!--" "-->")
beginanchored required )
                                                       CM ON 0 INF und und und 0 )

                                                     ((Gr) Data AS 0 INF und und und 0
                                                       (( ("<![CDATA[" "]]>")
beginanchored required )
                                                         CDATA ON 0 INF und und und 0 )
```

```
                                         (((ScN "\n \t\r")) SDATA OF 0 INF
"\[^<\]\*" und und 0)
                                         )
                                       );: End Data Set
                                      );: End Data Section
                                    (((ScN " \n\t\r")(Ed "/>") endanchored
required )
                                       Empty ON 0 1 und und und und)
                                     );: End Container
                                   ) ;:= {0:Y};: End Begin Tag last-name
                                  );: End Sequence DataSet
                               );: End Non-Sequence DataSet
                             );: End Data Section
                           (((ScN " \n\t\r")(Ed "/>") endanchored required )
                              Empty ON 0 1 und und und und)
                          );: End Container
                        ) ;:= {0:Y};: End Begin Tag author

                        ;: <=========== price Begin Tag ===========>
                        (((Sc "\n \t\r") (Bd "<price") beginanchored)
                         price OF 1 1 und und und
                         (((NofN (1 1))) CT OF 1 1 und und und 0
                         (((ScN " \n\t\r") (">" "</price>") beginanchored
required )
                             DSN ON 0 1 und und und 0
                             ((Gr) DS AS 1 1 und und und und
                             (((Sc "\n \t\r") ("<!--" "-->") beginanchored
required )
                                CM ON 0 INF und und und 0 )

                               ((Gr) Data AS 0 INF und und und 0
                                 (( ("<![CDATA[" "]]>") beginanchored
required )
                                   CDATA ON 0 INF und und und 0 )
                                (((ScN "\n \t\r")) SDATA OF 0 INF "\[^<\]\*"
und und 0)
                                 )
                               );: End Data Set
                              );: End Data Section
                             (((ScN " \n\t\r")(Ed "/>") endanchored required )
                                Empty ON 0 1 und und und und)
                           );: End Container
                         ) ;:= {0:Y};: End Begin Tag price
                       );: End Sequence DataSet
                     );: End Data Section
                    (((ScN " \n\t\r")(Ed "/>") endanchored required )
                       Empty ON 0 1 und und und und)
                  );: End Container
                ) ;:= {0:Y};: End Begin Tag book
             );: End Data Set
           );: End Data Section
          (((ScN " \n\t\r")(Ed "/>") endanchored required )
             Empty ON 0 1 und und und und)
        );: End Container
      ) ;:= {0:Y};: End Begin Tag bookstore
))    ;: End Root Nodebookstore
)))
;:- End MsgStructure Definition

Sample Input XML File
<?xml version='1.0' encoding="ASCII" ?>
<?kill me?>
<!DOCTYPE bookstore SYSTEM "abcde" >
<bookstore>
```

