Commerce One XDK Pro™

Developer's Guide

Version 4.0

Corporate Headquarters
4440 Rosewood Drive
Pleasanton CA 94588

www.commerceone.com

XDK Pro Developer’s Guide, Version 4.0

January 2001

Preface

Purpose of this Book

The *XDK Pro Developer’s Guide* explains how to use the Commerce One XML Parser (CXP), how to use the SOX to Java compiler, and how to interface with CXP via SAX.

Who Should Read this Book

This document is intended for CommerceOne System administrators and customers.

What’s Included in this Book

The following information is included in this book.

Chapter 1 Introduction

Chapter 1 provides a brief overview of XDK Pro.

Chapter 2 What is SOX

Chapter 2 describes SOX and SOX features.

Chapter 3 Programming Models

Chapter 3 describes the programming interfaces used by application developers to build electronic commerce applications and services that manipulate XML documents.

Chapter 4 How to Use the Commerce One XML Parser (CXP)

Chapter 4 explains how to use the Commerce One XML Parser (CXP) and provides examples of using CXP.

Chapter 5 Interfacing with CXP via SAX

Chapter 5 describes how to use CXP with SAX and provides a SAX sample.

Chapter 6 How to Use the SOX to Java Compiler (X2J)

Chapter 6 describes the X2J options and provides examples of using X2J.

Chapter 7 From a Document to a Bean and Back (RoundTrip)

Chapter 7 describes the SimpleRoundTrip Application.

Chapter 8 Creating and Manipulating a SOX Bean

Chapter 8 explains how to create and manipulate a SOX bean.

Chapter 9 Document Framework

Chapter 9 describes the programmatic interfaces used by applications for handling and manipulating documents.

Related Information

In this book, the terms below are defined as:

Enter	Refers to typing letters or numbers on the computer keyboard. If upper or lower case is mandatory, this is stated. If it is not, then either may be used.
Press	Refers to pressing one of the special keys on the computer keyboard, such as Tab, Ctrl or Alt. If it is necessary to press-and-hold a special key followed by another key, this is stated.
Click	Refers to positioning the mouse pointer (or cursor) over a screen button image and clicking the left mouse button. If pressing the right mouse button is required, this is stated as right-click.

XDK Pro Technical Support

Please write the names and numbers for your installation’s technical support contact personnel below:

If you cannot resolve a problem using thsi manual, contact your technical support representative and ask him/her to contact Commerce One Technical Support by email at **xdk_support@commerceone.com**.

Contents

Preface v
Purpose of this Book v
Who Should Read this Book v
What’s Included in this Book v
Related Information vi
XDK Pro Technical Support vii

Contents.. ix

Chapter 1: Introduction... 1-1
In This Chapter 1-1
Recommended Reading Order 1-1
Other Useful Documents 1-2

Chapter 2: What is SOX?... 2-3
In This Chapter 2-3
SOX Definition 2-3
SOX Versus DTDs 2-3
Additional SOX Features 2-4
Definitions 2-5
What to Read Next 2-6

Chapter 3: Programming Models3-1
In This Chapter 3-1
Programming Model Definition 3-1
What to Read Next 3-3

Chapter 4: Using the Commerce One XML Parser.......................... 4-1
What is CXP? 4-1
Class Paths 4-3
CXP Options 4-3
How to Use Schema Paths and URNs 4-6

How to Use Catalogs 4-8
IExamples of Using CXP 4-10
Example of Using a Sample File with CXP 4-12
Interpreting Error Messages 4-14
General Validation Errors 4-14
Encoding Errors 4-14

Chapter 5: Interfacing with CXP via SAX 5-1

Simple Event API for XML (SAX) 5-1
CXP and SAX 5-1
Using CXP with SAX 5-3
SAX Sample 5-6
Examples 5-8
CXP and XT 5-8
Options 5-9
Examples 5-10

Chapter 6: How to Use the SOX to Java Compiler (X2J) 6-1

In This Chapter 6-1
What is X2J? 6-1
Class Paths 6-1
X2J Options 6-1
How URNs are Used in Code Generation 6-4
Examples of Using X2J 6-5
Examples of Using a Sample File with X2J 6-7
Troubleshooting 6-9

Chapter 7: The Simple RoundTrip Application 7-1

In This Chapter 7-1
The SimpleRoundTrip Application 7-4
An Example 7-5

Chapter 8: Creating and Manipulating a Sox Bean 8-1

In This Chapter 8-1
Example 8-1

Chapter 9: Document Framework 9-1
In This Chapter 9-1
Terminology 9-2
Document Representation Classes 9-5
    Representations 9-5
    DocumentObject 9-7
    DocumentStream 9-8
    DocumentBytes 9-10
Document Representation Factory and Schema Lookup 9-12
    Entity Manager 9-13
    External Source 9-13
    DocumentFactory 9-14
Document Interfaces 9-16
    Document 9-17
    Doclet 9-18
    Marshaller 9-19
    UnMarshaller 9-19
    Type 9-20
    Version 9-21
    Identity 9-22
Envelope 9-22
    Main Abstraction 9-22
    Properties 9-24
    URICatalog Document 9-27
    Attachments 9-27
    Client Side Usage 9-28
    Service Side Usage 9-30
Sending and Receiving Documents 9-33
    Interfaces 9-33
    Exceptions 9-33
    DocumentListener 9-35
    DocumentResponder 9-35
    DocumentServant 9-36
Document Wrappers 9-37
    Reply 9-37
    Forward 9-37

Chapter 1
Introduction

In This Chapter

This chapter provides a brief introduction to XDK Pro.

Recommended Reading Order

Depending upon your level of expertise with XML and XML Schemas, we recommend the following reading order.

- If you are not familiar with either XML or SOX (Schema for Object-Oriented XML), then you will need to do some background reading on XML before reading this document. See the next section titled, *Other Useful Documents,* for a pointer to documentation on XML.
- If you have XML experience, but do not know SOX, then you should start with Chapter 2, *What Is SOX*, and then read some of the other documents referred to in *Other Useful Documents*, before going on to the remaining chapters in this document.
- If you have XML and SOX experience, and would like to learn more about the SOX beans, then you can go directly to Chapter 3, *Programming Models*.
- If you have XML and SOX experience, and would like to develop applications that use a SOX Validation Parser, refer to Chapter 4, *How to User the Commerce One XML Parser.*

Other Useful Documents

Fro more information on XML, read the XML 1.0 Specification located at http://www.w3.org/TR/PR-xml.html.

We recommend that you read the document SOXTutorial.pdf, (found on this CD), to learn how to read and write SOX documents. Once you have read this document, you should be well equipped to start writing your own SOX documents.

For a more in-depth knowledge of the syntax and features of SOX, read SOXSpecification.pdf, (also included on this CD).

For more details on how to use SAX, go to:

- http://www.megginson.com/SAX/

  and

- http://www.megginson.com/SAX/SAX1/

These sites contain complete descriptions of SAX 1.0 and SAX 2.0.

For more information on XSLT, go to http://www.xml.com/pub/rg/XSLT.

For more information on XT and the canonical XML format, see James Clark’s web site at http://www.jclark.com

Chapter 2
What is SOX?

In This Chapter

This chapter describes SOX (Schema for Object-Oriented XML).

SOX Definition

A schema is a set of rules that defines the structure of a document.

A Document Type Definition (DTD), is a particular type of schema language that is used to define XML documents. Given a schema (or a DTD), you can create instances of XML documents that conform to that schema. You can use the validating parser to automatically check whether an XML instance document conforms to a schema.

SOX is an XML schema definition language developed by Commerce One to support the use of XML for electronic commerce. We developed SOX because we believe that DTDs are inadequate for the purposes of e-commerce. DTDs are not sufficient to meet the scalability, reliability, and extensibility requirements of a large, distributed, rapidly evolving electronic market place. Also, DTDs are generally considered quite difficult to use. SOX is an easy to use alternative to DTDs, that also supports the needs of any highly decentralized environment (for example the Internet).

SOX Versus DTDs

The main features of SOX that support usability and scalability in distributed, e-commerce environments are:

- SOX adds to XML the ability to define types for data. SOX supports a set of intrinsic data types and has the ability to support user-defined data types such as ranges of integers. For example, the text value of an element or the value of an attribute can be declared to be an integer, and a conforming SOX validator will check that constraint.

This enhances the safety and reliability of the applications that use the XML instance documents defined according to SOX schemas as opposed to DTDs.

- SOX enhances XML by providing the ability to extend previously defined element types via the use of namespaces and inheritance. In short, it adds object-oriented programming concepts to XML. Although DTDs support parameter entities, which can be used for reusability and extensibility, parameter entities are difficult to use; and they introduce significant risk with respect to the safety and reliability of applications that use the resulting XML document instances.
- SOX encourages reuse of elementtype and datatype definitions via namespaces. This means that the definitions in one schema can reuse the definitions in other schemas by importing these other schemas. Definitions that are generally useful can then be reused and extended any number of times by other schemas. This potentially decreases the amount of definitions needed, and promotes a higher degree of consistency in the way similar data is defined in widely different schemas. In contrast, DTDs do not have namespace support. This is a problem when DTDs are scaled to thousands of marketplaces around the world.

Additional SOX Features

Commerce applications must be able to define data types other than strings, for example prices, quantities and dates. From a programming point of view, it would be much easier to be able to treat these pieces of data as strongly typed values such as prices, quantities, dates and so on. The only data type DTDs support in element values is **string**. While it is possible to build specialized mechanisms to allow parties to tell each other the data type through the use of attributes, it is more descriptive and maintainable to describe the appropriate data types in a schema language. In addition it enables a greater amount of validation of the data, as well as facilitating more immediate error catching [XML99].

XML is becoming one of the backbones of electronic commerce. This means that schemas for document types will proliferate. In such a situation, writing a new schema is much less error prone and convenient if we can build upon previously defined element types, and not define everything from scratch. DTD mechanisms to support reuse across schemas are extremely labor intensive and error prone. Placing support for reuse via the explicit import of namespaces directly in the schema language, with further support in document instances, is safer from a programming point of view and it scales well to a distributed world.

Next, various groups might share a basic document type, but each group will need to customize that document type for their own purposes. Also, as the needs of a market place change, document types will need to be extended or changed. Unfortunately, changing document types implies that the applications that use these document types will have to be updated if they are not to break. To facilitate the de-coupling of changes to document types from the evolution of the applications that use these document types, SOX provides element type extension and versioning mechanisms. For example, an application using a basic purchase order will not break when it is provided with a purchase order that has been customized to handle the needs of the chemicals industry [XML99].

Definitions

The rest of the documentation uses the following terminology:

- **Schema** refers to a SOX schema document. A schema defines the element types and data types that can be found in an XML instance of that schema.
- An XML instance document is **valid** if it is well-formed and its contents and structure obey all the rules specified in the schema or DTD it claims to conform to. Generally a validating parser tests a document's validity.
- **SAX** refers to the Simple event API for XML.
- **XT** refers to James Clark's Java implementation of an XSLT processor.

What to Read Next

Now that you have some information on what SOX is, you should get some knowledge about how to use it.

We recommend that you read the document SOXTutorial.pdf, (included on this CD), for a tutorial on reading and writing SOX documents. The tutorial enables you to start writing your own SOX documents.

The sample SOX schemas and XML instances used in the SOXTutorial.pdf document are included in this installation. They are located underneath the sample directory.

For a more in-depth knowledge of the syntax and features of SOX, refer to the document SOXSpecification.pdf, included on this CD.

Chapter 3
Programming Models

In This Chapter

This chapter describes the X2J programming model including the syntax, semantics, and failure semantics.

Programming Model Definition

The term *programming model* refers to the programming interfaces that application developers use to build electronic commerce applications and services that manipulate XML documents. A programming model presents an abstract yet well-defined interface to which applications can be built. In our case, a programming model defines the syntax, semantics, and failure semantics of accessing various parts of a document.

- **Syntax** - defines exactly how a particular schema is mapped to classes and operations in a specific programming language.
- **Semantics** - define the behavior of the system when calls and operations are performed.
- **Failure semantics** - define the conditions under which failures can occur, how they are presented to the application, and the actions an application can take to address a failure.

The characteristics of an application are strongly influenced by the type of programming model that is used. The choice of programming model influences how easy or difficult it is to change an application as new document types are introduced. In some cases we need programming models that allow data to be discovered dynamically. In other cases it is more important to easily extract and insert data of the correct type out of the programmatic representation of a document and thereby ensure a higher-level of safety.

In the XML arena, applications are currently developed using the Document Object Model (DOM) and the Simple Event API for XML (SAX) programming models. The Commerce One XDK Professional software package introduces a new, typed, Java Beans based programming model for XML called X2J. The model enables the development of safer, more robust, but still flexible XML based applications. The X2J programming model is enabled by the SOX XML Schema language discussed in the previous chapter. It is designed to take advantage of Java's object-oriented features to enable extensibility and evolvability for XML based applications.

