COMMERCE ONE

XDK version 1.0

Developer's Guide

Commerce One XDK Version 1.0 User's Guide

1 INTRODUCTION **3**

1.1 OTHER USEFUL DOCUMENTS 3

2 WHAT IS SOX? **4**

2.1 SOX VERSUS DTDS 4
2.2 MORE ON SOX FEATURES 4
2.3 DEFINITIONS 5
2.4 WHAT TO READ NEXT 5

3 HOW TO USE THE COMMERCE ONE XML PARSER (CXP) **6**

3.1 WHAT IS CXP? 6
Validating an XML Instance against a DTD 6
Validating a SOX Schema 6
Validating an XML Instance against a SOX Schema 6
3.2 CLASS PATHS 6
3.3 CXP OPTIONS 6
3.3.1 Examples of Using CXP 8
3.3.2 Example of Using a Sample File With CXP 9
3.4 HOW TO USE SCHEMA PATHS AND URNS 9
3.5 HOW TO USE CATALOGS 10
3.6 INTERPRETING ERROR MESSAGES 11
3.6.1 General Validation Errors 11
3.6.2 Encoding Errors 12

4 INTERFACING WITH CXP VIA SAX **12**

4.1 SIMPLE EVENT API FOR XML (SAX) 12
4.2 CXP AND SAX 13
4.3 USING CXP WITH SAX 13
4.4 SAX SAMPLE 15
4.4.1 EXAMPLES 16
4.5 CXP AND XT 16
4.5.1 Examples: 17

REFERENCES **17**

1 Introduction

The entire XDK Team would like to thank you for downloading the Commerce One XDK (XML Development Kit). We recommend that you to take a few minutes to skim this document before using our software.

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 **Section 1.1** for a pointer to XML documents.
- If you have XML experience, but do not know SOX, then you should start with **Section 2**, and then read some of the other documents referred to in **Section 1.1**, before going on to the remaining sections in this document.
- If you have XML and SOX experience, and would like to learn more about our parser and SOX validation, then you can go directly to **Section 3**.
- If you have XML and SOX experience, and would like to develop applications on top of our software, then **Section 4** is for you.

1.1 Other Useful Documents

For more information on XML, read the XML 1.0 specification.

We recommend that you read the document SOXTutorial.pdf, (found in this package),to get a tutorial on reading and writing 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 in this package).

For more details on how to use SAX, go to http://www.megginson.com/SAX/ and http://www.megginson.com/SAX/SAX2/. Here you will find complete descriptions of SAX 1.0 and SAX 2.0 respectively.

For more information on XSL, go to http://www.xml.com/pub/Guide/XSL.

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

2 What is SOX?

A schema is defined as a set of rules that defines the structure of a document. A document Type Definition (DTD[1]), is a particular type of a schema language that is used to define XML documents. Given a schema (or a DTD), one can create instances of XML documents that conform to that schema. This conformance of an XML instance document to a schema can be checked automatically via a validating parser.

The Schema for Object-Oriented XML (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 has been designed as an easy to use alternative to DTDs, that also supports the needs of <u>any</u> highly decentralized environment (for example the Internet).

2.1 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, 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 document type definitions via namespaces. This means that the definitions in one schema can reuse the definitions in other schemas by importing these other schemas. In contrast, DTDs do not have namespace support. This is a problem when DTDs are scaled to thousands of market places around the world.

2.2 More on SOX Features

Commerce applications require the ability 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 is **string**. While it is possible to build specialized mechanisms to allow parties to tell each other the data type of an attribute, 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.

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

[1] See the XML Specification for greater detail.

support in document instances, is safer from a programming point of view and 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.

2.3 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 program called a validating parser tests a document's validity.

- **SAX** refers to the Simple event API for XML.

- **XT** refers to James Clark's implementation of an XSL processor, in Java.

2.4 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, (found in this package),to get a tutorial on reading and writing SOX documents. Once you have read this document, you should be well equipped 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 in …/commerceone/xdk/sample.