```
<book genre = "ISBN1234567" used="T" >
    <author>
        <name>
        STC Document Department
        </name>
    </author>
    <title>
    Monk Programmer's Reference Guide
    </title>
    <price>
    $200.00
    </price>
</book>
<book used="F" genre="ISBN7654321">
    <author>
        <first-name>
        Stc
        </first-name>
        <last-name>
        Writer #1
        </last-name>
    </author>
    <price>
    $100.00
    </price>
    <title>
    Editor User's Guide
    </title>
</book>
</bookstore>
```

Monk XML Schema Converter

The Monk XML Schema Converter enables you to convert XML Schema files (**.xsd**) to Event Type Definition (ETD) files with an **.ssc** extension. This chapter explains how this functionality works with e*Gate Integrator.

9.1 XML Schemas and Monk: Introduction

e*Gate users have the ability to convert **.xsd** files to **.ssc** files using the Monk XML Schema Converter. This Converter converts an XML Schema file (**.xsd**) to a SeeBeyond ETD file (**.ssc)**.

Note: The XML Data Reduction format (.xdr) is not yet supported in this version.

9.2 XML Schema Versions: Monk

This release of the Monk XML Schema Builder supports the World Wide Web Consortium (W3C) XML Schema standard as of the 2000-04-07 W3C Recommendation. Documentation for this standard can be found at:

http://www.w3.org/XML/Schema

Any newer version of XML Schema may not be processed properly.

The XML Schema defines the rules of how the XML document looks. It is similar to the data type definition (DTD) file in that it defines the relationships of elements that appear in the target document.

The XML Schema is actually a collection of element types and attribute types. The element types define the skeleton of the XML document under the parent/children relationship paradigm. The attribute types only define attributes for element types and do not have any relationships to other attribute types.

9.3 How Monk XML Schema Converter Works

The Monk XML Schema Converter is a plug-in to the ETD Editor in the e*Gate Enterprise Manager. The Editor is where the **.xsd** file is selected, which is then converted into an e*Gate Event Type Definition file with an **.ssc** extension.

The resulting **.ssc** file follows the rules in the given XML Schema file and can map the input XML data file into its nodes. The mapped data can then be processed through the e*Gate system via the Collaboration process.

The following figure shows how the Monk XML Schema Converter converts an XML Schema (**.xsd** file) into an ETD (**.ssc** file).

Figure 22   Monk XML Schema Conversion Process
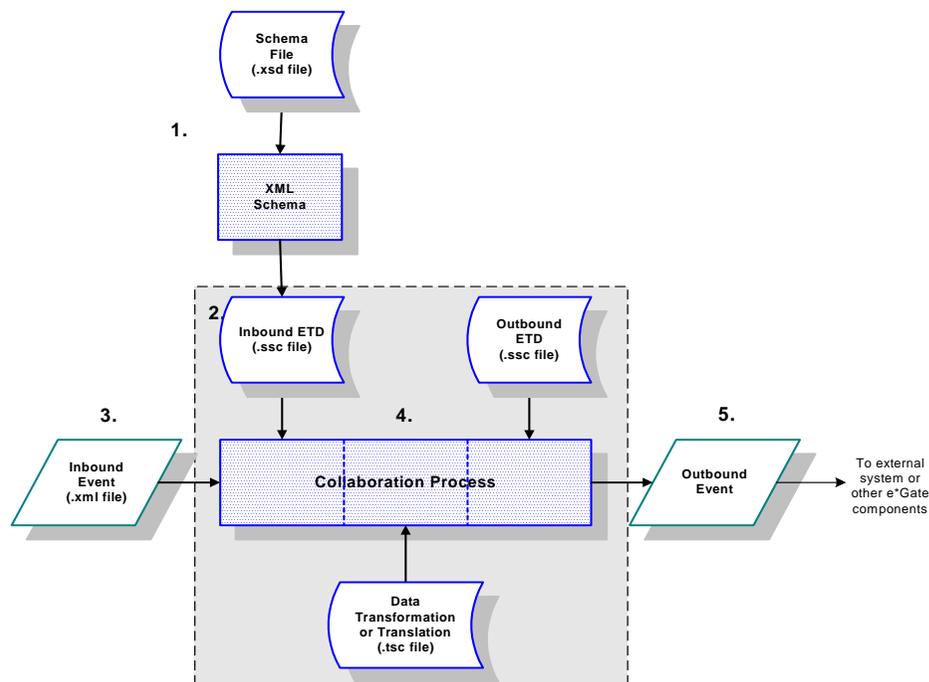
In general, this conversion process happens as follows:

1 The user opens the XML Schema Converter and directs it to read a **.xsd** file as input.

2 The Converter creates an ETD file.

3 At runtime, data in XML format is sent in the e*Gate system.

4 The ETD created in Step 2 above is used to parse the data within a Collaboration.

5 The parsed Event data is then published for use by another external system or other e*Gate components.

The **.xsd** file is converted into an ETD file only once when the e*Gate configuration is initially established. The generated ETD file successfully parses all data that complies with its source **.xsd** file. If there are any changes required to the **.xsd** file, the ETD file must be re-generated to match the changes in the XML input.

9.4 Feature Summary

The following table provides a feature summary for the Monk XML Schema Converter.

Table 8 Monk XML Schema Converter Feature Summary

Feature	Explanation
Support for DOCTYPE/SYSTEM/DTD in the output	Any document generated by the XML Toolkit must set the **DOCTYPE** parameter to point to the originating DTD or XML Schema.
Internal/external parameter entity	This refers to support entities, which are analogous to macros in C programming.
Element occurrence indicators	The ssc generated by the Converter allows the Monk parser to recognize optional and repeating content for nodes based on indicators in the DTD or XML Schema. For example: <ELEMENT alchoholic_drink (#PCDATA) <ELEMENT vegetable (#PCDATA)> <ELEMENT meat (#PCDATA)> <ELEMENT starch (#PCDATA)> <ELEMENT appetizer (#PCDATA)> <ELEMENT dessert (#PCDATA)> <ELEMENT dinner (alchoholic_drink+ |(appetizer?, meat, starch+, vegetable*, dessert*))> This example shows the three significant occurrence indicators in a DTD: + - one or more of ? - zero or one of * - zero or more of XML Schemas use a different notation that the three above qualifiers equate to supporting the **minOccurs** and **maxOccurs** qualifiers. These are similarly supported by the XML Toolkit.
Element choice	The "or" operator is supported. In XML DTDs, this is encoded with the "|" character. In XML Schemas, this is accomplished using the "choice" qualifier. The previous example demonstrates this feature, allowing a dinner to be either one or more "alchoholic_drinks" or the sequence of appetizers, meats, etc.
Element sequence	The sequence operator is supported as shown in the "dinner" element of the previous example. XML DTDs encode this with the comma operator. XML Schemas encode this with the "sequence" qualifier.
Default attributes	this supports the XML Schema capacity for setting default values in documents. This is documented in Sec. 2.2 "Complex Type Definitions, Element & Attribute Declaration" of the XML Schema Specification.
General entity	This refers to support entities. General entities apply to XML documents themselves (as opposed to parameter entities, which apply to DTDs).

9.5 Implementation

This section explains how to implement XML Schema in the ETD Editor. Additional command-line arguments are defined and node mapping is explained. Sample files and a sample conversion are included.

9.5.1 Using XML Schema

XML Schema is accessed in the ETD (Event Type Definition) Editor.

To access XML Schema using the Build tool

1   From the e*Gate Enterprise Manager window, click the ETD Editor Toolbar button
    ▉ to launch the ETD Editor.

2   On the ETD Editor's Toolbar, click **Build**.

    The Build an Event Type Definition dialog box appears (see **Figure 24 on page 90**).

Figure 23   Build an Event Type Definition Dialog Box — 3
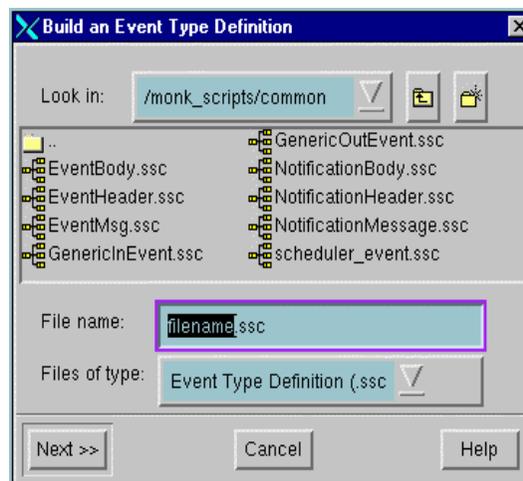
3   In the **File Name** box, type the name of the output file you wish to build. *Do not specify any file extension*—the Editor will supply an "ssc" extension for you.

4 Click **Next**. Another Build an Event Type Definition dialog box appears (see the following figure).

Figure 24 Build an Event Type Definition Dialog Box — 4
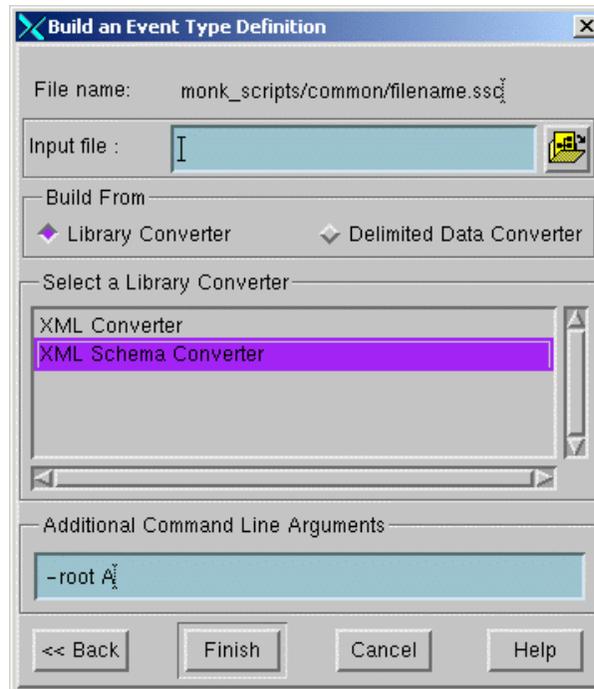
5 In the **Input file** box, type the path location and name of the **.xsd** input file you want to use.

6 Under **Build From**, select **Library Converter**.

7 Under **Select a Library Converter**, select **XML Schema Converter**.

8 In the **Additional Command Line Arguments** box, specify the command line argument **-root** <root element name>. Refer to the following section for a description of these command line arguments.

9 Click **Finish**.

Note: *The nodes are now generated in a collapsed GUI format by default. To generate them in an expanded format, use the Additional Command Line argument "-expanded".*

9.5.2 **Command-line Arguments**

The following switches allow you to control how XML Schema generates the **.ssc** files:

abb
XML Schema uses abbreviated names when generating fixed message structure nodes.

depth <num>
This is the maximum depth (of the tree) that XML Schema will generate for element loops.

expanded
>All nodes will be expanded in the GUI.

noattlist
>This tells XML Schema not to generate a message structure to handle the attribute list in XML. If you are sure your input XML files won't contain useful data in the attribute list, you can turn this on to speed up the parsing process of your input files.

nsattr
>This generates the extra structure for each element to map the "xmins" attributes for the namespace. You need to specify the *-root* every time, the other parameters are the same.

root <name>
>Specify the root directly, as XML Schema may not be able to find the root element in the DTD file if the root element is not declared as the first element in the file.

9.5.3 Understanding the ETD Structure

The first step to using the ETD is to understand the structure of the nodes in the context of the XML message being created. Each level is structured in the same way, so once you understand how the structure works you will be able to find the nodes you need to populate in your Collaboration Rules files.

The ETD contains a number of nodes that do not explicitly correlate to the XML Schema file but are required by the Monk engine to parse the XML data correctly. These *facilitator* nodes are listed in **Table 9 on page 91**.

Table 9   Facilitator Nodes in the ETD

| Name	Abbreviation	Description
Container	**CT**	A container node for an XML element. This node allows the short and long forms of XML tags to coexist in the structure.
Data Section	**DSN**	Identifies a data section within an XML element. This is the long form of the XML tag.
Data Set	**DS**	Identifies a data set within an XML element. The subelements within a data set can occur in any order.
Empty	**Empty**	This is the short form of the corresponding DSN node XML tag.
Comment	**CM**	XML comment.
Data	**Data**	This node holds the data for the element.
Attribute Set	**AS**	Identifies an XML attribute set within an XML element.
Equal Sign	**EQ**	The equal sign ("=") within an XML attribute.
Value	**Value**	This node holds the value for the XML attribute.
CDATA	CDATA	This node holds the CDATA Sections. (Used by system.)
SDATA	SDATA	This node holds the non-CDATA Sections. (Used by system.)

The facilitator nodes always occur in a set order and define the structure of the XML message. In the e*Xchange ETD, the facilitator nodes define three types of branches:

  ▪ XML element without sub-elements

  ▪ XML element with sub-elements

  ▪ XML attribute

Figure 25 on page 92 illustrates the ETD structure for a basic XML sample:

Figure 25   XML Sample

XML Element without Sub-elements

Figure 26 on page 93 illustrates the ETD structure for an XML element that does not have sub-elements. Each XML element contains one child node, **CT**. **CT** identifies the parent node as an XML element. The **CT** node contains two child nodes: **DSN** and **Empty**. **DSN** maps the long form of the XML tag (</*tag*>) and **Empty** maps the short form (</>).

The **DSN** and **DS** nodes always occur as parent-child pairs. In this type of branch, **DS** is the parent node for two types of child nodes:

  ▪ **CM**, which holds XML comments for the element

  ▪ **Data**, which holds the data for the element

Figure 26   XML Element without Sub-elements
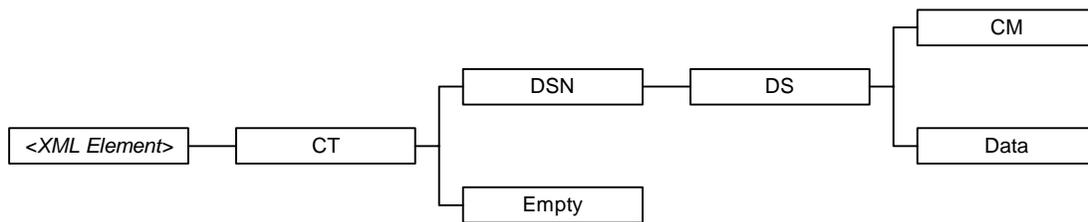
The **Data** node contains the actual data for the XML element that is defined. When you are creating your Collaboration Rules scripts, you must map your XML element data to the **Data** nodes at the terminal end of the element's branch.

## XML Element with Sub-elements

The following figure illustrates the ETD structure for an XML element that has sub-elements.

Figure 27   XML Element with Sub-elements

Notice that the only difference between this diagram and the previous diagram are the *<Sub-element>* child nodes in place of the **Data** child node shown in Figure 1. The **DSN** and **DS** nodes always occur as parent-child pairs. In this type of branch, **DS** is the parent node for two types of child nodes:

- **CM**, which holds XML comments for the element

- *<Sub-element>*, the name of a sub-element of the parent element

The **DS** node will always contain a **CM** child node to hold XML comments. Each *<Sub-element>* node will then contain an ETD structure of its own, with the *<Sub-element>* node as the parent node for the branch.

XML Attribute

The following figure illustrates the ETD structure for an XML attribute.

Figure 28   XML Attribute
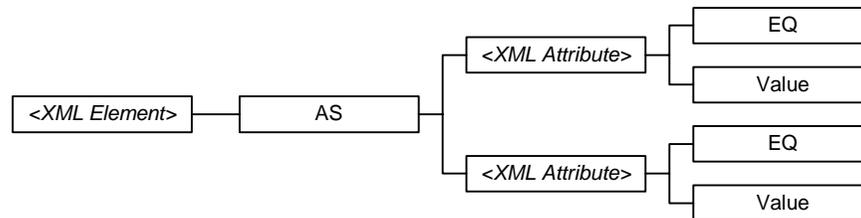
In this case, the XML element contains one child node, **AS**, which identifies the branch as XML attributes of the parent element. The **AS** node contains the **<XML Attribute>** nodes as child nodes.

Each **<XML Attribute>** node has two child nodes: **EQ** to represent the equal sign (=) in the attribute and **Value** which holds the actual value for the attribute. When you are creating your Collaboration Rules scripts, you must map your XML attribute value to the **Value** nodes at the terminal end of the attribute's branch.

9.5.4 Using the ETD Editor

This section describes an additional feature that has been added to the ETD Editor window to provide support for XML.

Node Properties Dialog Box

The Node Properties dialog box is now reorganized into the following tabs:

- **General** covers the general features, for example, file name and path (see **Figure 29 on page 95**).

- **Content** covers tag characters, default characters, and scavenger data.

- **Repetition** allows you to enter order, group repetition, and matching characteristics, including the **N of N** attribute.

- **Delimiters** allows you to enter an expanded array of delimiter attributes, including a begin delimiter for a fixed-ETD node.

Figure 29 on page 95 shows an example of the Node Properties dialog box.

Figure 29   Node Properties Dialog Box
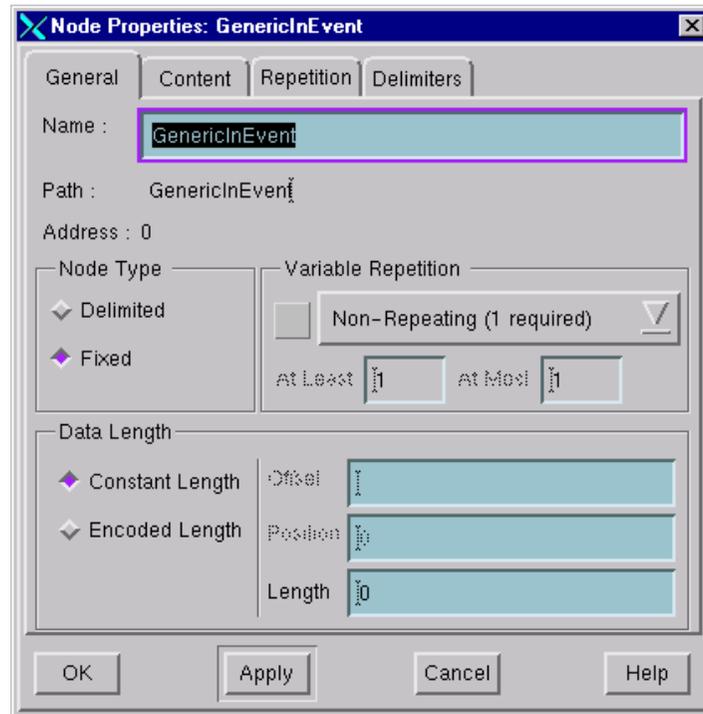
9.5.5 XML Schema Implementation Examples

The following XML Schema example shows how to define element types and attribute types.

Bookstore.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

        <xsd:element name="bookstore">
                <xsd:complexType content="elementOnly">
                        <xsd:sequence minOccurs="0"
maxOccurs="unbounded">
                                <xsd:element name="book"
type="bookType"/>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>

        <xsd:complexType name="authorType" content="elementOnly">
                <xsd:choice>
                        <xsd:element name="name" type="xsd:string"/>
                        <xsd:sequence>
                                <xsd:element name="first-name"
type="xsd:string"
/>
                                <xsd:element name="last-name"
type="xsd:string"/
>
                        </xsd:sequence>
                </xsd:choice>
```

```
            </xsd:complexType>

            <xsd:complexType name="bookType" content="elementOnly">
                    <xsd:sequence>
                            <xsd:element name="title" type="xsd:string"/>
                            <xsd:element name="author" type="authorType"
minOccurs="
0" maxOccurs="unbounded"/>
                            <xsd:element name="price" type="xsd:string"/>
                    </xsd:sequence>
                    <xsd:attribute name="genre" type="xsd:string"
use="required"/>
                    <xsd:attribute name="used" type="xsd:string"
use="required"/>
            </xsd:complexType>

    </xsd:schema>
```