X2J represents XML schema documents as SOX Beans that are typed Java Bean classes, and XML instance documents as instantiations of these classes. X2J was designed from the perspective of e-commerce, and therefore tries to fulfil the following criteria, which we consider to be of high relevance for e-commerce applications:

- Safe and static checking of how documents are used.
- Safe introduction of extensions.
- Memory footprint and performance considerations.
- The ease of use and the support a programming model provides to developers.
- Flexibility and dynamic discovery of data.
- Maintenance and cost of ownership implications.

The typed X2J programming model consists of both a generic set of classes and interfaces, as well as classes and interfaces that are specific for a particular document type. Document type specific classes and interfaces expose data as being of the type that was defined in the schema. If an attribute was defined as an integer in the schema it will be an integer in the programming model. The process of producing such a programming model requires an XML Schema that has type information.

What to Read Next

For a comparison between DOM, SAX, and X2J, and a more detailed overview of X2J, refer to Koistinen et al. [XML99], included in this package.

We recommend that you first read the document BeanTutorial.pdf (found in this package), for a tutorial on reading and using the SOX Beans corresponding to SOX documents.

Next, proceed to Chapter 6, *How to Use the SOX to Java Compiler (X2J)*.

Chapter 4
Using the Commerce One XML Parser

What is CXP?

CXP is a validating XML 1.0 and SOX 2.0 Parser. You can use it to validate document instances against DTDs that are XML 1.0 compliant, or schemas that are SOX 2.0 compliant. You can also use it to validate schemas against the SOX 2.0 specification. CXP automatically recognizes whether a document instance should be parsed as an XML 1.0 instance of a DTD, a SOX 2.0 schema document, or an XML 1.0 instance of a SOX 2.0 schema. This is determined from the first few lines of the document.

Note that a SOX schema is, in fact, a valid XML 1.0 document itself, conforming to a DTD called `schema.dtd`. All SOX schemas must conform to this DTD to be valid SOX. If a SOX schema is used as input, CXP will validate the schema against its DTD in addition to doing a SOX-level validation of the schema document. The SOX validation is done to check for those SOX restraints that cannot be checked by merely validating against the DTD.

The following three cases are the use cases for CXP. They are distinguished from each other by the first lines in the document that is parsed.

Validating an XML Instance against a DTD

If you pass CXP a document that starts with the header below, it is validated as an XML 1.0 document against the DTD specified in `example.dtd`. The keyword `tree` in the `DOCTYPE` declaration is the name of the root element of the instance. The name of the root element and the URL of the DTD may vary from one document to another.

```
<?XML version="1.0">
<!DOCTYPE tree SYSTEM
"urn:x-commerceone:document:com:mycompany:xml:example.dtd$1.0">
```

Validating a SOX Schema

If you pass CXP a SOX schema that starts with the header below, it is validated in two ways. First it will be validated as a valid XML 1.0 document against the DTD specified in `schema.dtd`. Second, it is validated as a SOX 2.0 schema to ensure that it conforms to the SOX restrictions that the DTD cannot check. The parser determines that the docuemnt is a SOX schema due to the name of the root element being "schema" and the URL being that of schema.dtd. Therefore all SOX schemas will start with this exact doctype tag.

```
<?XML version="1.0">
<!DOCTYPE schema SYSTEM
"urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">
```

Validating an XML Instance against a SOX Schema
If you pass CXP an XML 1.0 instance that starts with the header below, it is validated against the schema CBL.sox as an instance of the SOX 2.0 schema specified in `CBL.sox`. The schema URL may vary from one document to another.

```
<?soxtype urn:x-commerceone:document:com:commerceone:CBL:CBL.sox$1.0?>
```

Class Paths

It is fine to not have a classpath system variable set when running CXP. The provided scripts for running CXP sets the classpath system variable appropriately.

CXP Options

This section assumes that you have installed XDK in the default location. You should run CXP with the provided script. On Windows, this script is named `cxp.bat` and is located in `c:\commerceone\xdkpro`. On Solaris this script is named `cxp.sh` and is located in `/opt/CMRCxdkp`.

Running CXP with no options presents usage information on the screen:

```
cxp.INFO: Usage is
"cxp [-p schema path][-c catalog-system-id]
[-o output file][-n parse n times][-enc encoding]
[-novalid no validation][-e parse entity][-t timing info]
[-v verbose output][-g canonical form]
[-help help screen] [-f accept files][Document(s): document list]"
```

The following table describes the CXP options

CXP Option	Description
-p <schema-path>	The schema path; a semicolon delimited list of paths to use as root for schema search. This is similar in notion to a class path. CXP needs to know the root path from which to resolve any URNs that the input document refers to. If the document is a SOX schema, or an XML instance conforming to a SOX schema, then the schema refers to `schema.dtd` and it is necessary to include a path that can resolve the URN of that file. In this installation, this path is the `c:\commerceone\xdkpro` directory on windows and `/opt/CMRCxdkp` on Solaris. This is the same directory that the CXP script is located in. This option is ignored when a catalog is specified (see the `-c` option). If neither this option, nor the -c option, is specified, CXP assumes that you are providing only relative filenames in your document. These filenames are relative to the current directory. Note that some command line applications require you to quote semi-colon separated paths. If you have not placed your schemas in the XDK installation, you also have to add a path to your schema.
-c <catalog-system-id>	The catalog-system-id causes CXP to load a URN catalog. A URN catalog maps schemas or DTDs specified in the input document to their physical locations on the file-system. When this option is supplied, the `-p <schema-path>` option is ignored. For more information, refer to the section on how to use catalogs.
-o <output file>	CXP sends the output to the file specified.
-n <number>	The number of times CXP parses each document. The value should be an integer. If the option is not used, the document will be parsed once. This option can be useful with the -t option to determine parse time.

CXP Option	Description
-enc <encoding>	Specifies the XML file encoding to use. Supported encodings are the same as the encodings supported by the particular JVM being used. Some possible values are "UTF8", "Unicode", "SJIS", "8859_1", "8859_15", "ASCII", "Big5", "GB2312", "KSC5601", "Cp874", "JIS". If this option is not used, the default encoding is UTF8. No matter what encoding you use, all the files encountered in one parse must use the same encoding. You cannot for example, parse an instance that uses one encoding, that in turn refers to a schema that uses another encoding.
-novalid	CXP does not validate the document against a DTD or a schema if this option is specified. This option does not take an argument. This option has no effect when validating a SOX schema, as a SOX schema is always validated.
-e	CXP parses an XML document as an external entity if this option is specified. This option does not take an argument.
-t	CXP outputs the amount of time it took to do the parse if this option is specified. This option does not take an argument.
-v	CXP outputs verbose messages during the parse if this option is specified. This option does not take an argument. With this option, CXP displays additional information related to instance validation while validating the document. This information is helpful in finding and fixing problems in your document.
-g	Outputs the canonical form of the input document if this option is specified. This option does not take an argument. See http://www.jclark.com for information on canonical formats.
-help	CXP shows detailed help information (similar to this) if this option is specified. This option does not take an argument.

CXP Option	Description
-f	The `-f` option forces CXP to accept regular file names for the input files. By default CXP assumes that the input document name provided is a URI. Since a URI will always contain at least one colon, ":", CXP can recognize that an input file that contains no colons must be a file name, and process the input as a file name instead of a URI. However, in those cases where you use a filename that contains a colon such as `c:\schemas\TestSchema.sox`, you can tell CXP to interpret the input as a file name by using the `-f` option. For example, in Windows, given the file po.xml, you can use either "`cxp \commerceone\po.xml`" or "`cxp -f c:\commerceone\po.xml`". This option does not take an argument.
Document(s)	The name of one single document, or several documents separated by spaces, that CXP should parse. This can be in either file or URI format, (see the `-f` option above).

How to Use Schema Paths and URNs

As discussed previously, an XML instance of a SOX 2.0 schema always starts with a soxtype declaration. The soxtype declaration contains the URI of the schema to which the instance document claims to conform. CXP is able to automatically locate a schema from a URI, as long as you follow a strict formula for the URI, in the shape of a Commerce One format URN. The same URN format is used for namespace imports.

In the following example, let us assume that the schema is contained in a file `sample.sox`. Since the Commerce One implementation uses the URI of the schema to determine the physical location of the schema, a strict formula has to be followed in constructing the URI if the schema is to be located and used by CXP:

1. The URI must always start with "`urn:x-commerceone:document:`". This part specifies that the Commerce One specific scheme for the mapping, and must be used verbatim in order for the mapping to function properly.

2. The part after the scheme states the directory hierarchy in which the file is located. In this example the remaining portion of the URI will be `sample:xdk:sox:sample.sox$1.0`. The portion of the

URI before sample.sox is a representation of a partial path to the file `sample.sox`, with the file separator replaced by colons. This path is followed by the name of the schema file, in this case `sample.sox`, and then the version token. The version token in this release must always be 1.0, as versions are not yet supported.

3. Determine what the root of your schema tree is on your file system. This is a location in the file hierarchy underneath which all your schemas are located. The root is the same as the argument to the -p option explained in the section titled *CXP Options* earlier in this chapter. The root is represented as (ROOT) in this example.

4. Exactly underneath the root, the path to the schema file in this example has to start with `"sample\xdk\sox\"`. Notice that the part of the path following the root is exactly the same as the URN fragment specified in step 3, up to the file name `sample.sox`, with the colons replaced by file separators.

5. Next, the version is reflected in the path to the schema by an extra directory level: `n1_0`. This directory is the last directory in the path, and the schema is located in this directory. The schema must be physically located in a directory representing the version. The version is modified before being used in the path, by adding an "n" before the version, and substituting the period, ".", with an underscore, "_". Version 1.0 therefore becomes `"n1_0"` in the physical path of the file. Thus, the file sample.sox is located in the directory `"(ROOT)\sample\xdk\sox\n1_0"`.

6. The complete physical path to the file represented by the URN `urn:x-commerceone:document:sample:xdk:sox:sample.sox$1.0` is therefore: `(ROOT)\sample\xdk\sox\n1_0\sample.sox`

To use the above-described URN mechanism to allow CXP to locate schemas you need to give CXP the `(ROOT)` directory. This directory is given to CXP as the value for the schemapath option `-p`.

How to Use Catalogs

If you do not want to use the URN mapping mechanism described in the previous section in your SOX schemas and XML document instances, you can instead use catalogs to locate the necessary schemas or DTDs. A catalog is a valid XML document that maps the URN of a schema or a DTD to a physical location on your file system. When a catalog file is provided, CXP will use this catalog to search for the schemas or DTDs specified in the input document. If no mapping is found between the URN and a physical file, CXP will display an error message saying that the SOX schema or DTD could not be loaded.

For example, if you want to create an XML instance of a SOX schema sample.xml and you don't want to use the URN mapping mechanism, you can use the following header instead:

```
<?soxtype urn:sample.sox?>
```

Note that CXP requires the mapped URI to start with "urn:".

In order for CXP to successfully locate the schema referred to in the `soxtype` declaration, you need to create a catalog file, for example `catalogs.xml`, that maps urn:sample.sox to a physical location. The following is a sample `catalogs.xml file`.

```
<?xml version="1.0"?>
<catalog>
    <map uri="urn:sample.sox" to="/sample/xdk/sox/n1_0"/>
</catalog>
```

The file names that you map to, must use forward slashes as file separators.

Note that the URI used in the catalog has to exactly match the one found in the soxtype declaration.

Also note that if the URN that you are mapping to in the catalog is a filename, it must not contain any colons. If any colons are present, then CXP will assume that the filename is a URI. If you need to have a colon in your filename, such as in the case of mapping to a file on a different drive, then the filename must start with "`file:///`" in order for it to be a valid URI on your local system.

Now you can parse `sample.xml` from your command line for NT as follows.

```
cxp -c catalogs.xml sample.xml
```

Parse `sample.xml` from your command line for Solaris, as follows.

```
./cxp.sh -c catalogs.xml sample.xml
```

CXP will now expect to find
`sample.sox` in `\sample\xdk\sox\n1_0`

Be sure to use an identical URI in both the XML instance and the SOX schema that it is an instance of. For example, the schema start tag of the `sample.sox` document must look like this:

```
<schema uri="urn:sample.sox">
```

Otherwise, even if you map the URI in the catalog file, you will still get an error message.

A catalog can also include other catalogs, by using the `include` element:

```
<?xml version="1.0"?>
<catalog>
    <include location="/commerceone/catalogs/other_catalog.xml">
</catalog>
```

This catalog now includes the `other_catalog.xml` catalog file. This enables multiple catalogs to be used in one parsing. In the case of a relative filename, the filename specified in the location attribute is relative to the location of the catalog in which it is included.