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

3 How to use the Commerce One XML Parser (CXP)

3.1 What is CXP?

CXP is a validating XML 1.0 and SOX 2.0 Parser. It can be used to validate document instances against DTDs that are XML 1.0 compliant, or schemas that are SOX 2.0 compliant. It can also be used to validate schemas against the SOX 2.0 specification encoded in a DTD. CXP will automatically recognize whether a document instance should be parsed as an XML 1.0 instance of a DTD, a SOX 2.0 schema document, or a 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. If a SOX schema is used as input, CXP will do a SOX-level validation of the schema document in addition to validating the schema against its DTD.

Here are the three 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 will be 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.

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

Validating a SOX Schema
If you pass CXP a SOX schema that starts with the header below, it will be 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 will be validated as a SOX 2.0 schema to ensure that it conforms to the SOX restrictions that the DTD cannot check.

```
<?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 will be validated against the schema CBL.sox as an instance of the SOX 2.0 schema specified in `CBL.sox`.

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

3.2 Class Paths

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

3.3 CXP Options

CXP should be run with the provided script, cxp.bat, that resides in the /../commerceone/xdk directory. This script will set the classpath system variable properly.

Running CXP with no options will present 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 CXP options are:

`-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 search for the SOX schema or DTD that the input document will be validated against. This option is ignored when a catalog is specified (see the `-c` option). If neither this option, nor the `–c` option, is specified, CXP will assume 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.

`-c <catalog-system-id>` The catalog-system-id causes CXP to load an URN catalog. An 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.

`-o <output file>` CXP sends the output to the file specified.

`-n <number>` The number of times CXP parses each document.

`-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 can for example not 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 if this option is specified. This option does not take an argument. This option has no effect when validating against 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.

`-f` By default CXP assumes that the input document name provided is a URI. However, when the input document name does not contain a colon, ":", CXP will recognize that the document name must be a file. However, in those cases where the filename contains a colon, ":", such as c:/schemas/TestSchema.sox, you have to tell CXP to interpret this name as a file, rather than a URI. This is done with the `-f` option. For example, given the file `po.xml`, you can use either "`cxp \commerceone\po.xml`" or "`cxp -f c:\commerceone\po.xml`". The `-f` 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).

3.3.1 Examples of Using CXP

Let's assume that the sample document below is located at `c:\testdocs\langstring.xml`, and let's assume that its schema is located at `c:\mywork\com\commerceone\CBL\n1_0\CBL.sox`.

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

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

- Example 1:

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

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

- Example 2:

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

- Example 3:

  ```
  > cxp -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:

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

This will give you more information about instance validation. Sample output:

```
cxp.INFO:ns="urn:x-
commerceone:document:com:commerceone:CBL:CBL.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 a definition belongs to namespace 1. This content model says that the document can have a root tag of `"LangString"`.

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

3.3.2 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
`$(HOME)/commerceone/xdk/sample/xdk/sox/n1_0`.
The sample instances of these schemas, are located in several directories under
`$(HOME)/commerceone/xdk/sample/xdk/instances/`.

For example, corresponding to the SOX schema file
`$(HOME)/commerceone/xdk/sample/xdk/sox/n1_0/Film.sox`,
there is an XML instance document
`$(HOME)/commerceone/xdk/sample/xdk/instances/basic/Film.xml`.

To validate `Film.xml` against the schema it conforms to, go to the directory
`$(HOME)/commerceone/xdk`, and type the following on the command line:

```
cxp.bat -p $(HOME)/commerceone/xdk sample/xdk/instances/basic/Film.xml
```

3.4 How to Use Schema Paths and URNs

As discussed previously, an XML instance of a SOX 2.0 schema will always start 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 URN. The same URN format is used for namespace imports.

In the following example, let us assume that the schema that is referred to is contained in the 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 the scheme for the mapping, and must be used verbatim in order for the mapping to function properly.

2. The part after the scheme works in a manner similar to the Java classpath mechanism with the addition of the "`$1.0`" token to indicate a version. The version token in this release must always be "`$1.0`" and is a placeholder for version functionality in later releases.

3. 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.

4. 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 Section 3.2. The root is represented as (ROOT) in this example.

5. 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.

6. 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 <u>must</u> 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`".
7. 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 part of the schemapath option `-p`.

3.5 How to Use Catalogs

If you do not want to use the URN mapping mechanism described in **Section 3.4** 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 a 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/sample.sox"/>
</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 as follows.

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

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

Note that you need 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 both URIs 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 will now include the `other_catalog.xml` catalog file. This enables multiple catalogs to be used in one parsing. Note that 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.

3.6 Interpreting Error Messages

3.6.1 General Validation Errors

1. 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 as well 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.

3.6.2 Encoding Errors

When 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 will not be able to process it correctly.

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

4 Interfacing with CXP via SAX

4.1 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 SAX 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[2] and handles the SAX events produced by the parser in the handlers[3] defined in SAX specifications. James Clark's XSL processor XT is an example of such an application. XT is provided in this installation.

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

[2] A driver is a class made available by the parser. It implements some or all of SAX APIs.
[3] A handler is a class that implements a SAX handler interface such as DocumentHandler, ErrorHandler, or DTDHandler.

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.

4.2 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
- `Parser.parse(InputSource)` - Only InputSource instances created from a systemID will work. Reader and InputStream versions are not implemented.

CXP implements the SAX 2.0 Configurable interface. This interface allows a user to set parser features and properties. The provided methods are `set/getFeature`, and `set/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 feature that can be `set/get` is:
"http://xml.org/sax/features/validation", the associated value is of type boolean, and will turn validation on or off. A value of `true` will turn validation on, a value of `false` will turn validation off.

The properties that can be `set` are:

- "http://commerceone.com/sax/properties/schemapath" – the associated value is of type String, and sets the schema path used by the parser.
- "http://commerceone.com/sax/properties/catalog" – the associated value is of type String, and sets the catalog URI used by the parser.
- "http://commerceone.com/sax/properties/catalogs" – the associated value is of type Vector, populated with Strings, and 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.

4.3 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;
```

2. Implement 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( "http://xml.org/sax/features/validation",
validation );

// getting the validation mode
```

```
validation = configurable.getFeature(
“http://xml.org/sax/features/validation” );

// set schema path
String path = “/commerceone/xdk/xml/;/myschemas”;
configurable.setProperty(
"http://commerceone.com/sax/properties/schemapath", path );

// set catalog
String catalog = “file:///d:/mycatalogs/catalog.xml”;
configurable.setProperty(
"http://commerceone.com/sax/properties/catalog", catalog );

// set catalogs
// 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(
"http://commerceone.com/sax/properties/catalogs", 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.

4.4 SAX Sample

A functional SAX sample can be found in `commerceone\xdk\sample\apps`. 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).
- Attaches a document and an error handler to the parser.
- Starts parsing the XML document.
- 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:

```
jview SAXTest –p:schema_path –c:catalog –v 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.

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

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

Note that in the case that 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` will be ignored.

4.4.1 Examples

The following examples assume that you are using the Microsoft® Java VM `jview`.

```
>jview SAXTest file:///testdoc.xml
>jview SAXTest –p:/schemas;/commerceone/xdk /testdoc.sox
>jview SAXTest –c:/catalogs/catalog.cat -v ../myschemas/testdoc.sox
```

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`.

```
>jview /d:org.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.

4.5 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, 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 …/commerceone/xdk directory, that you can use to run this class.

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

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 (**Section 3.3**).

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

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

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

4.5.1 Examples:

```
jview com.commerceone.xdk.standards.xsl.XSL /xml/file.xml /xml/file.xsl

jview com.commerceone.xdk.standards.xsl.XSL –p /schemas/myschemas
../file.xml ../file.xsl

jview com.commerceone.xdk.standards.xsl.XSL –c /catalogs/catalog.cat
/work/file.xml /work/file.xsl
```

In order for these 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 …/commerceone/xdk/lib. A batch file, `cxsl.bat` 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.

References

[XML99] Koistinen et al., “XML Programming Models for Electronic Commerce Systems,” to appear in XML’99.