Explanation

In the previous XML Schema, the element type **bookstore** is the root element. It contains a sequence of the element type **book** and occurrences from zero to infinite. The book's attribute **type** is set to point to **bookType**, which means it is referring to another definition. As you can see, the **bookType** definition near the end of the example could be used with other elements as well.

The reason that **bookType** is not defined as an element type is because it does not appear on the target XML document. In the **bookType** complex type definition, these attributes are defined: **name, genre**, and **used**. These attributes belong to **bookType** and automatically belong to the element type **book**, as well.

In the previous example, you can see many uses of the text string **xsd:** because it is the prefix for the XML namespace defined at the top of the schema as:

   http://www.w3.org/1999/XMLSchema

Do not try to resolve this URL. Actually, it is in the URI format. The purpose of the XML namespace is to make sure that each element's name is unique.

When designing an XML document, it is important to identify every piece of content to send and receive. Once all the single pieces of information are identified, the relation between them becomes obvious. The next step is to create the XML Schema to describe the content of each piece of information and their relationships.

Generally, reusing a definition is recommended because it saves time, and consistency is not an issue. The following example illustrates how to reuse an element type defined from another schema:

Purchaseorder.xsd

```
    <schema xmlns="http://www.w3.org/1999/XMLSchema">

    <include schemaLocation="Address.xsd" />

    < element name="purchaseOrder" type="PurchaseOrderType"/>

    <complexType name="PurchaseOrderType">
            <element name="shipTo" type="Address"/>
    <element name="items"/>
    <attribute name="orderDate" type="date"/>
```

```
            </complexType>

        </schema>
```

Address.xsd

```
        <schema xmlns="http://www.w3.org/1999/XMLSchema">

        <xsd:complexType name="Address">
            <xsd:element name="name"   type="xsd:string"/>
            <xsd:element name="street" type="xsd:string"/>
            <xsd:element name="city"   type="xsd:string"/>
            <xsd:element name="state"  type="xsd:string"/>
            <xsd:element name="zip"    type="xsd:decimal"/>
            <xsd:attribute name="country" type="xsd:NMTOKEN"
                    use="fixed" value="US"/>
        </xsd:complexType>
        </schema>
```

The address itself is from another schema and is being used in **PurchaseOrder.xsd**.

Supported Components and Features

- Element declarations
- Attribute group definitions
- Attribute declarations
- Complex type definitions
- Simple type definitions
- Group declarations
- All, choice, sequence declarations
- Annotations
- Type derivation
- Anonymous types
- Nested element declaration
- Include
- Import

XSLT Collaboration Service

This chapter explains how to use the Monk XSLT Collaboration Service (XSLT stands for Extensible Stylesheet Language Transformations) and how to create its associated Collaboration Rules.

10.1 Introduction

The Monk XSLT Collaboration Service enables the development of external Collaboration Rules that extracts selected information from a well-formed XML document, transform the information, and output it in another well-formed XML document. It also enables users to transform an XML document into another XML document with a different set (or subset) of data or tags.

10.1.1 Requirements

The XSLT Collaboration Service runs on Windows and requires:

- e*Gate version 4.1 or later

- MSXML3 Parser Technology Preview Release from Microsoft. The latest release can be found at:

    http://msdn.microsoft.com/downloads/c-frame.htm?/downloads/ webtechnology/xml/msxml.asp

*Note: **XMLINST.EXE**, the utility program that comes with MSXML and is installed in the \WINDOWS\SYSTEM32\directory, must be executed. This causes MSXML to be set as the default XML module rather than an older **MSXML.dll**.*

10.1.2 Architecture

This interface prescribes that the XSLT translate file (**.xsl**) must be written in a well-formed XML document using the Extensible Stylesheet Language (XSL). The implementation will accept a well-formed XML document as input and will return a well-formed XML document as output.

10.2 Creating XSLT Collaboration Rules

Collaboration Rules must be created for the XSLT Collaboration Service. The procedure given in this section describes how to commit an **.xsl** file to the Sandbox and create a Collaboration Rule.

Files can be committed to the Registry using one of two ways. This document describes the **stcregutil.exe** command line utility, implementing the **-fr** and **-fc** commands. Another option is to use the Enterprise Manager's **Commit to Sandbox** (the command is located on the **File** menu) method, followed by the **Promote to Run Time** option.

10.2.1 Committing .xsl Files to the Registry

To use the stcregutil Command Line Utility

- Run the **stcregutil** utility by typing the following text at the command line:

```
stcregutil –rh registry –rs schema –un user-name
   –up password -fc monk_scripts/common file.xsl
```

*Note: For more information about the **stcregutil.exe** command-line utility, see the **e*Gate Integrator System Administration and Operations Guide**. The example is printed on more than one line for clarity, but must be issued as a single command line.*

To use the Commit to Sandbox and Promoting to Run Time Method

1 Launch the e*Gate Enterprise Manager.

2 From the **File** menu, select **Commit to Sandbox.**

3 Select and open the local **.xsl** file to be made available to the XSLT Collaboration Service (see **Figure 30 on page 100**).