IExamples of Using CXP

Let’s assume that the sample document below is located at `c:\testdocs\langstring.xml` on windows and `/testdocs/langstring.xml` on Solaris. Let’s assume that its schema is located at `c:\mywork\com\mycompany\lang\n1_0\LangString.sox` on Windows, and `/mywork/com/mycompany/lang/n1_0/LangString.sox` on Solaris.

```
<?soxtype
"urn:x-commerceone:document:com:mycompany:lang:LangString.sox$1.0"?>
<LangString Lang="EN"></LangString>
```

Here are some examples of how to use some of the CXP command line options:

- **Example 1** (Windows):

```
cxp -p \mywork \testdocs\langstring.xml
```

- **Example 1** (Solaris)

```
./cxp.sh /mywork /testdocs/langstring.xml
```

This is the simplest way to parse the document on the command line. CXP will search for `com\mycompany\lang\n1_0\LangString.sox` starting at the `\mywork` directory.

- **Example 2** (Windows):

```
cxp -p \mywork -n 2 -t \testdocs\langstring.xml
```

- **Example 2** (Solaris)

```
./cxp.sh -p /mywork -n 2 -t /testdocs/lanstring.xml
```

CXP will parse this document twice and output the timing information on your screen:

```
5077

10

2543.0 / 10.0
```

This means that parsing the document took 5077 milliseconds in the first iteration and 10 milliseconds in the second iteration. The average time for an iteration was 2543 milliseconds. Excluding the first iteration, the average time for an iteration was 10 milliseconds. The reason the parse was so much faster on the second run is that the schema is cached on the first run and reused on the second.

- **Example 3** (Windows)::

```
cxp -p \mywork -novalid -g -o LangString.out
\testdocs\langstring.xml
```

- **Example 3** (Solaris):

```
./cxp.sh -p /mywork -novalid -g -o LangString.out
/testdocs/langstring.xml
```

CXP will bypass the validation against the SOX schema and only verify the well-formedness of the XML document, generate a canonical form, and save it to the file LangString.out.

- **Example 4** (Windows):

```
cxp -v -p \mywork \testdocs\LangString.xml
```

- **Example 4** (Solaris)

```
./cxp.sh -v -p /mywork /testdocs/LangString.xml
```

This will give you verbose information about the parse. Here is some sample output:

```
cxp.INFO:ns="urn:x-commerceone:document:com:mycompany:lang:
LangString.sox$1.0"ordinal="1"
cxp.INFO: Element: soxtype:1[LangString:1]
cxp.INFO: Attribute group: LangString:1[Lang?]
```

`cxp.INFO:ns="urn:x-c..." ordinal="1"` says that the parser has recognized a new namespace which it will refer to by number 1 from now on.

`cxp.INFO: Element: soxtype:1[LangString:1]` is the representation of a content model for the document itself. ": 1" indicates that the definition belongs to namespace 1. This content model says that the document can have a root tag of "LangString".

Attribute group: `LangString:1[Lang?]cxp.Info` says that the "LangString" element has an optional attribute "Lang".

Example of Using a Sample File with CXP

In this distribution we have included several sample SOX 2.0 schemas and XML instances of those schemas. The sample SOX schemas are located in (for Windows):

```
C:\commerceone\xdkpro\sample\xdk\sox\n1_0
```

or (for Solaris):

```
/opt/CMRCxdkp/sample/xdk/sox/n1_0
```

The sample instances of these schemas, are located in several directories under (for Windows):

```
C:\commerceone\xdkpro\sample\xdk\instances\
```

and under (for Solaris):

```
/opt/CMRCxdkp/sample/xdk/instances/
```

For example, corresponding to the SOX schema file (for Windows):

```
C:\commerceone\xdkpro\xdk\sample\xdk\sox\n1_0\Film.sox,
```

and for Solaris::

```
/opt/CMRCxdkp/sample/xdk/sox/n1_0/Film.sox
```

There is an XML instance document (for Windows):

```
C:\commerceone\xdkpro\sample\xdk\instances\basic\Film.xml
```

and for Solaris:

```
/opt/CMRCxdkp/sample/xdk/instances/basic/Film.xml
```

To validate `Film.xml` against the schema to which it conforms, go to the directory:`\commerceone\xdkpro`, and type the following on the command line (for Windows):

```
cxp -p \commerceone\xdkpro\ sample\xdk\instances\basic\Film.xml
```

For Unix, go to the directory:
`/opt/CMRCxdkp/xdk`, and type the following on thetype:

```
./cxp.sh -p /opt/CMRCxdkp
sample/xdk sample/xdk/instances/basic/Film.xml
```

Interpreting Error Messages

This section describes general validation errors and encoding errors.

General Validation Errors

Given the following error message:

```
<ERROR creator="Validation">file:///TEMP/langstring.xml:2:23: Value
specified for enum "LangCode" is not one of the legal enumerated
values: must be one of "AA, AB, AF, AM, AR, AN"</ERROR>
```

The `creator="Validation"` part designates which part of the parsing process is generating the message. "Validation" indicates that the parser is generating a validation error while parsing the actual document instance.

Other common creators are:

- `creator="AST"`: Errors generated from parsing the schema pertaining to the referential integrity of the schema. This tells the parser whether all the types used were actually defined properly.
- `creator="CXP Lexer"`: Low level errors generated by the XML lexer. These can occur in both schemas and documents and usually pertain to problems with IO, invalid encodings or general syntax errors.
- `creator="CXP Parser"`: These errors come from the XML parser section of CXP and relate to well-formedness of XML documents. See the XML 1.0 specification for a more thorough explanation of the difference between well-formed and valid.

The parser may also generate <WARNING>, <FATAL>, <CRITICAL>, <INFO>, or

<STATUS> messages with the same format. The `file:///TEMP/langstring.xml:2:23:` part of the message informs you that the error occurred at line 2 column 23 in the file langstring.xml.

Encoding Errors

When the parser reports an error such as:

```
Invalid character number....
```

or

```
problem with IO or possible invalid character
for current encoding: Missing byte-order mark
```

it usually means that the parser is using an inappropriate encoding. By default, CXP expects all XML files to be in UTF8 (which is backward compatible with 7-bit ASCII). If you want to process documents with a

different encoding, such as UTF16 or 8-bit Latin, you must use the `-enc <encoding>` option. If you want to parse a Unicode document, you need to make sure that your document has the appropriate byte-order mark for your system; otherwise CXP is not able to process it correctly.

Note .... All input files must use the same encoding. CXP cannot dynamically switch encoding schemes while processing.

Chapter 5
Interfacing with CXP via SAX

Simple Event API for XML (SAX)

SAX is a public interface that a developer can use to gain access to CXP. SAX is an event API, which means that the parser serializes the instance document into a series of events, each corresponding to some significant logical or physical element in the document. For example: `startDocument`, `endDocument`, `startElement`, `endElement`, and `characters` are some of the supported events.

A SAX application connects to an XML parser through a SAX driver. A driver is a class made available by the parser. It implements some or all of SAX APIs.

A SAX application handles the SAX events produced by the parser in the handlers defined in the SAX specification. A handler is a class that implements a SAX handler interface such a Document Handler, ErrorHandler, or DTDHandler.

James Clark's XSL processor XT is an example of such an application. XT is provided in the XDK.

There are two versions of SAX: 1.0 and 2.0.

- SAX 1.0 provides the functionality for instantiating the parser, parsing the instance documents and receiving basic document, error and DTD events.
- SAX 2.0 includes all the SAX 1.0 functionality, defines methods for configuring the parser, and provides more elaborate DTD events. A parser is free to implement any or none of SAX 2.0 features.

CXP and SAX

CXP implements most of SAX 1.0 and some of SAX 2.0. The SAX 1.0 features that it does not implement are:

- `Parser.setLocale()`
- `DTDHandler events`
- `EntityResolver events`

CXP implements the SAX 2.0 Configurable interface. This interface allows a user to set parser features and properties. The provided methods are `setFeature` and `getFeature`, and `setProperty` and `getProperty`.

Each of the set methods takes two parameters: the name of the feature/property (which is in the form of a URL), and the value of the feature/property. The names of all the features and properties are defined in the java interface com.commerceone.xdk.standards.sax.SAX20Strings which can be found in the XDK API java doc.

The feature that can be `set/get` in the CXP SAX implementation are:

- **Validation**. The name of this feature is "http://xml.org/sax/features/validation". This name is defined in SAX20Strings as the constant SAX20_VALIDATION_FEATURE. The associated value is of type boolean, and turns validation on or off for the CXP parser. A value of `true` turns validation on, a value of `false` turns validation off.

The properties that can be **set** are:

- **Schema path**. The name of this feature is "http://commerceone.com/sax/properties/schemapath". This name is defined in SAX20Strings as the constant COMMERCEONE_SCHEMAPATH_PROPERTY. The associated value is of type String. It sets the schema path the parser uses to locate schemas from URN's See the -p option for the CXP parser for more information.
- **Catalog**. The name of this feature is "http://commerceone.com/sax/properties/catalog". This name is defined in SAX20Strings as the constant COMMERCEONE_CATALOG_PROPERTY. The associated value is of type String. It sets the catalog file the parser uses to locate schemas from URN's See the -c option for the CXP parser for more information.

- **Catalogs**. This enables you to specify more than one catalog file. The name of this feature is "http://commerceone.com/sax/properties/catalogs".
  This name is defined in SAX20Strings as the constant COMMERCEONE_CATALOGS_PROPERTY. The associated value is of type Vector, populated with Strings. It sets the catalog URIs used by the parser.

The CXP classes that implement the SAX drivers are `SAX10Driver` and `SAX20Driver`. They are in the package com.commerceone.xdk.standards.sax.

`SAX10Driver` implements SAX 1.0. Since it does not provide configuration capabilities, there is no way to set the validation mode (which is off by default) on the parser.

In order to run CXP in validating mode (which is how one takes advantage of all the advanced CXP capabilities) `SAX20Driver` has to be used.

Using CXP with SAX

The following steps describe how an application can use CXP with SAX.

1. The following import statement must be present in your code:

```
import org.xml.sax.*;
import com.commerceone.xdk.standards.sax;
import com.commerceone.xdk.standards.sax.SAX20Driver;
import com.commerceone.xdk.standards.sax.SAX20Strings;
```

2. Implement the handler interfaces in one or more handler classes. One possibility is to extend `HandlerBase`, a SAX standard helper class that implements all the handler interfaces and provides default behavior (does nothing) for all the methods. You can then implement the methods that are relevant for your implementation

.

```
class AllHandler extends HandlerBase
  {
    public void startDocument()
      throws SAXException
    {
      // handle the startDocument event
    }
    public void endDocument()
      throws SAXException
    {
       // handle the endDocument event
    }
    public void startElement(String name, AttributeList atts)
       throws SAXException
    {
      // handle the startElement event
    }
    public void endElement(String name)throws SAXException
    {
      // handle the endElementEvent
    }
}
```

3. To instantiate the parser, put one of the following statements in your code:

```
org.xml.sax.Parser parser = new
com.commerceone.xdk.standards.sax.SAX10Driver();
```

or

```
org.xml.sax.Parser parser = new
com.commerceone.xdk.standards.sax.SAX20Driver();
```

4. To set the handlers, use the following:\

```
// create one or more handler objects
AllHandler handler = new AllHandler();

// set the relevant handlers
parser.setDocumentHandler( handler );
parser.setErrorHandler( handler );
```

5. To set features/properties in `SAX20Driver`, use the following:

```
// cast the parser to Configurable
org.xml.sax.Configurable configurable =(org.xml.sax.Configurable)parser;

// set validating mode
boolean validation = true;
configurable.setFeature( SAX20_VALIDATION_FEATURE, validation );

// getting the current validation mode
validation = configurable.getFeature( SAX_VALIDATION_FEATURE );

// set schema path
String path = "/commerceone/xdk/xml/myschemas";
configurable.setProperty( COMMERCEONE_SCHEMAPATH_PROPERTY, path );

// setting the catalog property
String catalog = "file:///d:/mycatalogs/catalog.xml";
configurable.setProperty( COMMERCEONE_CATALOG_PROPERTY, catalog );

catalogs is a vector of strings that are catalog filenames.
Vector catalogs = new Vector();
catalogs.addElement("http://www.commerceone.com/catalogs/cat.xml");
catalogs.addElement("file:///myschemas/samplecatalog.xml");
catalogs.addElement("file:///d:/test/testcat.xml");

configurable.setProperty( COMMERCEONE_CATALOGS_PROPERTY, catalogs );
```

6. Start the parsing of a SOX or XML instance document as follows:

```
String systemId = "file:///commerceone/xdk/samples/sample.xml";
parser.parse(systemId);
```

7. At this point in the processing, events start to arrive at the registered handlers.

It is also possible to parse an `org.xml.sax.InputSource` instance, that can be instantiated with either a System ID, a `java.io.Reader` object or a `java.io.InputStream` object. In order for this to work, a System ID has to be set on the InputSource object. Once the InputSource object has been created, this can be done with a call to the method `InputSource.setSystemId()`.The system ID provided is necessary for CXP to work, and is used as a working path for the source. The System ID will be used as a base for any relative file names that are encountered.

SAX Sample

A functional SAX sample can be found in the file `C:\commerceone\xdk\sample\apps\SAXTest.java`. on Windows and `/opt/CMRCxdxp/sample/apps/SAXtest.java` on Solaris.
This sample is intended as:

- an example of how SAX can be used,
- a possible starting point in developing a SAX application with CXP, and
- a debugging utility that sends the contents of the DocumentHandler events to the console while parsing a document.

This sample does the following:

- Instantiates a parser (either the default parser or the one specified on the command line).
- Sets a number of parser features, according to what the user specifies on the command line.
- Attaches a document and an error handler to the parser.
- Starts parsing the XML document.

- Receives the events generated by the parser during the parse.
- Prints a representation of the document to the screen.

Note that for the sample to work, you must both compile the `SAXTest.java` file and set your local classpath to point to the compiled SAXTest class as well as the jar files that reside in the `lib` directory of this installation.

This sample can also be connected to any SAX-supporting XML parser and it can be used either from the command line or as part of another package. The command line usage is:

```
javac SAXTest <options> <instance URI>
```

The options to `SAXTest` are:

`-p:<schema_path>` sets the schema path. The value should be a directory name. This option has the same functionality as CXP's -p option (Section 3.3).

`-c:<catalog>` sets the catalog URI. The required format is a URI or a file name.

`-e:<encoding>` Sets the encoding that should be assumed while parsing.

`-v` turns on validation. By default, the sample application sets the parser in non-validating mode.

`-f` sets the force file mode to on. By default force file is off.

`instance URI` The XML or SOX instance document to parse. The required format is a URI or a file name.

When the `-v` option is not used, the `-p` and `-c` parameters are ignored. In validating mode (with option -v), use one of either `-p` or `-c`, but not both. If you specify both `-p` and `-c`, `-p` is ignored.

Examples

The following examples assume that you are using the Sun VM java.

```
java SAXTest file:///testdoc.xml
java SAXTest -p:/schemas;/commerceone/xdk
/testdoc.xml
java SAXTest -c:/catalogs/catalog.cat -v
/testdoc.xml
```

The CXP SAX driver is used by default, but this sample can be connected to a different parser that supports SAX. To do this, set the system variable `"org.xml.sax.parser"` to the class that implements the `SAX10 driver`.

```
java -Dorg.xml.sax.parser=com.xyz.Parser SAXTest testdoc.xml
```

If you use another parser (as shown above), only the non-validating mode is supported, since other parsers do not necessarily implement the SAX 2.0 interfaces to configure the parser. Even if they do, they do not have the same property and features names.

CXP and XT

XT also constitutes an example of a SAX application. It is available in source code format, and has been used and tested with many SAX parsers.

James Clark's XSL processor, XT version 11051999, is installed as part of the CXP package. XT can be plugged into any parser that supports SAX 1.0. It can be used with CXP practically without any modification, when the parser is run in non-validating mode.

If validation is desired, some changes are necessary to allow configuring the parser. We provide a class called `com.commerceone.xdk.standards.xsl.XSL` that connects CXP with XT. We also provide a command line utility `cxsl.bat` in the `C:\comerceone\xdkpro` directory on Windows and the `/opt/CMRCxdkp` directory on Solaris, that you can use to run this class.

```
java com.commerceone.xdk.standards.xsl.XSL
-p <schema_path> -c  <catalog_file> -novalid
<xml_file.xml> <xsl_file.xsl> <out_file>
```

Note.......... In the current implementation, neither the XSL class, nor the command line tool supports XSL include statements.

Options

The supported options are:

`-p <schema_path>` - sets the schema path. The value should be a directory name. This option has the same functionality as CXP's -p option (Chapter 4.)

`-c  <catalog_file>` - sets the catalog. This argument must be in filename format.

`-novalid` - sets the parser to non-validating mode (validation is on by default)

`<xml_file.xml>` - The XML file to use. This argument must be in filename format.

`<xsl_file.xsl>` - The XSL file to use. This argument must be in filename format.

`<out_file>` - A file to print the output to, instead of the screen. This argument must be in filename format.

Examples

In order for the following examples to work, you have to set the class path either as an system environment variable or on the command line, to include the sample class, as well as all the jar files contained in `C:\comerceone\xdkpro\lib` directory for Windows and the `/opt/CMRCxdkp/lib` directory on Solaris. A batch file, `cxsl.bat` for Windows and `cxsl.sh` for Solaris that contains the class path as well as the command line is provided as a more convenient way of running the class XSL. XT can also be used as part of another application, through its published interfaces.

The following example is the Windows version:

```
java com.commerceone.xdk.standards.xsl.XSL
\xml\file.xml \xml\file.xsl
 java com.commerceone.xdk.standards.xsl.XSL
 -p \schemas\myschemas \myfiles\file.xml
 \myfiles\file.xsl

 java com.commerceone.xdk.standards.xsl.XSL
 -c \catalogs\catalog.cat \myfiles\file.xml
 \myfiles\file.xsl
```

The following example is a Solaris version:

```
java com.commerceone.xdk.standards.xsl.XSL /xml/
file.xml /xml/file.xsl
 java com.commerceone.xdk.standards.xsl.XSL
 -p /schemas/myschemas /myfiles/file.xsl

 java com.commerceone.xdk.standards.xsl.XSL
 -p /schemas/myschemas/ /myfiles/file.xml
 /myfiles/file.xsl
```

Chapter 6
How to Use the SOX to Java Compiler (X2J)

In This Chapter

This chapter describes how to use the SOX to Java Compiler.

What is X2J?

X2J is a SOX 2.0 to Java translator. It is used to map SOX 2.0 compliant XML Schemas to Java code. It can also be used to validate SOX 2.0 schemas.

Class Paths

You do not need to have a classpath system variable set when running X2J. The provided script for running X2J sets the classpath system variable appropriately.

X2J Options

X2J should be run with the provided script, `x2j.bat` for Windows and `x2j.sh` for Solaris, that resides in the `C:\commerceone\xdkpro` directory on windows and `/opt/CMRCxdkp` directory on Solaris. Running X2J with no options presents usage information on the screen:

```
X2J.INFO: Usage is X2J [-emit emit code ]
[-v verbose ][-help help screen ]
[-? usage ][-enc XML File encoding ][-p schema path ]
[-d Root Directory for generated code ]
[-f accept files ]
[schema: SOX schema file ]
```

The X2J options are:

X2J Option	Description
`-emit`	X2J will emit Java code corresponding to the input SOX 2.0 schema. This option does not take an argument. If this parameter is specified, then the -d parameter below also has to be specified. Note that if you do not use this option, X2J will not generate any code.
`-v`	X2J outputs verbose messages during the parse if this option is specified. This option does not take an argument. With this option, X2J displays additional information related to compiling the schema. This information is helpful in finding and fixing problems in your schema.
`-help`	X2J shows detailed help information (similar to this) if this option is specified. This option does not take an argument.
`-enc <encoding>`	Specifies the XML file encoding to use. Supported encodings are the same as the encodings supported by the particular JVM being used. Some possible values are "UTF8", "Unicode", "SJIS", "8859_1", "8859_15", "ASCII", "Big5", "GB2312", "KSC5601", "Cp874", "JIS". If this option is not used, the default encoding is UTF8. No matter what encoding you use, all the files encountered in one compile must use the same encoding. You cannot for example compile a schema that uses one encoding, that in turn refers to a schema that uses another encoding.

X2J Option	Description
`-p <schema-path>`	The schema path; a semicolon delimited list of paths to use as root for schema search. This is similar in notion to a class path. X2J needs to know the root path from which to resolve any URNs that the input document refers to. If the document is a SOX schema, or an XML instance conforming to a SOX schema, then the schema refers to schema.dtd and it is necessary to include a path can resolve the URN of that file. In this installation, this path is the `c:\commerceone\xdkpro directory` on Windows and `/opt/CMRCxdkp` on Solaris that contains the command line tool scripts. If you have not placed your schemas n the XDK installation, you must also add a path to your schemas.
`-d <root-directory for generated code>`	An absolute path underneath which the generated code will be placed, according to the package name of the generated code. The package name of the generated code depends upon the URN of the schema being compiled. This is similar in notion to the schema path. This parameter has to be specified if the `-emit` flag is specified, and it is not useful to specify it unless the `-emit` flag has been specified.
`-f`	The `-f` option will force CXP to accept regular file names for the input files. By default CXP assumes that the input document name provided is a URI. Since a URI will always contain at least one colon, ":", CXP can recognize that an input document that contains no colons must be a file name, and process the input as a file name instead of a URI. However, in those cases where you use a filename that contains a colon such as `c:\schemas\TestSchema.sox`, you can tell CXP to still interpret the input as a file name. The `-f` option enables this by forcing CXP to always interpret the input as a file name. For example, on Windows, given the file po.xml, you can use either `"cxp \commerceone\po.xml"` or `"cxp -f c:\commerceone\po.xml`". This option does not take an argument.
`schema`	The name of one single schema that X2J should compile. This can be in either file or URI format, (see the -f option above).

How URNs are Used in Code Generation

A detailed discussion of the mapping between URNs and schema locations is provided in the section titled *CXP Options* in Chapter 4.. It describes how our software uses the schema path in conjunction with the schema or DTD URN, to locate the corresponding DTD and/or schema files. In this section, we focus on how the schema URN is used, in conjunction with the root directory for the generated code specified by the `-d` option, to determine the directory path where the Java files will be generated.

In the following example, let us assume that the schema is contained in the file `sample.sox`. The Commerce One implementation uses the URN of the schema to determine both the physical location of the schema, and the package name of the Java code corresponding to the schema. In this example the URN of `sample.sox` is:

```
urn:x-commerceone:document:sample:xdk:sox:sample.sox$1.0
```

The root of the generated code is `$<ROOT DIRECTORY>.` This is what is provided with the argument to the -d option to X2J.

1. Recall that the URN always starts with `"urn:x-commerceone:document:"`. This part specifies the scheme for the mapping, and does not map to anything in the path to the generated code.
2. The `"$1.0"` token at the end of the URI indicates the schema version. The version token in this release is always `"$1.0"` and does not currently map to anything in the path to generated code.
3. The remaining portion of the URN, `sample:xdk:sox:sample.sox`, is used to create the package name of the generated code, and maps to the location of the generated code. For the path the colons are replaced with file separators, and the period, '.' in `sample.sox` is replaced by an underscore. The resulting path is `sample\xdk\sox\sample_sox`. For the package name the colons are replaced with periods, ".", and the period, '.' in `sample.sox` is replaced by an underscore. The resulting package name is `sample.xdk.sox.sample_sox`.

4. The root of the generated code tree on your file system is here represented as <ROOT DIRECTORY>. This is the path specified with the -d option to X2J. Combining this with the path generated above, the location of the generated code on your file system will be `"<ROOT DIRECTORY>\sample\xdk\sox\sample_sox".`

Examples of Using X2J

Let's assume that the sample schema below is located at `c:\mywork\com\mycompany\lang\n1.0\LangString.sox` for windows or `/mywork/com/mycompany/lang/n1.0/LangString.sox` for Solaris.