Figure 30   Select Local file To Commit
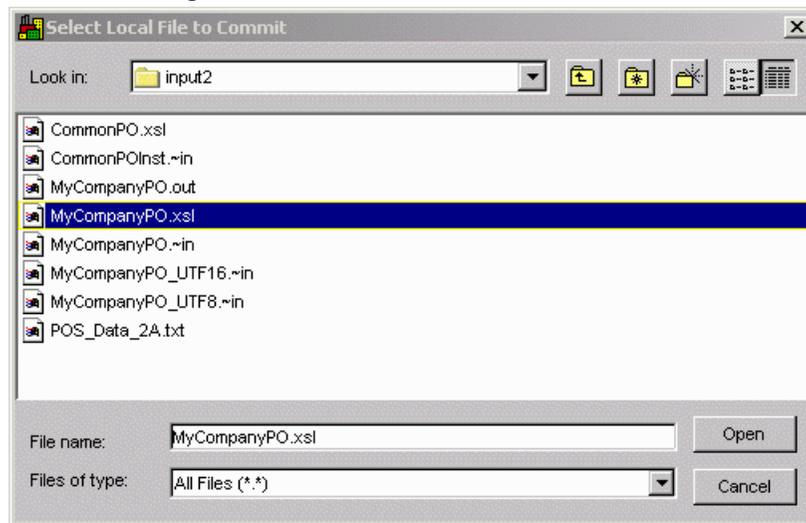
4   Select the **\monk_scripts\common** directory to make the committed **.xsl** file available to the XSLT Collaboration Service (see the following figure).

Figure 31   Commit to Sandbox Dialog Box

Note: *The .**xsl** file being used must reside in the **\monk_scripts\common** directory. If the .**xsl** file resides elsewhere, the XSLT Collaboration Service is not able to locate it.*

A message appears, confirming that the file is committed to the Sandbox (see **Figure 32 on page 101**).

Figure 32   File Committed
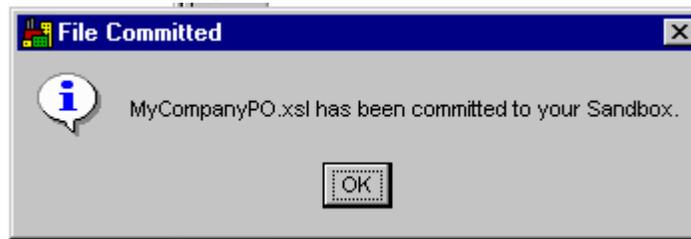
5   Click **OK** and return to the main Enterprise Manager window.

6   From the **File** menu, select **Promote to Run Time**. The Select File to Promote to Run Time dialog box appears (see the following figure).

Figure 33   Select File to Promote to Run Time Dialog Box
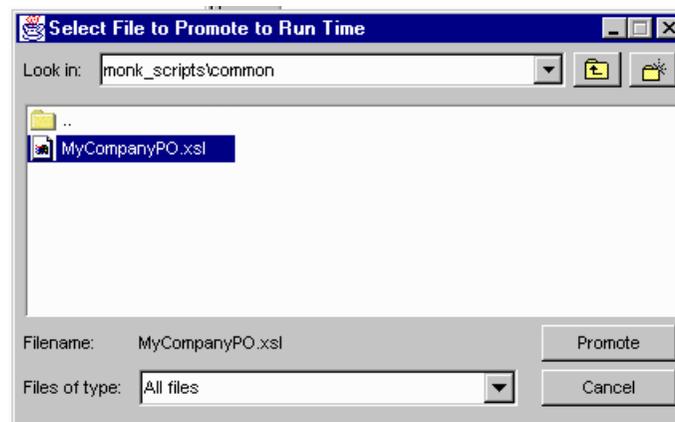
7   Select the **.xsl** file to be promoted and click **Promote**. The **File Promoted** message box appears.

Figure 34   File Promoted
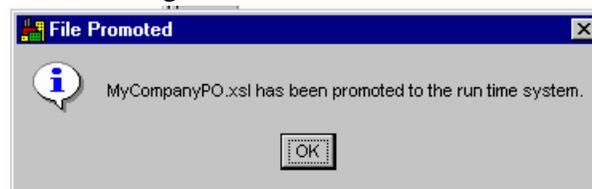
8   Click **OK** and return to the main Enterprise Manager screen.

10.2.2 Creating a Collaboration Rule

To create a Collaboration Rule

1   From the left side of the e*Gate Enterprise Manager Main window, select **Collaboration Rules**.

The list of Collaboration Rules appears on the right pane of the window.

2   Select the **Create New Collaboration Rules** icon in the middle of the window as shown in **Figure 35 on page 102**.

Figure 35   Enterprise Manager: Creating New Collaboration Rules
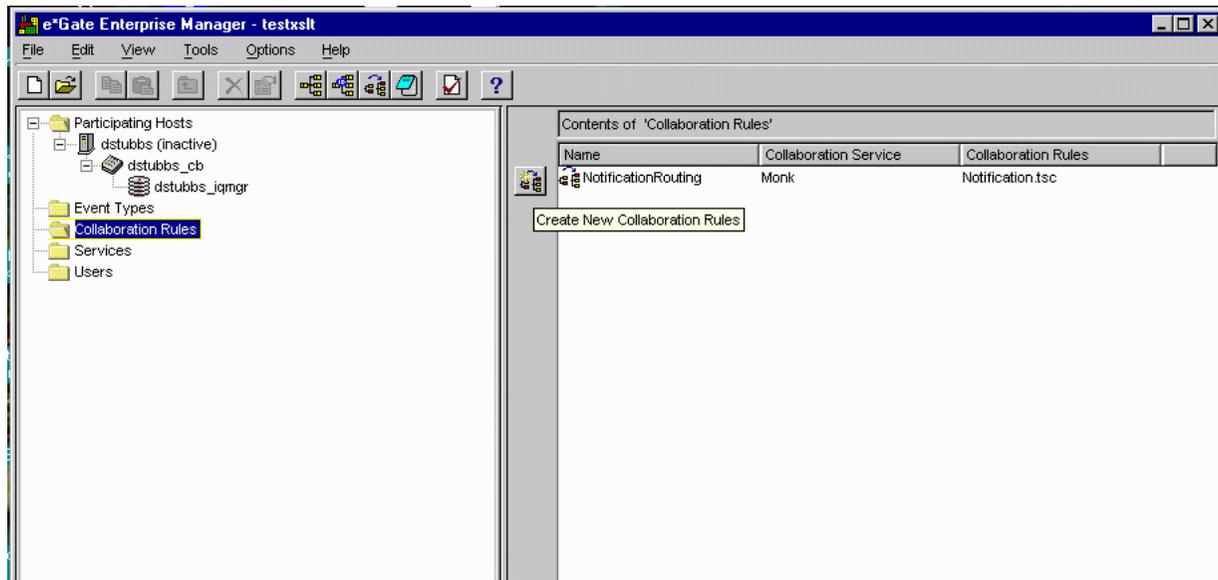
3   The New Collaboration Rules Component dialog box appears. Enter any name for the new rule (see the following figure) and select **OK**.

Figure 36   New Collaboration Rules Dialog Box
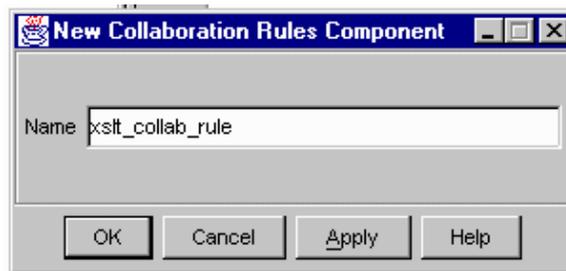
The newly created Collaboration Rule appears in the right pane of the Enterprise Manager window.

4   Double-click on the new Collaboration Rule. The Collaboration Rules Properties dialog box appears.

5   Under the **General** tab and under **Service**, select **XSLT** (see **Figure 37 on page 103**).

Figure 37 Collaboration Rules Properties
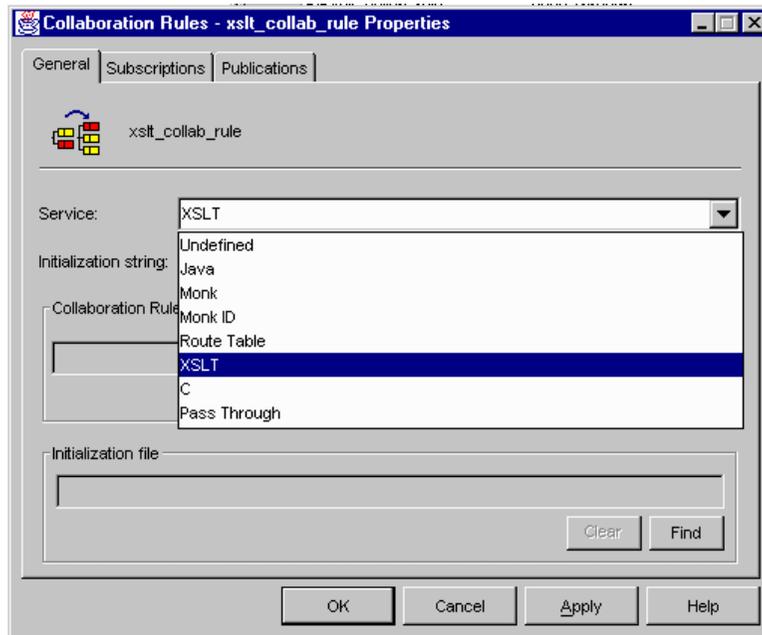
6 Input the **Initialization string** as follows:

```
-v true -r true -s false
```

The meaning of these parameters is shown in the following table.