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
"urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">
<schema uri=
"urn:x-urn:x-commerceone:document:com:mycompany:lang:LangString$1.0
<elementtype name="LangString">
      <empty/>
      <attdef name="Lang" datatype="string">
             <required/>
      </attdef>
</elementtype>
</schema>
```

Here are some examples of how to use some of the X2J command line options:

Example 1 for Windows:

```
x2j -p \commerceone\xdkpro
\mywork\com\mycompany\lang\n1_0\LangString.sox
```

Example 1 for Solairs:

```
./x2j.sh -p /opt/CMRCxdkp /mywork/com/mycompany/lang/n1_0/LangString.sox
```

This is the simplest use of X2J. It has the effect of validating the schema LangString.sox.

Example 2 for Windows:

```
x2j -p \commerceone\xdkpro\ -d \soxbeans
-emit \mywork\com\mycompany\lang\n1_0\LangString.sox
```

Example 2 for Solaris:

```
./x2j.sh -p /opt/CMRCxdkp/ -d /soxbeans -emit
/mywork/com/mycompany/lang/n1_0/LangString.sox
```

X2J validates the schema LangString.sox, and emits corresponding Java code as specified by the SOX to Java mapping. The emitted code has the package name `com.mycompany.lang.LangString_sox,` as per the schema URI. For more details on the schema URI to Java package mapping, see the Bean Tutorial included in this software package. Finally, the emitted code is placed in the directory `\soxbeans\com\mycompany\lang\LangString_Sox` for Windows or in the directory `/soxbeans/com/mycompany/lang/LangString_Sox` for Solaris. This is a result of combining the generated bean root directory as specified with the `-d` option, and the package name of the generated code.

Examples of Using a Sample File with X2J

In this distribution we have included several sample SOX 2.0 schemas. The sample SOX schemas are located in
`c:\commerceone\xdk\sample\xdk\sox\n1_0`
for Windows or
`/opt/CMRCxdkp/sample/xdk/sox/n1_0`
for Solaris.

To compile the SOX schema file
`Film.sox`,
go to the sample directory
`C:\commerceone\xdkpro` for Windows or `/opt/CMRCxdkp`
for Solaris and type the following on the command line (for Windows)::

```
x2j.bat -p \commerceone\xdk -d \soxbeans -emit
sample\xdk\sox\n1_0\Film.sox
```

or for Soalris:

```
./x2j.sh -p /opt/CMRCxdkp -d /soxbeans -emit
sample/xdk/sox/n1_0/Film.sox
```

Compiling SOX Beans

When generating the Sox Beans from a SOX schema, a Java package will be generated as described in the previous section.

To compile the sox beans, use any java compiler, and in the directory containing the beans, compile all the generated java files. This can easily be done by running any java compiler in the directory where the beans were generated, specifying the file to compile to be *.java:

In Windows:

```
javac -d  /commerceone/xdkpro/classes *.java
```

In Solaris::

```
javac -d  \commerceone\xdkpro\classes *.java
```

The above example will compile all the java files in the current directory, and place the compiled classes, in the directory /commerceone/xdkpro/classes.

Keep in mind that you need to set the classpath to contain the XDK jar files as well as a path to the generated beans (which is the directory specified with the -d option to x2j) in your classpath. Either set it in the system, or as an option to the compiler when compiling the beans. The XDK jar files can be found in `C:\commerceone\xdkpro\lib` on Windows and `/opt/CMRCxdkp/lib` on Solaris.

Also keep in mind that if your schema imports a second namespace, you need to generate the Sox beans generated from that namespace first, before you can compile the beans emitted from your current schema. The classpath provided to the java compiler must include a path to the beans generated from the imported namespace. For example, the sample schema House.sox imports a different namespace Rooms:

```
<schema uri="urn:x-commerceone:document:sample:xdk:sox:House.sox$1.0">
<namespace prefix="room"
namespace="urn:x-commerceone:document:sample:xdk:sox:Rooms.sox$1.0"/>
```

In order for the Java Beans generated from House to compile, you need to first compile the Java Beans generated from Rooms.sox, and then include a path to the Rooms.sox beans in the classpath when compiling the beans from House.sox.

Troubleshooting

Problem	Solution
X2J reports: `Entity "<external-subset>" at location "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0" cannot be opened due to:` **`Domain urn does not exist.`**	You did not specify a schema path, so X2J can not interpret the urn mapping. Use the `-p` option to specify a schema path that points to your schemas, as well as schema.dtd
X2J reports: `Entity "<external-subset>" at location "urn:xcommerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0" cannot be opened due to: Could not find file /commerceone/xdk/foo/com/commerceone/xdk/xml/n1_0/schema.dtd.`	The schema path you provided was not sufficient to find schema.dtd. Make sure the path you specify with the `-p` option includes a path to schema.dtd.
The application executed successfully, but you cannot locate the generated beans.	You did not specify the `-emit` and `-d` options

Problem	Solution
X2J reports an error such as: `Mismatched token: line(20), expecting MDC, found`	You may be using a lesser version of the Virtual Machine than recommended. See the readme in the Doc Directory in this installationfor the VM version to use.

X2J reports an error such as
`Invalid character number....`
or:
`problem with IO or possible invalid character for current encoding: Missing byte-order mark`
	Your document does not use an appropriate encoding, or may even contain mixed encodings.
	By default, X2J expects all XML files to be in UTF8 (which is backward compatible with 7-bit ASCII). If you want to process documents with a different encoding, such as UTF16 or 8-bit Latin, you must use the `-enc <encoding>` option. If you want to compile a Unicode document, you need to make sure that your document has the appropriate byte-order mark for your system, otherwise X2J will not be able to process it correctly. In addition, make sure your document only contains one encoding. If it has been assembled from different sources, it may contain several different encodings.
	Note that all input files used in one run must use the same encoding. X2J cannot dynamically switch encoding schemes while processing.

Chapter 7
The Simple RoundTrip Application

In This Chapter

This chapter describes how an application can use the XDK to read in an XML instance document and obtain a Document Object, and how the Document Object can be converted back into an XML instance document. A functional example of this exercise can be found in (Windows):

```
C:\commerceone\xdkpro\sample\apps\SimpleRoundTrip.java
```

or in Solaris:

```
 /opt/CMRCxdkp/sample/apps/SimpleRoundTrip.java
```

SimpleRoundTrip Application Basic Code

This section contains some of the basic code in the SimpleRoundTrip application.

1. Create the CXP Parser and initialize the XDK:

```
import com.commerceone.xdk.base.parser.CXPParser;
import com.commerceone.xdk.initialize.XDK;

CXPParser cxpParser = new CXParser();
try {
  XDK.init();
}
catch(com.commerceone.xdk.excp.initialize.
AlreadyInitializedException aie){
 //Handle Exception
}
```

2. Set the schema path for the parser (for Windows)::

```
String schema_path = "\commerceone\xdk";
cxpParse.setURNPath(schema_path);
```

For Solaris:

```
String schema_path = "/opt/CMRCxdkp”;
cxpParse.setURNPath(schema_path);
```

3. Convert the input XML instance document file into a URI:

```
import com.commerceone.util.net.URI;


File fsysid = new File("Film.xml");
URI fileURI = FileURIDomain.toURI(fsysid);
```

4. Open the URI corresponding to the XML instance document file as an External Source:

```
import com.commerceone.xdk.base.parser.URIEntityManager;

Import com.commerceone.xdk.base.parser.ExternalSource;

ExternalSource source = cxpParse.getEntityManager().open(fileURI);
```

5. Create a Document Factory:

```
import com.commerceone.xdk.metadox.factory.DocumentFactory;

DocumentFactory docFactory = new DocumentFactory();
```

6. Use the Document Factory to create a Document Object out of the External Source:

```
import com.commerceone.xdk.metadox.model.object.DocumentObject;

DocumentObject indoc =(DocumentObject)docFactory.fromSource(source);
```

7. Create an XML instance document out of the Document Object:

```
indoc.toStream(System.out);
```

The SimpleRoundTrip Application

To use the `SimpleRoundTrip` application from the command line, you must compile the `SimpleRoundTrip.java` file, and use X2J to compile the schema corresponding to the XML instance document that you wish to use as input to the `SimpleRoundTrip` application. Also, you must set your classpath to point to the compiled SimpleRoundTrip class, the classes corresponding to the Sox Beans generated by the X2J compiler, and the jar files that reside in the `lib` directory of this installation.

Then, to use this application from the command line:

```
java SimpleRoundTrip -p <options> <XML instance>
```

Running `SimpleRoundTrip` with no options will present usage information on the screen:

```
usage: ./SimpleRoundTrip [-options]

  -p <path>      Schemapath
  out <file>     File to which output is written
  -f             Accept files as input
  -?             This message
  -help          This message
  inputFile      Input File
```

The SimpleRoundTrip options are:

Option	Description
`-p <schema-path>`	The schema-path; for more detail on schema paths see Chapter 4 or Chapter 6.
`-out <outfile>`	The File to which the output XML instance document should be written.
`-f`	This option forces the input XML instance document name to be treated as a file name rather than as the name of a URI. This option takes no parameter. For more detail on this option see Chapter 4.

Option	Description
`-?`	Present the usage information.
`-help`	Present the usage information.
`inputFile`	The name of the XML instance document file or URI that is to be round-tripped.

An Example

For this example to work, the Classpath must be set to include the Jar files in the lib directory. The Simple Round Trip class and the compiled beans for Film.sox.

The following example is for Windows::

```
java SimpleRoundTrip -p \commerceone\xdkpro\
sample\xdk\instances\namespaces\Film.xml
```

The following example is for Solaris::

```
java SimpleRoundTrip -p /opt/CMRCxdkp/
sample/xdk/instances/namespaces/Film.xml
```

Chapter 8 Creating and Manipulating a Sox Bean

In This Chapter

This chapter describes how an application can use the XDK to create and manipulate the Java Beans that result from compiling SOX schemas.

Example

In the following example, we will use the schema (for Windows)

```
C:\commerceone\xdkpro\sample\xdk\sox\n1_0\Beverage.sox
```

and (for Solaris)

```
/opt/CMRCxdkp/sample/xdk/sox/n1_0/Beverage.sox
```

and the bean `Beverage.java` as derived from the elementtype `Beverage` defined in `Beverage.sox`

In order for this example to work, you also have to compile Sox Beans from the schema `Container.Sox`, which is used by the Beverage schema. You must do this before attempting to compile the Beverage beans with a Java compiler.

.

```
<elementtype name="Beverage">
   <model>
      <sequence>
         <element name="Name" type="string"/>
         <choice>
            <element name="Can" type="AluminumCan"
                     prefix="containers"/>
            <element type="GlassBottle"
                     prefix="containers"/>
            <element type="PaperCup"
                     prefix="containers"/>
         </choice>
      </sequence>
   </model>
   <attdef name="Volume" datatype="float">
      <required/>
   </attdef>
   <attdef name="VolumeUnit" datatype="Unit">
      <default>fluid ounces</default>
   </attdef>
   <attdef name="Price" datatype="float">
      <required/>
   </attdef>
   <attdef name="Carbonated" datatype="boolean">
      <default>true</default>
   </attdef>
</elementtype>
```

Given the elementtype `Beverage`, you get the interface `Beverage` and the class `BeverageImpl` that implements `Beverage from x2j`. For details on why the `Beverage` elementtype maps to the presented `Beverage` interface, refer to the enclosed Sox Bean Tutorial.

```
public interface Beverage extends ElementType {
  public BeverageAttributes
getBeverageAttributes();

  public void setName(String s);
  public String getName();

  public void setBeverageChoice(
      sample.xdk.sox.Beverage_sox.BeverageChoice
s);
  public sample.xdk.sox.Beverage_sox.BeverageChoice
       getBeverageChoice();
};
```

There are two methods of obtaining an instantiation of the `Beverage` bean:

- You can instantiate the bean yourself:

```
import sample.xdk.sox.Beverage_sox.*;

Beverage myBeverage = new BeverageImpl();
```

- As described in the previous chapter, you can use the `DocumentFactory` to read in an instance of a `Beverage`, and get back a `DocumentObject`. This `DocumentObject` can then be cast into a populated `Beverage` bean:

```
import sample.xdk.sox.Beverage_sox.*;

DocumentObject indoc = (DocumentObject)
               docFactory.fromSource(source);
Beverage myBeverage = (Beverage) indoc;
```

Once you have a bean, it is populated using the `set()` methods, and accessed using the `get()` methods. Thus, to get the `name` of the `Beverage`:

```
String myBeverageName = myBeverage.getName();
```

To set the name of the `Beverage`:

```
myBeverage.setName("Coca Cola");
```

To access the attributes of the `Beverage`:

```
BeverageAttributes myBeverageAttributes =
                myBeverage.getBeverageAttributes();
```

Note that you never need to create the `BeverageAttributes` yourself. These are instantiated when the `Beverage` bean is instantiated. To set a particular attribute, say `Volume`:

```
myBeverage.getBeverageAttributes().setVolume(51.0);
```

A functional example of this exercise can be found in (for Windows):

```
C:\commerceone\xdkpro\sample\apps\BeanManipulationExample.java
```

and (for Solaris):