Table 10 Initialization String Parameters

Parameter	Value	Description
-v	True or false. The default is true.	Validate on parse.
-r	True or false. The default is true.	Resolve externals references.
-s	True or false. The default is false.	Preserve white space.

7 Under **Collaboration Rules**, select the previously committed .**xsl** file (see **Figure 38 on page 104**).

Figure 38   File Selection
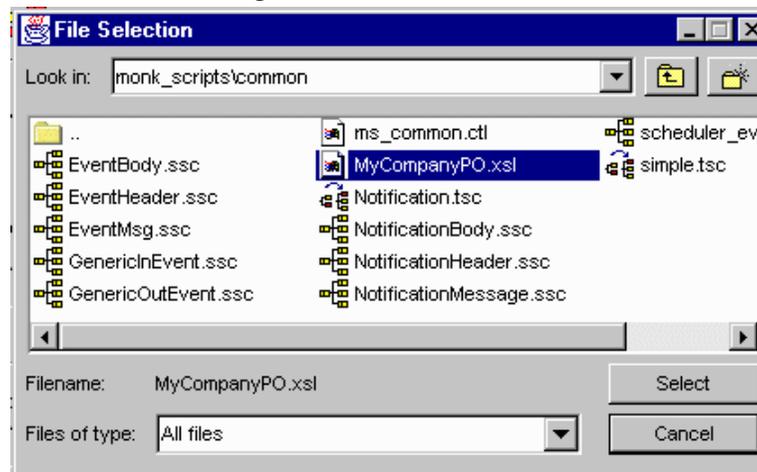
8 Configure the **Subscriptions** and **Publications** tab as you would for any other Collaboration Rule. See the *e*Gate Integrator User's Guide* for details.

10.3   Implementing the XSLT Collaboration Service

To implement the XSLT Collaboration Service

1 Define the Event Types which the XSLT Collaboration will subscribe to and publish.

2 Create the Collaboration Rules that use the XSLT Collaboration Service.

3 Configure a BOB or e*Way to execute this Collaboration.

4 Configure any other e*Gate components as necessary to create a working schema.

5 Test the schema, making any corrections as necessary to the e*Gate configuration or to any Collaboration Rules.

10.4   Sample Conversion

Here is an example consisting of three files:

XML Stylesheet - Input File

```
- <MyCompanyPO>
<POHeader Purpose="00" Type="NE" Number="00120033"
CreationDate="2000-06-22" />
<CarrierDetail Name="UPS" TransportType="G" Routing="ROUTING
INFORMATION" TransitTime="2330" />
- <BillTo>
<Address Name="Sam, Inc." Address1="1284 Main St." Address2="Bldg
111" City="Los Angeles" State="FL" PostalCode="91213" Country="USA" /
>
<ContactInfo ContactType="AM" ContactName="John Doe"
ContactNumber="33485" />
```

```
</BillTo>
- <ShipTo>
<Address Name="Sam, Inc." Address1="1284 Main St." Address2="Bldg
111" City="Los Angeles" State="CA" PostalCode="91213" Country="USA" /
>
<ContactInfo ContactType="BY" ContactName="Ken Smith"
ContactNumber="33485" />
</ShipTo>
- <Item>
<ItemHeader LineNumber="1" Quantity="5" Price="399.99"
UnitOfMeasure="EA" BuyerPart="1001428" VendorPart="AE3348"
UPC="145421" />
<ItemDescription Type="F" Description="TV" />
</Item>
- <Item>
<ItemHeader LineNumber="2" Quantity="1" Price="33.66"
UnitOfMeasure="EA" BuyerPart="1001563" VendorPart="AE3342"
UPC="134684" />
<ItemDescription Type="F" Description="Antenna" />
</Item>
</MyCompanyPO>
```

XSL File - Translation File

```
- <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:msxsl="urn:schemas-microsoft-com:xslt" xmlns:var="urn:var"
xmlns:user="urn:user" version="1.0">
<xsl:output method="xml" indent="yes" omit-xml-declaration="yes" />
- <xsl:template match="/">
<xsl:apply-templates select="MyCompanyPO" />
</xsl:template>
- <xsl:template match="MyCompanyPO">
- <MyCompanyPO>
- <xsl:for-each select="Item">
- <Item>
- <xsl:for-each select="ancestor::*[1]/BillTo">
- <xsl:for-each select="Address">
- <Address>
- <!--
 Connection from Source Node "Name" to Destination Node "Name"
-->
- <xsl:attribute name="Name">
<xsl:value-of select="@Name" />
</xsl:attribute>
</Address>
</xsl:for-each>
</xsl:for-each>
- <xsl:for-each select="ItemHeader">
- <ItemHeader>
- <!--
 Connection from Source Node "Price" to Destination Node "Price"
-->
- <xsl:attribute name="Price">
<xsl:value-of select="@Price" />
</xsl:attribute>
</ItemHeader>
</xsl:for-each>
</Item>
</xsl:for-each>
</MyCompanyPO>
</xsl:template>
</xsl:stylesheet>
```

Output File

```
<MyCompanyPO>
<Item>
<ItemHeader VendorPart="AE3348" Price="399.99" />
<ItemHeader VendorPart="AE3342" Price="33.66" />
</Item>
</MyCompanyPO>
```

Monk DTD Generator

The Monk DTD Generator converts Event Type Definition (ETD) files to XML files with a **.dtd** extension. This chapter explains how this functionality works with the e*Gate Integrator.

11.1 Introduction

Due to the popularity of XML, e*Gate users want to have the ability to convert Monk ETDs (**.ssc** files) to XML files with a **.dtd** extension. The Monk DTD Generator provides this capability.

How the Monk DTD Generator Works

The Monk XML DTD Generator is accessed from the **Export to DTD** option from the **File** Menu of the Monk ETD Editor in the Enterprise Manager. This Editor is where you can select an existing e*Gate ETD file with an **.ssc** extension. The resulting **.dtd** file exactly maps to the XML structure, and can then be saved and opened using any external editor (for example, *Notepad*).

This following steps explain briefly how the XML DTD Generator is used to convert an ETD file with a **.ssc** extension into a XML file with a **.dtd** extension.

1 An existing ETD file, with the **.ssc** extension, which was committed to the runtime schema, is opened from the **Export to DTD** option from the **File** Menu of the Monk ETD Editor.

2 The Generator takes the **.ssc** file and converts it to an XML file with a **.dtd** extension.

3 The newly created DTD file maps exactly to the XML structure.

11.2 Implementation

This section explains how to implement the DTD Generator and including how to access the feature in the graphical user interface (GUI).

11.2.1 Using the XML DTD Generator

To promote files to the run-time environment

To use the Monk XML DTD Generator, you must start with an **.ssc** file that has been promoted to the runtime environment.

Note: *See the **e*Gate Integrator User's Guide** for information on committing files to the Sandbox and promoting files to the Registry.*

You can use any of the following tools to promote files from the Sandbox to the run-time environment (see the following table).

Table 11   Tools To Promote Files from Sandbox

Tool to promote files from the Sandbox to run-time	Where to find more information
Monk ETD Editor	ETD Editor's online Help system and *e*Gate User's Guide*
e*Gate Enterprise Manager	e*Gate Enterprise Manager's online Help system and *e*Gate User's Guide*
stcregutil.exe (command-line utility)	*e*Gate Integrator System Administration and Operations Guide*

The first time you open an existing file in an e*Gate editor, e*Gate copies the file to your Sandbox. Once you save the file, it remains in your Sandbox until you either promote it to the run-time schema or manually remove it from your Sandbox.

When you open an existing file (in this case, an **.ssc** file) in an e*Gate editor, the file you request simply appears on screen (assuming no advisory locks are placed on that file by another user).

11.2.2 Creating DTDs Using the Monk DTD Generator

The Monk DTD Generator is accessed from the **File** Menu, **Export to DTD** option in the ETD Editor.

To access the XML DTD Generator

1   From the e*Gate Enterprise Manager screen, click the **ETD Editor** button to launch the ETD Editor.

2   From the ETD Editor's Toolbar, select the **Open** button.

The Open Event Type Definition dialog box appears (see the following figure).

Figure 39 Open Event Type Definition Dialog Box
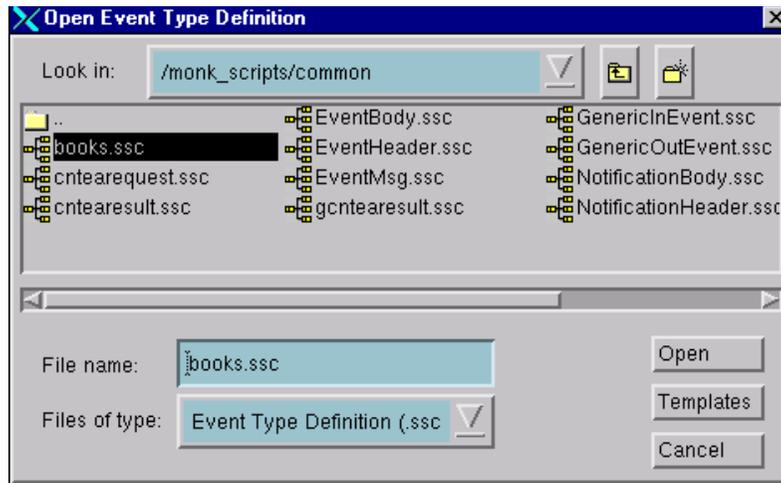
3 In the **File name** box, type or select the name of the .**ssc** file you want to use to generate the XML file with a .**dtd** extension. Look in the following location for your committed .**ssc** files:

> **d:egate\client\monk_scripts\common**

This is where the Enterprise Manager commits your files.