```
/opt/CMRCxdkp/sample/apps/BeanManipulationExample.java
```

The code presented above is some of the basic code in the `BeanManipulationExample` application. The arguments to the application are the same as those to the `SimpleRoundTrip`, and in fact, the `BeanManipulationExample` application requires that `SimpleRoundTrip` be compiled first.

Chapter 9
Document Framework

In This Chapter

In this chapter we describe the programmatic interfaces used by applications for handling and manipulating documents. We call these interfaces the Document Framework. The Document Framework is the contract between the XDK and third party application developers. It provides a set of interfaces to the XDK, to be used by third parties such as service developers. These interfaces provide an abstract view of a document regardless of the programmatic representation that is used for the document and therefore an application can handle a document of any representation appropriately. The Document Framework is designed with the following intents:

1. The XDK needs to define document representations so that users can act on documents without knowing about content, format or representation, while applications can create representations of documents that enables them to manipulate the document in the most efficient manner.
2. Third parties must be able to handle attachments as well as versions of documents.

The Document Framework allows a variety of document representations, locales, and formats.

Terminology

Term	Description
Document	This is the base abstraction in the Document Framework. Documents have type, identity, content and more.
Document Representation .	A document can have one representation at a time out of several possible alternatives. Some possible representations are DocumentObject (Bean, COM), DocumentStream (raw character stream) and DocumentBytes(raw byte stream). Representations are only meaningful for Service implementations
Document Content	The actual content of a document, e.g. a PurchaseOrder document holds information about item, price, references and more.
Document Format	In a normalized world we would only assume XML derived from SOX. However, we know that this is not always the case for XDK users. Formats are meaningful concepts for users of services as well as service implementers. It is part of the contract, so both Service Description and Service Specification documents list format requirements.
Document Type	An XML document has a document type. In SOX derived XML we talk about SOX type. In DTD derived XML we talk about DOC type.
Document Type Version	As Document Types change, it is important to identify versions of types. For example if a new field was added to a PurchaseOrder, it is still of the same type as before, but now has a new version number. A service accepting PurchaseOrder can decide what versions it accepts, if external mapping is required, etc. It is important to remember that Document Version (instance) and Document Type Version (type) are two different concepts.

Term	Description
Document Version	Documents are often versioned. For example a bookkeeping document (Document Exchange Protocol) gets signatures appended as Business documents are migrating through the system. The specific bookkeeping document has changed, that is it has a new version, but its identity is still the same. It is important to remember that Document Version (instance) and Document Type Version (type) are two different concepts.
Document Identity	A document needs to have a unique identity. The scope of the uniqueness is application specific, but XCC still needs the guarantee that two documents that are different do not have the same identity. The Util package in the Commerce Platform system provides an Interface called Identity with implementations for UUID and Long. An application specific identity, say PurchaseOrder number, combined with the scope should also be allowed. For unknown Document Streams a size/hash value can be used together with the scope.
Document Locale	For internationalization reasons a document has an associated locale. This is to identify Language Code, Country Code and Variant. The language codes are the lower-case two-letter codes as defined by ISO-639. The country codes are the upper-case two-letter codes as defined by ISO-3166. The Variant codes are vendor and browser-specific. For example, use WIN for Windows, MAC for Macintosh and POSIX for POSIX.
Document Wrapper	This is a class that contains Documents. Examples of Document Wrappers are Envelopes and Reply wrappers. The way we contain the Document Stream and create another representation of a Document is different from DocumentWrapper. A DocumentWrapper is a Document.
Envelope	This is the base abstraction in the Document Framework. Envelopes hold one and only one document. They can also hold attachments. Properties are kept to facilitate routing and application service code. An Envelope is a DocumentWrapper and ergo a Document.

Term	Description
Message	When an Envelope is transmitted over a wire it becomes a message. That message is a 1-1 mapping of the envelope into an on-the-wire format using MIME and XML.
Quality of Service (QoS)	A term used to qualify the level of service in various aspects. In the case of exchanging documents, it is interesting to talk about security, guaranteed delivery, response time and such. Synchronous or asynchronous are NOT QoS statements.
URN	A Uniform Resource Name (URN) is a persistent, location-independent resource identifier. The syntax of a URN consists of three parts: a reserved "urn:" identifier, a Namespace Identifier (NID) string, and a Namespace Specific String (NSS NIDs can only contains the characters a-z, A-Z, and the dash (-) character). Examples: urn:x-someidentifier:somestring-19980101231145, urn:isbn:1-56592-169-0. We want to stress the difference between identities, which are meant for machines, and symbolic names, which are meant for humans. A UUID is an identity. A URN is a symbolic name.
URI Catalog	Catalog to map a symbolic URI to a URI that has transport and location information.
UUID	A UUID is a 16-byte globally unique identity. We want to stress the difference between identities, which are meant for machines, and symbolic names, which are meant for humans. A UUID is an identity. A URN is a symbolic name.

Document Representation Classes

Representations

An XML Document has several different possible programmatic representations. In the XDK, we have identified five basic representations, but we anticipate that additional representations may be required in the future. The five basic representations are:

- *DocumentObject*: Representation as strongly typed Java objects.
- *DocumentStream*: Representation as a raw character stream.
- *DocumentBytes*: Representation as a raw byte stream.

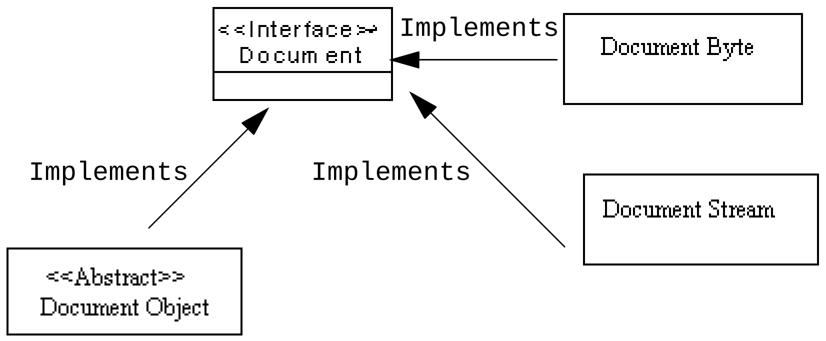
The DocumentStream and DocumentBytes representations are in fact intended to support non-XML documents as well.

We use the term *programming model* for the view of an incoming XML document used by an e-commerce application developer. Different uses of a document will require different programming models. The idea of the Document Framework is to provide a structure that enables efficient programmatic manipulation of documents.

Term	Description
DocumentObject [SOX to Java Mapping]	A more strongly typed programming model than DOM. With this model programmers get a more convenient and safer way of manipulating and reading documents that conforms to SOX schemas. Each element is represented by a typed object that has the methods for accessing and manipulating the content specified for the element in the Schema.

Term	Description
DocumentStream	Provides a simple character stream that contains the document. The document has not been processed in any way when the application receives it. This programming model is very important for MarketSite hubs that perform only routing. This programming model enables fast routing and de-couples routing nodes from any document specific knowledge. Programming with formats other than XML will benefit from this representation.
DocumentBytes	Provides a simple byte stream that contains the document. The document has not been processed in any way when the application receives it. This programming model is similar to DocumentStream, except it uses raw bytes.

The Document Framework has separate classes for each of these programming models. A representation of a particular document either extends the corresponding Document Framework class, or is encapsulated by it. The figure below illustrates that each document representation has a corresponding class that implements the `Document` interface.



`DocumentTree, DocumentObject` and `DocumentEvents` are representations of XML documents. `DocumentStream` and `DocumentBytes` on the other hand can be used to represent XML documents or documents in any other form, such as binary EDI documents, MicroSoft Word documents and so on.

DocumentObject

The `DocumentObject` is an abstract base type for all interfaces that implement the X2J representation of documents. The X2J Mapping defines a class `ElemenTypeImpl` that extends the `DocumentObject` class. For each elementtype X, according to X2J, there is a generated interface X, and a generated class `Ximpl` that implements `X`. Since `DocumentObject` implements `Document,` the class `Ximpl` corresponding to X also implements `Document`. The figure below illustrates the mapping for elements in the X2J mapping. The emitted entities are filled with gray.

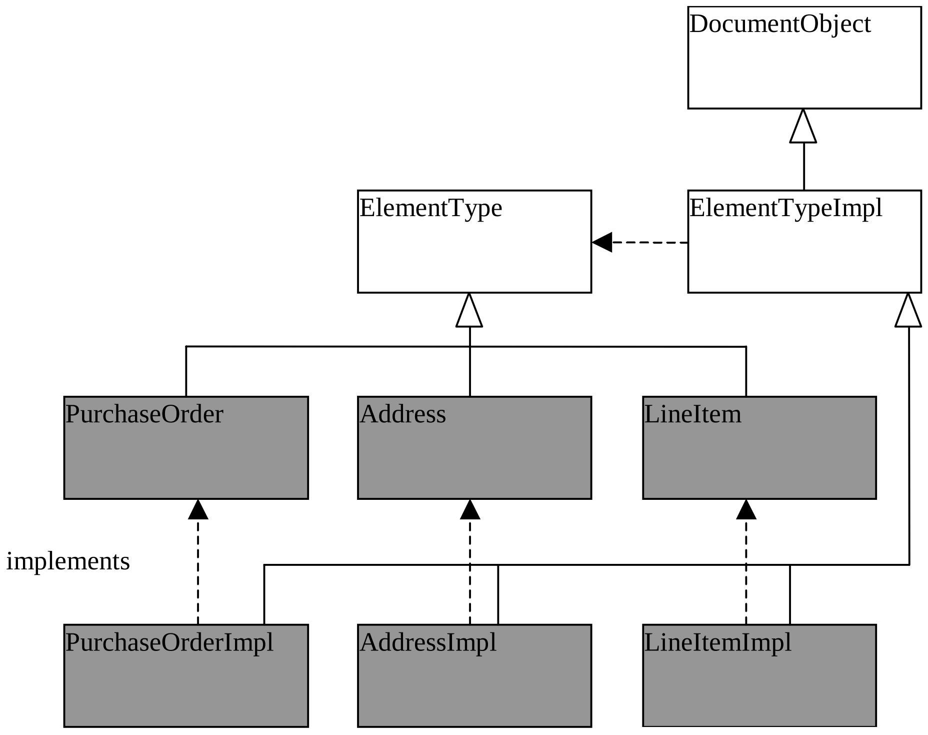


Figure 9-1 The DocumentObject Representation

DocumentStream

The `DocumentStream` is a base class for all character stream based document representations. The `DocumentStream` programming model allows an application to keep a document as a raw stream. The stream can be an XML document or a document in some other format. The application can retrieve the stream and process it in any way that it wants. It could even apply one of the other XML document representations to it using the `UnMarshall` interface methods.

The `DocumentStream` representation is particularly useful for non-XML documents or when there is no need to interpret the document content in order to handle the document. Header based routing is an example of the latter.

```
package com.commerceone.xdk.metadox.model.stream;

public class DocumentStream extends AbstractDocument
 implements AnnotatedDocument,CharacterDocument
{
    public DocumentStream(DataSource s);
    public DocumentStream(DataSource s, Type type);

     public Reader getStream();
    public void toStream(Writer w);
}
```

The `DocumentStream` deals with `DataSources`:

```
package com.commerceone.xdk.swi.metadox.marshall;

public interface DataSource
{

    public void toStream(Writer writer)
        throws java.io.IOException;

    public Reader getReader()
        throws java.io.IOException;

    public void close()
        throws java.io.IOException;

    public DataSource copy()
        throws java.io.IOException;

}
```

A simple data source is `ReaderDataSource`, which converts a Java `Reader` to a `DataSource`:

```
package com.commerceone.xdk.metadox.model.stream;

import java.io.Reader;

public class ReaderDataSource implements DataSource
{
  public ReaderDataSource(Reader r)
}
```

The `DocumentStream` representation is immutable. When the representation is created we can assign an identity or MIME type to the representation. For the other document representations, the MIME type will implicitly always be something along the lines of text/XML.

DocumentBytes

The `DocumentBytes` is an abstract base class for all bytestream based document representations. The `DocumentBytes` programming model allows an application to keep a document as a raw stream. The application can retrieve the stream and process it in any way.

The `DocumentBytes` representation is particularly useful for binary documents when no concept of character encoding is appropriate.

```
package com.commerceone.xdk.metadox.model.bytes;

public class DocumentBytes extends AbstractDocument
 implements AnnotatedDocument,BinaryDocument
{
     public DocumentStream(DataSource s);
     public DocumentStream(DataSource s, Type type);

     public void toStream(OutputStream os);
}
```

The DocumentByte **deals with** ByteSources:

```
package com.commerceone.xdk.swi.metadox.marshall;

public interface ByteSource
{
    public void toStream(OutputStream os)
        throws java.io.IOException;

    public InputStream getInputStream()
        throws java.io.IOException;

    public void close()
        throws java.io.IOException;

    public ByteSource copy()
        throws java.io.IOException;
}
```

A simple data source is `StreamByteSource`, which converts a Java `InputStream` to a `ByteSource`:

```
package com.commerceone.xdk.metadox.model.bytes;

import java.io.InputStream;

public class StreamByteSource implements ByteSource
{
  public StreamByteSource(InputStream is)
}
```

The `DocumentBytes` representation is immutable. When the representation is created we can assign an identity or MIME type to the representation. For the other document representations the MIME type will implicitly always be something along the lines of application/octetstream.

Document Representation Factory and Schema Lookup

The goal of this section is to describe the support provided by the Document Framework for creating Document Representations. For this purpose, the Document Framework provides the `DocumentFactory`. The purpose of the `DocumentFactory` is to define a general interface for creating document representations, given an external source. Each document representation (or programming model) has to provide a factory class that implements the `DocumentFactory` interface. The figure below describes the relationship between the interfaces `BasicDocumentFactory` and `DocumentFactory` and their implementations.

To be able to describe the `DocumentFactory` interface, we need to first describe an `EntityManager`. The purpose of an `EntityManager` is to return a reference to an external data source given an URI. This external data source can in turn be handed over to the `DocumentFactory` for the creation of a Document Representation. There can exist many different implementations of `EntityManagers`, each using its own underlying lookup mechanisms. In some cases the lookup mechanism is a file mapping URIs to concrete addresses. In other cases an `EntityManager` implementation might use an LDAP directory for looking up a concrete address. Furthermore, the `EntityManager` may use different mechanisms for different types of URIs.

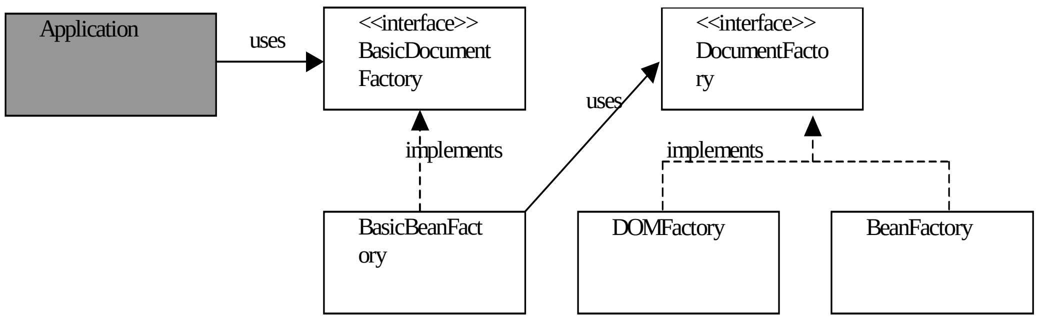


Figure 9-2 Factory Interfaces

The `DocumentFactory` interface requires that the client know about appropriate policies for finding schema information so that incoming documents can be validated against the schema they conform to. Both

`DocumentFactory` and `BasicDocumentFactory` provide a method named `fromSource`. The argument to `fromSource` is different in the two interfaces. For the `DocumentFactory` the `fromSource` expects an `ExternalSource`. In the `BasicDocumentFactory fromStream` expects a `Reader`. This means an implementation of `BasicDocumentFactory` must establish the policies on which a client relies. On the other hand, it simplifies the usage of the factory from the client's perspective.

Entity Manager

The entity manager takes a URI and resolves it to an `ExternalSource` that can be used for XML processing. The `EntityManager` interface can be implemented in many different ways. We envision the following implementations:

- An implementation that uses an URN catalog. The catalog maps URNs (A specialization of URI's) to files.
- An implementation that defines a mapping of URNs to a path that is used to search for files. This is similar to the `CLASSPATH` mechanism in Java where package names are mapped to files paths.
- An implementation that looks up an URI in an LDAP registry which returns a URL or file name. The manager then opens the named file or URL and returns a corresponding `ExternalSource`.

Here we show the interface for the `EntityManager`.

```
package com.commerceone.xdk.base.parser;

import com.commerceone.util.net.URI;

public interface EntityManager
{
   ExternalSource open(URI sysid)
       throws java.io.IOException;
}
```

External Source

The external source represents a stream (Reader) and context information that is considered valuable. In particular, it contains:

- A system identifier representing the resource.
- The URI that was used to locate the resource.
- The particular entity manager that was used to locate the resource.

```
package com.commerceone.xdk.base.parser;

import java.io.Reader;
import com.commerceone.util.net.URI;

public interface ExternalSource
{
   Reader getReader();

   String getSystemId();

   URI getURI();

   EntityManager getEntityManager();
```

Above we show the `ExternalSource` interface as it is currently defined in the XDK.

DocumentFactory

An application must be able to request a particular representation of a document from the Document Framework. More specifically, an application should be able to request an X2J object, DOM object, or other representation that is made available in the current installation.

Each document representation will provide a class that implements the `DocumentFactory` interface. When an application needs a particular representation, it will get an instance of the corresponding factory object, and call the `fromSource` method. The `fromSource` method requires an `ExternalSource` that may have been retrieved by using an `EntityManager`.

In some cases, one can establish a more widely applicable policy for URI lookup. Having such a policy will not only make lookup consistent, but it will also enable the usage of a simplified interface. The `BasicDocumentFactory` defines the interfaces that are used if a pre-defined URI lookup policy has been established. More specifically, an application library can implement `BasicDocumentFactory` in terms of a `DocumentFactory`. If an application uses a `BasicDocumentFactory`, it only needs to provide the `Reader` to create a document representation. This means that the application need not determine the lookup policies used. It does need, however, to be aware of the policies that are used, so that information can be installed appropriately. It also means that all applications using the same `BasicDocumentFactory` will use the same policies. This makes configuration more consistent and easier to maintain. The `DocumentFactory` and `BasicDocumentFactory` interfaces are outlined below.

```
package com.commerceone.xdk.metadox.factory;

public class BasicDocumentFactory implements
UnMarshaller
{
     public void fromSource(Reader reader);
}

public class DocumentFactory implements UnMarshaller
{
     public void fromSource(ExternalSource extSource);
}
```

Document Interfaces

In this section we describe the `Document` interface that all document representations are required to implement. Recall that one of the purposes of the Document Framework is to enable third parties to manipulate documents without any knowledge of the document representation. The Document interface enables us to fulfill this purpose. The structure of the Document interfaces is as illustrated by the figure below.

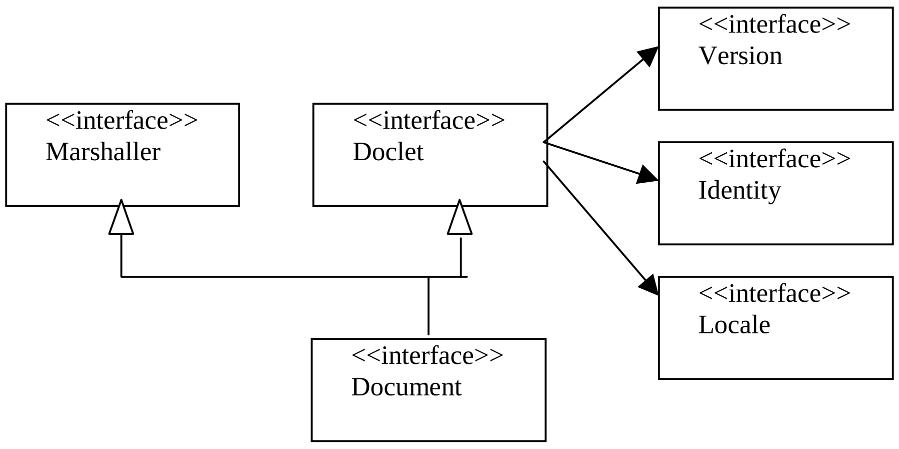


Figure 9-3 Document Interfaces

The Document interface collects all the functionality that a document representation must provide to any software component manipulating documents. This functionality can be divided into three parts:

- `Marshalling` - allows a client to create a stream representation of a document, and create a document from a stream respectively
- `UnMarshalling`- Same as Marshalling
- `Doclet` - provides methods for getting the document type, version, locale and identity of a document.

The following sections describe these interfaces in more detail.

Document

`Document` is the interface third parties use to handle a document regardless of its content and representation. To handle a document, an application only needs to know the information provided through the `Document` interface. As discussed earlier, specific document representations are subtypes of `Document`.

To retrieve document content, application services and clients need to pick a representation class - the generated X2J classes, `DocumentBytes`, or `DocumentStream`.

```
package com.commerceone.xdk.swi.metadox.meta;

public interface Document extends Doclet, Marshaller
{
     public Document getDocument();

     public BodyPartFactory getPartFactory();
}
```

For documents, the method `getDocument()` returns the current representation of the document. For Document Wrappers, e.g. Envelopes, the method returns the document they hold.

The method `getPartFactory()` returns a `BodyPartFactory`. The BodyPartFactory is what creates a MIME message part for the document in question.

Doclet

`Doclet` is the interface that defines the basic set of identifying information a document is required to support. This information is needed by components that cannot be dependent on the content of a document. Examples of such components include tracing, auditing, logging and billing. A `Doclet` has a `Type, Identity, Version` and `Locale`.

```
package com.commerceone.xdk.swi.metadox.meta;

import java.util.Locale;

public interface Doclet extends Serializable
{
    Type getType();
    Identity getIdentity();
    Version getVersion();
    Locale getLocale();
}
```

The `Type` information represents the document type to which the document instance complies. As part of the message sending (Envelope over the wire) the type information is extracted and put in the headers.

The `Identity` is for most documents a globally unique identity. The QoS of identity depends on implementation requirements. `Identity` may for example be used for tracing and audit log keys in databases. The Identity is normally not represented in XML, but is extracted and put in headers when sent.

The `Version` field for Document Instances is a placeholder for the future.

Finally, `Doclet` provides the locale that was defined for the document instance. Observe that we use the `Locale` type provided by JDK. Language, country and variants are available.

Marshaller

The Marshaller interface provides methods to marshal the document representation out to a stream. The operation requires an output stream. A character encoding can be optionally provided. Every programmatic representation of a document provides this capability since it implements the Document interface.

```
package com.commerceone.xdk.metadox.doclet;

public interface Marshaller
{
  void toStream(OutputStream os);
     void toStream(OutputStream os, String encoding);
}
```

UnMarshaller

The `Unmarshaller` interface provides a method for converting a document stream into a document representation. Recall that this function is actually performed by a DocumentFactory, which therefore needs to implement this interface.

To keep the `UnMarshaller` interface clean, the `DocumentFactory` is instantiated with the right `ExternalSource` and `EntityManager` reference. In this manner, the need for multiple factories using different environment setups is not excluded.

```
package com.commerceone.xdk.swi.metadox.marshall;

public interface UnMarshaller
{
     public Document fromStream(Reader reader);
}
```

Type

The interface Type is used to encapsulate type-related information. With this information applications can make processing decisions based on document type. For instance, an application can express that it will only accept PurchaseOrder documents of version "*2.1*" and "*2.2*", defined by owner "*urn:x-commerceone*". This information therefore allows for the management of change as the system evolves.

Note that XML documents and binary documents have different notions of a type. Binary documents only have a MIME type. The MIME type for XML is "*application/xml*". This allows for internationalized XML.

Document type is for XML documents derived from the SOX type. The value for other documents is undefined. A default value of "*Unknown*" is returned in such cases.

Owner is to be able to differentiate between the documents defined by different vendors or consortium. For example, a Purchase Order from Commerce One and the one defined by OBI are not likely to be identical.

```
package com.commerceone.xdk.swi.metadox.type;

public interface Type extends Serializable
{
       public String getOwner();
       public Version getTypeVersion();

       public String getDocumentType();
       public String getMimeType();

       public boolean equals(Object obj);
       public PropertyValue toPropertyValue();
       public ParameterList toParameterList();
}
```

Version

Version is an interface defined in `com.commerceone.util.identity.Version`. Versions apply to document instances and document types. Document representations also use the `Version` interface to represent schema versions.

We assume the following numbered version format: (1.1.32 - Major, minor, revision).

- "1.1" is earlier than "1.2"
- "1.2.233" is a preliminary version.
- "2.2" has the same major version as "2.89"

When creating a version object from a String we assume the `NumberedVersion` implementation. An exception is thrown if the format does not match.

```
package com.commerceone.util.identity;

public interface Version
{
    public String toString();
    public byte[] toBytes();
    public boolean equals(Object obj);
    public int compareTo(Version anotherVersion);
    public boolean sameMajor(Version version);
    public boolean isPreliminary();
}
```

Identity

`Identity` is an interface defined in `com.commerceone.util.identity.Identity`.

For most documents, the `Identity` is globally unique. The QoS of identity will depend on implementation requirements. A user readable format and a machine efficient format is provided.

```
package com.commerceone.util.identity;

public interface Identity extends Serializable
{
    public String toString();
    public byte[] toBytes();
    public boolean equals(Object obj);
}
```

Envelope

Main Abstraction

An `Envelope` is both a `Document` and a wrapper for a `Document`. Along with `Document`, it is one of the two main abstractions in the Document Framework.

An `Envelope` holds exactly one document. Other documents are added and managed as attachments. Attachments can be XML documents or of any other format. Envelopes also have a Property list with key, value pairs, a Context document, and a URI catalog document to resolve references to attachments. Finally, Envelopes can optionally contain a credential for security.

The interface declaration below shows the available methods for `Envelopes`. `Envelope` extends the `Document` interface.