Note: ***Do not specify any file extension.*** *The ETD Editor supplies an .**ssc** extension for you.*

4 Click **Open**.

5 The selected .**ssc** file opens in the ETD Editor (see the following figure).

Figure 40 ETD Editor Window
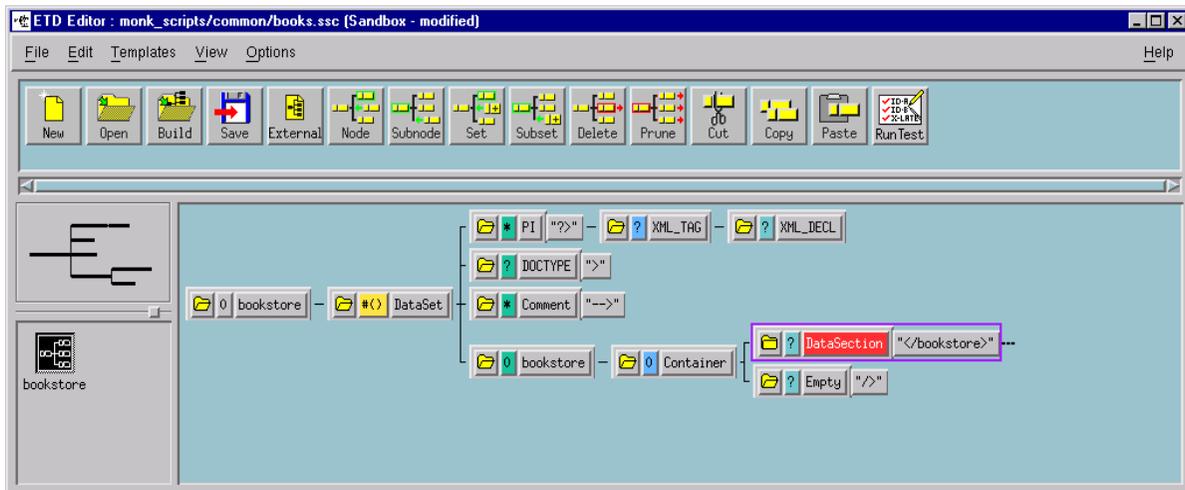
6 From the **File** Menu, select the **Export to DTD** option.

7 The Save as DTD dialog box appears (see the following figure).

Figure 41   Save as DTD Dialog Box
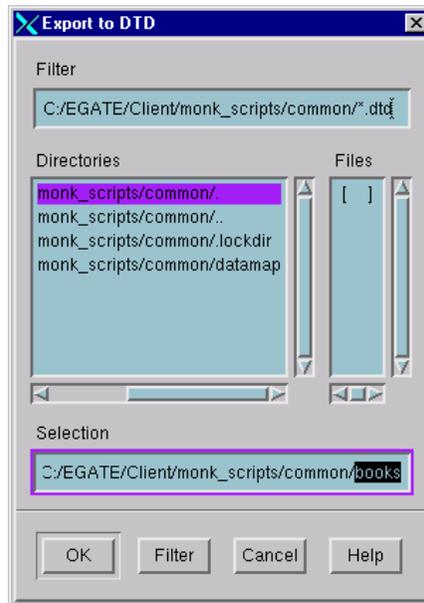
8 In the **Selection** box, type the name of the ETD input file to use to create the DTD file.

9 Click **OK**.

10 The DTD Exported dialog box appears to confirm that the XML DTD Generator successfully created the XML file with the **.dtd** extension (see the following figure).

Figure 42   DTD Export Dialog Box
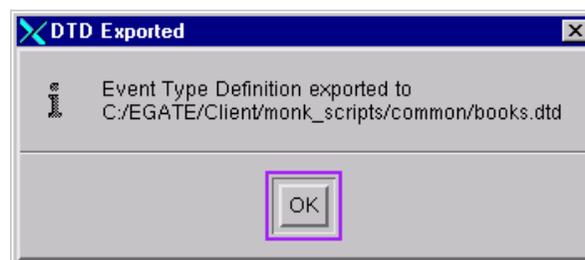
11 Click **OK** to close the dialog box.

12 From the **File** menu of the ETD Editor, select the **Save and Edit Using any External Editor** option to view the DTD file.

$event->xml Monk Function

This chapter explains the **$event->xml** Monk function that allows the easy generation of XML messages.

12.1 Introduction

Because of the wide availability of object-oriented tools that support XML, the **$event->xml** Monk function gives the user the ability to create an XML message dynamically in the course of a Collaboration.

In defining the business logic, it may happen that a sales order processed by e*Gate requires business logic that is encoded in a separate application. Often this interaction is accomplished synchronously, with object-oriented methods such as DCOM, CORBA, or Enterprise Java Beans. While those standards provide for invocation of the methods, the parameter or document exchange is increasingly based on XML.

The **$event->xml** Monk function facilitates transforming from non-XML to XML structures quickly and easily at run time. This is the simplest way of delivering parsed data to any XML-enabled application that can accept a DTD.

12.2 How the $event->xml Monk Function Works

It takes a parsed representation of a non-XML event and generates an XML message as illustrated in the following diagram.

Note: *The **$event->xml** Monk function must be used in conjunction with the DTD Generator so that the XML-enabled application has a DTD that matches the result of this function.*

The following figure illustrates how the **$event->xml** Monk function operates.

Figure 43 Operation of $event->xml

Step 1   Step 2

| External Non-XML | → | e*Gate | → | XML-Enabled Application |
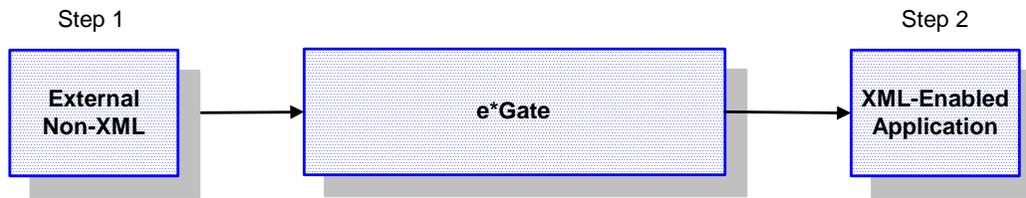
The two steps shown in the previous figure operate as follows:

1   In the first step, an external **.ssc** file generates a DTD using the DTD Generator.

2   In the second step, the DTD is given to the XML-enabled application.

The following page explains the basic properties of the **$event->xml** Monk function. After this explanation, a section that contains an example follows.

$event->xml

Syntax

```
($event->xml anEvent)
```

Description

$event->xml transforms some Event from a parsed e*Gate ETD structure into serialized XML data.

Parameters

Name	Type	Description
anEvent	message path	The Event to be transformed into an XML message.

Return Values

String
Returns the XML message.

Throws

None.

Additional Information

This function must be used in conjunction with the DTD Generator.

12.3 **$event->xml Example**

This section contains an example of the use of this Monk function.

test.monk

This is the Monk program that is used to demonstrate the sample.

```
(load "test.ssc")
(load "event-xml.monk")

(define (TestSSCToXML)
  (let ( (input ($make-event-map Purchase_Order-delm Purchase_Order-
struct))
       (output ""))

    (display "\n\n\nThe following shows how data mapped to event type
definition:\n")

    ($event-parse input
"PO1234*John Doe*3/1/00*2%/10 Net 30*
I234~Apple~12~.3~
I456~Orange~24~.4~")

    (newline)
    (display-event-dump input)

    (display "\n\nThe following shows the output of the $event->xml
function:\n\n")
    (define outputtest ($event->xml ~input%0))

    (display outputtest)

  ); end of let
)   ; end of TestSSCToXML

(TestSSCToXML)
```

mysscmsg.ssc

This is the ETD structure that maps the data.

```
;:- STC MsgStruct Version 3.1

;:- MsgStructure Header
;:- MsgStructure "Purchase_Order"
;:- UserComment "    "
;:- Version "e*Gate 4.1.1"
;:- FormatOption DELIMITED
;:- RepSeparator "Repetition Delimiter  " " "
;:- Escape "Escape Character Delimiter   " ""
;:- DefaultDelimiters "X12"
;:- End MsgStructure Header

;:- Delimiter Structure
(define Purchase_Order-delm '(
 ("~" separator)
 ("*" separator)
 (">" separator)
 ))

;:- Global Template Reference
;:- End Global Template Reference
```

```
                ;:- Local Template Definition
                ;:- End Local Template Definition

                ;:- MsgStructure Definition
                (define Purchase_Order-struct ($resolve-event-definition (quote
                  (Purchase_Order ON 1 1 und und und -1
                    (((Ed "*") ) Purchase_Order_Num ON 1 1 und und und -1);:= {0.0:N}
                    (((Ed "*") ) Purchaser ON 1 1 und und und -1);:= {0.1:N}
                    (((Ed "*") ) Date ON 1 1 und und und -1);:= {0.2:N}
                    (((Ed "*") ) Terms ON 1 1 und und und -1);:= {0.3:N}
                    (Items OS 1 INF und und und -1
                      (((Sc " \t\r\n")) Item_Num ON 1 1 und und und -1);:= {0.4.0:N}
                      (Descr ON 1 1 und und und -1);:= {0.4.1:N}
                      (Quantity ON 1 1 und und und -1);:= {0.4.2:N}
                      (Price ON 1 1 und und und -1);:= {0.4.3:N}
                    );:= {0.4:N}
                  ) ;:= {0:N}
                )))
                ;:- End MsgStructure Definition
```

log.txt

This is the output of the example. The actual output of the function can be seen at the end of this log.

```
stctrans (Information): *** Trace Mask Changed From 0x00000000-
0x00000000 To 0x80000000-0x00000002 - log file is off ***


The following shows how data mapped to event type definition:


((Modifiers):Name:Type:MinRep:MaxRep:Tag:Def:Offset:(Length|Encoding)
:Delim:BitFlags)
                  (():Purchase_Order:ON:1:1:::-1:(-1)::Su,Dc)
(Depth:Length:Children:FLAGS(Rep,Data,Arrayified,Constant,ChildData,S
ibData))
(0:73:1:DACB):PO1234*John Doe*3/1/00*2%/10 N
                  (():Purchase_Order:ON:1:1:::-1:(-1)::Su,Dc)
(1:73:5:RDACB) :PO1234*John Doe*3/1/00*2%/10 N
                  (():Purchase_Order_Num:ON:1:1:::-1:(-1):"*":RNu,Loc)
(2:6:1:DACB)   :PO1234
                  (():Purchase_Order_Num:ON:1:1:::-1:(-
1):"*":RNu,Loc)
(3:6:0:RDAC)   :PO1234
                  (():Purchaser:ON:1:1:::-1:(-1):"*":RNu,Loc)
(2:8:1:DACB)   :John Doe
                  (():Purchaser:ON:1:1:::-1:(-1):"*":RNu,Loc)
(3:8:0:RDAC)    :John Doe
                  (():Date:ON:1:1:::-1:(-1):"*":RNu,Loc)
(2:6:1:DACB)   :3/1/00
                  (():Date:ON:1:1:::-1:(-1):"*":RNu,Loc)
(3:6:0:RDAC)   :3/1/00
                  (():Terms:ON:1:1:::-1:(-1):"*":RNu,Loc)
(2:12:1:DACB)  :2%/10 Net 30
                  (():Terms:ON:1:1:::-1:(-1):"*":RNu,Loc)
(3:12:0:RDAC)   :2%/10 Net 30
                  (():Items:OS:1:4294967295:::-1:(-1):"~":Su,Dc)
(2:37:2:ACB)  :
I234~Apple~12~.3~
I456~Orange
                  (():Items:OS:1:4294967295:::-1:(-1):"~":Su,Dc)
(3:18:4:RACB)   :
```

```
I234~Apple~12~.3~
                        (((Sc "


")):Item_Num:ON:1:1:::-1:(-1):"~":RNu)
(4:4:1:ACB)     :I234
                        (((Sc "


")):Item_Num:ON:1:1:::-1:(-1):"~":RNu)
(5:4:0:RDAC)      :I234
                        (()):Descr:ON:1:1:::-1:(-1):"~":RNu)
(4:5:1:DACB)     :Apple
                        (()):Descr:ON:1:1:::-1:(-1):"~":RNu)
(5:5:0:RDAC)      :Apple
                        (()):Quantity:ON:1:1:::-1:(-1):"~":RNu)
(4:2:1:DACB)     :12
                        (()):Quantity:ON:1:1:::-1:(-1):"~":RNu)
(5:2:0:RDAC)      :12
                        (()):Price:ON:1:1:::-1:(-1):"~":RNu)
(4:2:1:DACB)     :.3
                        (()):Price:ON:1:1:::-1:(-1):"~":RNu)
(5:2:0:RDAC)      :.3
                       (()):Items:OS:1:4294967295:::-1:(-1):"~":Su,Dc)
(3:19:4:RACB)   :
I456~Orange~24~.4~
                        (((Sc "


")):Item_Num:ON:1:1:::-1:(-1):"~":RNu)
(4:4:1:ACB)     :I456
                        (((Sc "


")):Item_Num:ON:1:1:::-1:(-1):"~":RNu)
(5:4:0:RDAC)      :I456
                        (()):Descr:ON:1:1:::-1:(-1):"~":RNu)
(4:6:1:DACB)     :Orange
                        (()):Descr:ON:1:1:::-1:(-1):"~":RNu)
(5:6:0:RDAC)      :Orange
                        (()):Quantity:ON:1:1:::-1:(-1):"~":RNu)
(4:2:1:DACB)     :24
                        (()):Quantity:ON:1:1:::-1:(-1):"~":RNu)
(5:2:0:RDAC)      :24
                        (()):Price:ON:1:1:::-1:(-1):"~":RNu)
(4:2:1:DACB)     :.4
                        (()):Price:ON:1:1:::-1:(-1):"~":RNu)
(5:2:0:RDAC)      :.4
```

The following shows the output of the $event->xml function:

```
<?xml version="1.0"?><Purchase_Order><Purchase_Order_Num>PO1234</
Purchase_Order_Num><Purchaser>John Doe</Purchaser><Date>3/1/00</
Date><Terms>2%/10 Net 30</Terms><Items><Item_Num>I234</
Item_Num><Descr>Apple</Descr><Quantity>12</Quantity><Price>.3</
Price></Items><Items><Item_Num>I456</Item_Num><Descr>Orange</
Descr><Quantity>24</Quantity><Price>.4</Price></Items></
Purchase_Order>
```

Monk Capabilities and Troubleshooting

Capabilities of the Monk XML Toolkit are discussed in this chapter, along with some troubleshooting tips.

13.1 Capabilities

This section discusses the capabilities of the e*Gate Monk XML Toolkit components, the Monk DTD Converter and Generator.

13.1.1 Monk DTD Converter

The known limitations of the Monk DTD Converter are:

- **Only ASCII-encoded XML messages are currently supported.**
- **The Converter cannot handle self-describing XML events.**

  XML events are self-describing. The XML DTD Converter requires a DTD in order to generate an Event Type Definition that can be used in the Event Type Definition Editor and the Collaboration Editor. The ability to dynamically parse and read XML events is not the function of this Converter.

- **The Converter no longer defaults the first element in the DTD file as the root element.**

  The Convertor searches the file structure to see if any other root elements are present. If more than one element is present, it validates the first occurrence.

13.1.2 Monk DTD Generator

- The Monk DTD Generator tool can only be used with ETDs (**.ssc**) files in which every node and subnode name in the tree is unique.
- The node and subnode names of the **.ssc** file must follow the XML naming conventions.

13.2 Monk DTD Converter Troubleshooting

This section explains troubleshooting techniques available for the Monk DTD Converter.

Troubleshooting Question:

I ran the XML DTD Converter on a sample DTD file and failed to get the whole structure even though I specified the depth. Instead, the following error message appeared:

```
file:///d:/xmlconverter/sam/Web_Order.dtd: 23, 33: Element,
"ST_trans_set_header
", refers to undeclared element, "trans_set_id_code", in content
model
```

Answer:

An element was not declared in the DTD file called **trans_set_id_code**. To correct this problem, a line was added to define this element as **#PCDATA**.

Additional Troubleshooting

Before you can test the DTD Converter, you need to create the following Monk file:

```
(load "d:/egate/client/monk_scripts/common/books.ssc" )
(define input ($make-event-map bookstore-delm bookstore-struct))
($event-parse input input-string1)
```

Note: Please note that the path may change if the drive varies.

You can use any editor to create this text file. In this case, it's named **books.monk.**

If you want to test your own file, you need to change the following lines accordingly:

```
"d:/egate/client/monk_scripts/common/books.ssc"
"bookstore-delm (from the .ssc file)"
"bookstore-struct (from the .ssc file)"
```

After you have created the monk file, you can test the **.ssc** file by typing the following command from the Windows DOS prompt:

```
D:\eGate\client\monk_scripts\common>stctrans -d -md
-ims D:\XMLConverter\books\books.xml books.monk > log.txt
```

The output will be saved into a file called **log.txt**.