```
package com.commerceone.xdk.swi.metadox.meta;

import java.util.Properties;
import java.util.Enumeration;
import
com.commerceone.xdk.swi.metadox.property.ManagedProperties
;

import com.commerceone.util.net.URI;

public interface Envelope extends Document
{
    public Enumeration getAttachments();
    public Document getAttachment(URI ref);

    public void addAttachment(Document doc);
    public void addAttachment(Document doc, URI id);

    public Document getCredential();

    public ManagedProperties getProperties();

public CatalogReader getCatalog()
}
```

Properties

Envelopes have properties, which can be used for routing and bookkeeping. There are two different types of properties -- managed properties and user provided properties. Managed properties are write once and then read only, user provided properties have no such support. Properties are richer than the default Java properties class as they can have an associated parameter list (like RFC822 headers).

Note that none of the managed properties are set by end-user code!

Property	Who?	When?	Why?
x-Document-Type	Envelope	Envelope creation time	To enable fast routing on the server.
x-Message-Id	Envelope	Envelope creation time	To track messages.
x-Correlation-Id	Service	Service has reply document ready, want to publish back	To track messages, and resulting messages.
x-Request-Mode	Transmitter	Set by application.	To let the sender specify processing hints related to the request. Read more below.
x-Date-Received	Server Agent	When the Envelope reaches the server.	To keep some bookeeping regarding dates. System and legal reasons.
x-Date-Sent	Transmitter closest to the wire. E.g. an InternalPublisher doesn't set this property.	Just before the Envelope is sent over the wire	To keep some bookeeping regarding dates. System and legal reasons.

Property	Who?	When?	Why?
x-Receiver-Id	Transmitter returned from first lookup.	Lookup required receiver info, stored in resulting Transmitter instance. Set by the application.	To make sure the Envelope reaches the right destination. May it be a hosted, or integrated service. Used for routing on the server.
x-Sender-Id	Transmitter returned from first lookup.	Set by application.	To make sure the recipient has enough knowledge to lookup info it needs, e.g. preferred callback address.

The Envelope property `x-Request-Mode` is used to hold processing hints. A hint is meant to override any default values the receiver has stored, or lookup. However hints can be ignored due to transport, or server policies.

A class `EnvelopeHeaders` is prepared to declare constants for the keys of the managed properties.

```
package com.commerceone.xdk.swi.metadox.property;

public interface PropertiesConstants
{
   public static final String REQUEST_MODE_KEY =
    "x-  Request-Mode";
   public static final String SENDER_ID_KEY =
    "x-Sender-Id";
   public static final String RECIPIENT_ID_KEY =
    "x-Recipient-Id";
   public static final String DATE_SENT_KEY =
    "x-Date-Sent";
   public static final String DATE_RECEIVED_KEY =
    "x-Date-Recieved";
   public static final String DOCUMENT_TYPE_KEY =
    "x-Document-Type";
   public static final String MESSAGE_ID_KEY =
    "x-Message-Id";
   public static final String CORRELATION_ID_KEY =
    "x-Correlation-Id";

   public String[] getManagedKeys();
}
```

Application developers can add properties for their own processing. The `EnvelopeProperty` class takes an instance of `EnvelopeHeaders` in its constructor. It uses the value of `getManagedKeys()` to decide what properties are managed and what are user defined.

```
package com.commerceone.xdk.metadox.meta;

public interface EnvelopeProperties
{
   public PropertyValue get(String key, String
     default_value);
   public boolean set(String key, PropertyValue value);
   public boolean set(String key, String value);
}

public interface PropertyValue
{

   public String getValue();
   public ParameterList getParameters();

   public String toString();
}
```

URICatalog Document

The URI Catalog is maintained in the envelope, and is the first data that is used in resolving references from within a document. For instance, the document inside the envelope may refer to an attachment in the same envelope. The catalog helps resolve this relationship.

Attachments

There are two different levels of support for attachments:

1. Attachments may be stored in the Envelope.
2. A URI may be bound to an element in a Document. The URI is used to bind an attachment in the Envelope.

We refer to the latter by Named and Bound attachments. The former is just attachments. An Iterator on the Envelope is provided when named and bound attachments are not used.

Client Side Usage

On the client side, the programmer is concerned with creating the envelope and sending it to the business partner for processing by a business service. Where a Named and Bound attachment is used, the developer needs to set the reference attribute on the element. In that operation use a URI that is later used when adding the attachment to the envelope. Note that this is a very weak binding, but is the preferred choice.

In this example we are using the DocumentObject representation. We create the PurchaseOrder document and set some elements. We create a LineItem, introduce a reference to an external attachment, and add the line item to the purchase order. We now create a DocumentStream document that will contain an MicroSoft Word file. We create the envelope with the PurchaseOrder as the argument to the constructor. We then add the attachment using the URI we used in the line item

.

```
// Create Bean/
PurchaseOrder po = new PurchaseOrderImpl;
EnvelopeFactory ef = … // Get from transmitter/service
po.setAddress(…); // Document specific manipulation
…

LineItem line_item = new LineItemImpl();

// Associate the attachment with the line item element
// Reference is the name used in the SOX for this attribute
line_item.setReference("urn:customsdecl");

po.setLineItem(line_item);

// Read Attachment Document from Disk
FileInputStream fis = new FileInputStream("file.doc");
InputStreamReader reader = new InputStreamReader(fis);
DocumentStream attachment =
new DocumentStream(new ReaderDataSource(reader),"application/
ms-word");

// Create Envelope. Add a named and bound attachment
Envelope env = new ef.creatEnvelope(po);
env.addAttachment(attachment, "urn:customsdecl");

// Send Envelope
DocumentResponder responder = …;
Envelope reply_env = Responder.processDocument(env);
```

Service Side Usage

All document services implement the method `handleDocument()` from the DocumentListener interface. This method gets the same Envelope that was created at the client. A few more properties might have been added along the way.

```
void handleDocument(Envelope env)
{
   // Document is here as an unparsed DocumentStream
   Document doc = env.getDocument();

  // Keep one of the following 4 lines depending on
   programming model expected.

  // DocumentStream doc_stream =
     DocumentStream.createFrom(doc);
     PurchaseOrder po = PurchaseOrder.createFrom(doc);
     LineItem line_item = po.getLineItem();

  // Reference was the name used in the SOX description
     URI ref = line_item.getReference();
  // Get Attachment from Envelope
     DocumentStream attachment = env.getAttachment(ref);

  // Or, iterate over Attachments
     Enumeration attachments = env.getAttachments();
     while (attachments.hasMoreElements())
{
        Document att_doc = attachments.getNextElement();
}}
/ /}}
```

Externalization and Internalization

Each document and document wrapper knows how to externalize itself - this is given from the `Marshaller` interface. An envelope sent over a wire becomes a message. The message has an identity equal to the envelope id. The format of the message is Multipart MIME with RFC822 headers, where properties are translated into headers. The URI Catalog and each document, including attachments, becomes a part. A signed Document is represented as a multi-part with the document and the signature as XML documents are XML-ified, attachments get a native representation.

The following example illustrates how properties and documents have been packaged in a MIME message. The example also shows how an attachment of type MS-Word is referenced from within the element `LineItem`, and listed in the URN catalog for later resolution. The attachment is an MicroSoft Word document.

```
Message-ID:
<220.925008744770.JavaMail.kenneth@xke.commerceone.com>
Date: Sat, 24 Apr 1999 19:51:12 -0700 (PDT)
Subject: PO - 8a42d0a0-8783-0000-027f-000001234000
x-Message-Id: 8a42d0a0-8789-0000-027f-000001543000
x-Receiver-Id: urn:x-commerceone:tradingpartner:4711 (Office Depot)
x-Sender-Id: urn:x-commerceone:tradingpartner:3092 (Commerce One)
x-Document-Type: urn:x-commerceone:PO:1.0
Content-type: multi-part/related; boundary="-----";
Mime-Version: 1.0
Content-Description: Commerce One Envelope Message Format
version 1.0

-----
Content-type: application/xml; owner=x-commerceone type=PO
version=1.0
Content-ID: 8a42d0a0-8783-0000-027f-000001234000
Content-Description: Commerce One Document Format XML/SOX

<?xml version="1.0"?>
<!DOCTYPE PurchaseOrder SYSTEM
"urn:xcommerceone:sox:PurchaseOrder:1.0">
```

```
<PurchaseOrder>
<LineItem attachment="urn:customsdecl">
        Calculator
</LineItem>
</PurchaseOrder>

-----
Content-type: application/xml; type=catalog
Content-ID: 8a42d0a0-8782-0000-027f-000001567000


<?xml version="1.0"?>
<catalog>
<map urn="urn:customsdecl" to="uuid:8a42d0a0-8782-0011-027f-
000001987000"/>
</catalog>

-----
Content-type: application/ms-word
Content-ID: 8a42d0a0-8782-0011-027f-000001987000
Content-Disposition: attachment; filename=hippo.doc
Content-Description:

Lshfakshflkjasdfkv cxz BDyqFKS>JFH;kjZBC
DHF;HSDF:jhZXLJNF;DHLKJNflHF
…………

-----
```

Sending and Receiving Documents

Interfaces

Part of the Document Framework is the definition of the interfaces that are used to send and receive documents and envelopes. This is to achieve a common style for exchange of information. Third party application developers are responsible for the development of classes that implement these interfaces.

The Document Framework interfaces are

- `DocumentListener` (for a one-way outbound)
- `DocumentResponder` (for two-way)
- `DocumentServant` (for one-way inbound).

Exceptions

The framework comes with an exception hierarchy. Programmers can catch and handle fine-grained exceptions or a category of exceptions. Exceptions related to document exchange inherit from the exception class `DocumentExchangeException`.

The sub-category `EstablishException` is used for exceptions related to the establishment of the connection -- Bad URL, Connection unavailable, Access denied and more. The exception is always raised in system code. Completion code is always `NO`.

The sub category `TransferException` is used for exceptions related to the transfer of information. A connection was established, but information did not reach the intended service. The exception is always raised in system code. Completion code is always `NO.`

The sub category `ProcessingException` is used for exceptions related to the processing of information. This is typically related to marshalling and unmarshalling. The exception is always raised in system code. Completion code is always `NO` if the error happened at the server. If the error happened in error code on the way back, the completion code can be `YES` or `MAYBE`.

The sub category `ServiceException` is used for exceptions related to business level problems. The exception is always raised in application code. Completion code is defined by application semantics. Unhandled exceptions in services are caught and transferred back accordingly.

```
package com.commerceone.xdk.excp.metadox.send;

public class DocumentExchangeException extends
DocumentException
{
   public static final int YES=1;
   public static final int NO=2;
   public static final int MAYBE=3;

   public final int getCompletionStatus()
}

// Something went bad when setting up the connection
public class EstablishException extends
DocumentExchangeException {}

// Something went bad sending the information
public class TransferException extends
DocumentExchangeException {}

// Something went wrong processing the information sent
public class ProcessingException extends
DocumentExchangeException {}

// Something went wrong in the User Defined Service
public class ServiceException extends
DocumentExchangeException {}
```

DocumentListener

The DocumentListener interface is used for the outbound passing of envelopes between entities. The entities can be object, services, or servers. It can be local or remote. It can be queued or not.

The state and semantics of the implementation decide the appropriate action. A user only cares about the interface.

```
package com.commerceone.xdk.metadox.send;

public interface DocumentListener
{
    public void handleDocument(Envelope env)
      throws DocumentExchangeException;
}
```

DocumentResponder

The DocumentResponder interface is used for the two-way passing of envelopes between entities. The entities can be object, services, or servers. It can be local, or remote. Since a return envelope is expected this interface is only implemented for express, synchronous communication.

The state and semantics of the implementation decide the appropriate action. A user only cares about the interface.

```
package com.commerceone.xdk.metadox.send;

public interface DocumentResponder
{
public Envelope processDocument(Envelope envelope)
      throws DocumentExchangeException;
}
```

DocumentServant

The DocumentServant interface is used for the inbound passing of envelopes between entities. The entities can be object, services, or servers. It can be local or remote. Since a return envelope is expected, this interface is only implemented for express, synchronous communication.

The state and semantics of the implementation decide the appropriate action. A user only cares about the interface.

```
package com.commerceone.xdk.swi.metadox.send;

public interface DocumentServant
{
    public Envelope getDocument()
       throws DocumentExchangeException;
}
```

Document Wrappers

A set of DocumentWrapper classes is available, whose only purpose is to wrap a document in an intermediate wrapper, to allow routing based on a well-defined template instead of the actual document type. A Reply wrapper has no meaning if no service subscribes to Reply documents, knows how to unpack them, and how to handle the wrapped document. The same applies to Forward and Store wrappers.

Reply

When the application has a reply document for the request, it wraps the document in a Reply wrapper and publishes it back to the same router it subscribed to. This is not used by application services directly.

```
package com.commerceone.xdk.metadox.dox;

public class Reply extends DocumentWrapper
{
      public Reply(Document doc);
}
```

Forward

This can be used to wrap a document for forwarding within a server, or between servers. If no service accepted a document, the router wraps it in the Forward wrapper and publishes it again. By default, a Lost and Found service picks up forwarded documents. A smart service can try to find another router in the system that can route the forwarded document. This is not used by application services.

```
package com.commerceone.xdk.metadox.dox;

public class Forward extends DocumentWrapper
{
    public Forward(Document doc);
}
```

Store

This can be used to wrap a document that is supposed to be stored, such as when a service wants a document stored but does not want to be responsible for the task. A Storage service can be installed to take care of storing. Wrapping a document in a Store document just indicates a desire to have the document stored.

```
package com.commerceone.xdk.metadox.dox;

public class Store extends DocumentWrapper
{
public Store(Document doc);
}
```