Here is the log for the example:

```
MAPPED:XML_DECL[0]:OF:0:1::"":-1:(0)-> " version='1.0'
encoding="ASCII" ".
MAPPED:XML_TAG[0]:OF:0:1:::-1:(0):"xml"--> " version='1.0'
encoding="ASCII" ".
MAPPED:PI[0]:ON:0:-1:::-1:(0):"<?"-"?>"-> "xml version='1.0'
encoding="ASCII" ".
MAPPED:PI[1]:ON:0:-1:::-1:(0):"<?"-"?>"-> "kill me".
MAPPED:DOCTYPE[0]:ON:0:1::" bookstore SYSTEM "file:///d:/
XMLConverter/books/books.dtd" ":-1:(0):"<!DOCTYPE"-">"-> "
bookstore SYSTEM "abcde" ".
MAPPED:EQUAL_SIGN[0]:ON:1:1:::-1:(0):"="-> " ".
MAPPED:Value[0]:ON:1:1:::-1:(0):"""-"""-> "ISBN1234567".
```

```
MAPPED:genre[0]:OF:1:1:::-1:(0):"genre"--> " = "ISBN1234567"".
MAPPED:EQUAL_SIGN[0]:ON:1:1:::-1:(0):"="-> "".
MAPPED:Value[0]:ON:1:1:::-1:(0):"""-"""-> "T".
MAPPED:used[0]:OF:1:1:::-1:(0):"used"--> "="T"".
MAPPED:AttSet[0]:AS:0:1:::-1:(0)-> " genre = "ISBN1234567"
used="T"".
MAPPED:Data[0]:OF:0:-1:"\[^<\]\*"::-1:(0)-> "STC Document
Department
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
MAPPED:DataSection[0]:ON:0:1:::-1:(0):">"-"</name>"-> "
MAPPED:Container[0]:OF:1:1:::-1:(0)-> ">
MAPPED:name[0]:OF:0:1:::-1:(0):"<name"--> ">
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "".
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "".
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
MAPPED:DataSection[0]:ON:0:1:::-1:(0):">"-"</author>"-> "
MAPPED:Container[0]:OF:1:1:::-1:(0)-> ">
MAPPED:author[0]:OF:0:-1:::-1:(0):"<author"--> ">
MAPPED:Data[0]:OF:0:-1:"\[^<\]\*"::-1:(0)-> "Monk Programmer's
Reference Guide
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
MAPPED:DataSection[0]:ON:0:1:::-1:(0):">"-"</title>"-> "
MAPPED:Container[0]:OF:1:1:::-1:(0)-> ">
MAPPED:title[0]:OF:0:1:::-1:(0):"<title"--> ">
MAPPED:Data[0]:OF:0:-1:"\[^<\]\*"::-1:(0)-> "$200.00
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
MAPPED:DataSection[0]:ON:0:1:::-1:(0):">"-"</price>"-> "
MAPPED:Container[0]:OF:1:1:::-1:(0)-> ">
MAPPED:price[0]:OF:0:1:::-1:(0):"<price"--> ">
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
MAPPED:DataSection[0]:ON:0:1:::-1:(0):">"-"</book>"-> "
MAPPED:Container[0]:OF:1:1:::-1:(0)-> " >
MAPPED:book[0]:OF:0:-1:::-1:(0):"<book"--> " genre = "ISBN1234567"
used="T" >
MAPPED:EQUAL_SIGN[0]:ON:1:1:::-1:(0):"="-> "".
MAPPED:Value[0]:ON:1:1:::-1:(0):"""-"""-> "F".
MAPPED:used[0]:OF:1:1:::-1:(0):"used"--> "="F"".
MAPPED:EQUAL_SIGN[0]:ON:1:1:::-1:(0):"="-> "".
MAPPED:Value[0]:ON:1:1:::-1:(0):"""-"""-> "ISBN7654321".
MAPPED:genre[0]:OF:1:1:::-1:(0):"genre"--> "="ISBN7654321"".
MAPPED:AttSet[0]:AS:0:1:::-1:(0)-> " used="F"
genre="ISBN7654321"".
MAPPED:Data[0]:OF:0:-1:"\[^<\]\*"::-1:(0)-> "Stc
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
MAPPED:DataSection[0]:ON:0:1:::-1:(0):">"-"</first-name>"-> "
MAPPED:Container[0]:OF:1:1:::-1:(0)-> ">
MAPPED:first-name[0]:OF:0:-1:::-1:(0):"<first-name"--> ">
MAPPED:Data[0]:OF:0:-1:"\[^<\]\*"::-1:(0)-> "Writer #1
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
MAPPED:DataSection[0]:ON:0:1:::-1:(0):">"-"</last-name>"-> "
MAPPED:Container[0]:OF:1:1:::-1:(0)-> ">
MAPPED:last-name[0]:OF:0:-1:::-1:(0):"<last-name"--> ">
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
MAPPED:DataSection[0]:ON:0:1:::-1:(0):">"-"</author>"-> "
MAPPED:Container[0]:OF:1:1:::-1:(0)-> ">
MAPPED:author[0]:OF:0:-1:::-1:(0):"<author"--> ">
MAPPED:Data[0]:OF:0:-1:"\[^<\]\*"::-1:(0)-> "$100.00
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
MAPPED:DataSection[0]:ON:0:1:::-1:(0):">"-"</price>"-> "
MAPPED:Container[0]:OF:1:1:::-1:(0)-> ">
MAPPED:price[0]:OF:0:1:::-1:(0):"<price"--> ">
MAPPED:Data[0]:OF:0:-1:"\[^<\]\*"::-1:(0)-> "Editor User's Guide
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
```

```
MAPPED:DataSection[0]:ON:0:1:::-1:(0):">"-"</title>"-> "
MAPPED:Container[0]:OF:1:1:::-1:(0)-> ">
MAPPED:title[0]:OF:0:1:::-1:(0):"<title"--> ">
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
MAPPED:DataSection[0]:ON:0:1:::-1:(0):">"-"</book>"-> "
MAPPED:Container[0]:OF:1:1:::-1:(0)-> ">
MAPPED:book[1]:OF:0:-1:::-1:(0):"<book"--> " used="F"
genre="ISBN7654321">
MAPPED:DataSet[0]:AS:0:1:::-1:(0)-> "
MAPPED:DataSection[0]:ON:0:1:::-1:(0):">"-"</bookstore>"-> "
MAPPED:Container[0]:OF:1:1:::-1:(0)-> ">
MAPPED:bookstore[0]:OF:1:1:::-1:(0):"<bookstore"--> ">
MAPPED:DataSet[0]:AS:1:1:::-1:(0)-> "<?xml version='1.0'
encoding="ASCII" ?>
MAPPED:bookstore[0]:OF:1:1:::-1:(0):-> "<?xml version='1.0'
encoding="ASCII" ?>
```

Index

Symbols

$event->xml **111**
$event->xml Sample **114**
.dtd
    Document Type Definition file **17**, **20**
.ssc
    Event Type Definition file (Monk) **17**
.xdr **86**
.xsc
    Event Type Definition file (Java) **17**, **20**
.xsd **12**, **86**
    XML Schema file **17**
    XML schema file **20**

A

abb **72**, **90**
access XML Schema **89**
accessing the DTD Builder **23**
accessing the XML Schema Builder **31**
Additional Command Line Arguments **90**
Attribute **76**, **94**
Attribute Set **74**, **91**
attribute types mapping, DTD **28**

B

basic functions
    event-send-to-egate **113**
Build an Event Type Definition **71**, **89**, **109**
Build tool **71**, **89**, **108**

C

CDATA **74**, **91**
Command Line Arguments **72**, **90**
Comment **74**, **91**
Committing the .xsl File **99**
Considerations **117**
Container **74**, **91**
Content **77**, **94**
conventions, writing in document **13**
Converting ETDs to DTDs **12**, **86**, **107**
Creating an XML message dynamically **111**

Creating DTD's **108**
Creating XSLT Collaboration Rules **99**

D

Data **74**, **91**
Data Section **74**, **91**
Data Set **74**, **91**
Delimiters **77**, **94**
depth **73**, 90, **90**
document purpose and scope **11**
Document Type Definition (DTD) **16**
DTD Builder, description **20**
DTD examples **47**
DTD overview **16**

E

element declarations mapping, DTD **27**
Element with Sub-elements **76**, **93**
Element without Sub-elements **75**, **92**
Empty **74**, **91**
Equal Sign **74**, **91**
ETD Structure **73**, **91**
ETD structure **92**
event-send-to-egate **113**
expanded **73**, **91**
Extensible Markup Language (XML) **16**
Extensible Stylesheet Language **98**

F

facilitator nodes **73**, **91**
Frequently Asked Questions **118**
functions
    event-send-to-egate **113**

G

General **77**, **94**
generated classes, XML schema **35**

H

HTML **16**

I

identifier mapping, DTD **21**
Implementing the XSLT Collaboration Service **104**
installation
    Windows **18**
Installation Procedure **19**

installation procedure
    NT **19**
intended audience, document **11**
Introduction **16**, **20**

J

Java **20**
Java mapping
    DTD Builder **21**, **27**
    XML Schema Builder **35**
Java packages, XML schema **35**

L

Library Converter **72**, **90**
log.txt **115**

M

Mapping **78**
Mapping for Attributes **79**
Mapping for Elements **78**
Mapping for Occurrence **80**
Mapping for Sub-elements **79**
mapping of complexType data types, XML schema **35**
mapping of elements, XML schema **45**
mapping of simpleType data types, XML schema (W3C 2000 specifications)
    additional Java mapping **42**
    standard Java mapping **41**
mapping of simpleType data types, XML schema (W3C 2001 specifications)
    additional Java mapping **38**
    standard Java mapping **36**
Monk function
    $event->xml **111**
MSXML3 Parser Technology Preview Release **98**
mysscmsg.ssc **114**

N

noattlist **73**, **91**
nocdata **73**
node
    child **92**
    parent **92**
Node Property Sheet **77**, **94**
nsattr **91**

O

Object-oriented tools **111**
organization of information, document **12**

P

Pre-installation **18**
property mapping, DTD **21**

R

Registry APIs for XML Schema Metadata
    overview **57**
    package contents and setup **58**
    sample implementations **63**
    using the APIs **59**
Repetition **77**, **94**
root **73**, **91**
running the DTD Builder **24**
running the XML Schema Builder **32**

S

Sample Conversion **80**
SDATA **74**, **91**
SeeBeyond Web site
    additional information
        technical support **14**
stcregutil **99**, **108**
stcregutil.exe **99**
supported features
    XML Schema Builder **45**
supporting documents **14**
system requirements **17**

T

test.monk **114**
trans_set_id_code **118**
Transforming XML documents **98**
treedepth **73**
Troubleshooting **118**

U

Understanding the ETD Structure **73**, **91**
unsupported features
    XML Schema Builder **45**
using the DTD Builder
    accessing **23**
    running **24**
Using the ETD Editor **77**, **94**
Using the XML DTD Converter **70**, **89**, **108**