Commerce One MarketSite™

XML Commerce Connector™ Developers Guide

Version 1.1

Hacienda Business Park, Bldg # 4
4440 Rosewood Drive
Pleasanton, CA 94588

XML Commerce Connector Developers Guide

Commerce One MarketSite™

Version 1.1

March 2000

Contents

Preface

MarketSite is an extranet-based system that automates both the supply-side and buy-side of the business procurement process, from order receipt to payment. It consists of interconnected data centers around the world. Each data center, or *MarketSite Domain*, provides a common set of business and infrastructure services.

This manual describes how to create integrated applications that interact with a trading partner marketplace and services that reside on that marketplace.

Who This Manual is For

This manual is intended for developers that interconnect existing or new applications to MarketSite. It provides an overview of the XML Commerce Connector components necessary to define and implement a new trading partner application service and client.

Before You Start

This guide assumes you:

- Are knowledgeable in Java development.
- Are knowledgeable of the Extended Markup Language (XML) and the Schema for Object-oriented XML (SOX).
- Are knowledgeable of XML to Java Mapping, i.e., the conversion of an XML marked-up document to a JavaBean using a SOX schema incorporating a Document Type Definition (DTD).
- Know how to start and stop XML Commerce Connector (XCC) services.
- Are familiar with the concepts of electronic commerce and want to create an application that integrates with CommerceOne's MarketSite.

For information on installing and configuring the services and clients, please see the ***XCC Installation and Configuration Notes.*** For documentation of the XCC classes and interfaces, refer to the ***JavaDocs*** included with the XCC installation.

System Requirements

Before you install XCC, verify that you have the system requirements listed in this section.

Install the following third party software packages in the order in which they appear in this list:

- **Windows NT 4.0** with Service Pack 5 (SP5)
- **Microsoft Java Virtual Machine** (JVM) jview version 5.00.3181+ or Microsoft SDK (version 3.2 or higher). You can obtain this from:

  `http://www.microsoft.com/java/download.htm`

- **Cygnus Cygwin package,** a Win-32 port of the GNU development tools and utilities. The Cygwin library provides a UNIX-like API on top of the Win32 API. This tool is included on the product CD ROM.

See the ***XCC Installation and Configuration Guide*** for complete installation prerequisites and information.

This book assumes that you have used these products and are familiar with common Windows and web browser terminology, use of the mouse, and so on. If not, you should contact your system administrator and obtain the necessary documentation and training.

Document Conventions

Note .....Used to indicate items of particular interest.

Caution.....Used to indicate an item, which if ignored, may cause errors or other problems.

Warning! Used to indicate an item which is extremely important.

Text in `monotype` is used to indicate file names or code.

In code examples, values you must supply are shown as `<value>`.

Glossary entries are in italic.

Book names are in bold italic type.

Acronyms

Acronym	Meaning
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
SAX	Simple API for XML
SOX	Schema for Object-oriented XML
SSL	Secure Sockets Layer
X2J	XML to Java
XCC	XML Commerce Connector
XML	eXtensible Markup Language

Document Chapters

This manual includes the following sections:

Chapter 1 "Getting Started With XCC"

This chapter provides an introduction to the XML Commerce Connector (XCC) Document and Service Frameworks and the Schema for Object-oriented XML (SOX).

Chapter 2 "Using Documents"

Describes how to use the *Document Framework*, a collection of Java interfaces, to create, read, and update documents and envelopes.

Chapter 3 "Writing an Application Service"

Use the *Service Framework* to integrate applications with XCC. You can use the classes provided or extend them as needed. You then install this service on an XCC Server, enabling you to receive and send documents within your market place.

Chapter 4 "Understanding Clients"

This chapter describes how to create both stand-alone clients and service clients for creating and receiving XCC documents.

Appendix A "Glossary"

This appendix contains brief explanations of the many technical terms describing the trading partner services and clients.

Appendix B "References"

This appendix contains a list of key technical references.

Chapter 1
Getting Started With XCC

MarketSite is an extranet-based system that automates both the supply-side and buy-side of the business procurement process, from order receipt to payment. It consists of interconnected independent marketplaces around the world. Each independent marketplace, or *MarketSite Domain*, provides a common set of business and infrastructure services.

This manual describes how to connect applications that interact with a trading partner marketplace and services that reside on that marketplace.

About XML Commerce Connector

XML Commerce Connector (XCC) enables you to integrate your company's applications and data into a CommerceOne MarketSite-based electronic commerce marketplace.

XCC has the following main components:

- **XML runtime** This processor validates XML document structure and converts XML documents to Java™ (X2J) objects.
- **Service Framework** This set of Java abstract classes and interfaces enables you to develop document-centered applications that receive and respond to MarketSite messages.
- **Document Framework** This set of Java classes and interfaces enables you to create envelopes, insert documents into envelopes, and add attachments. This framework enables MarketSite applications and business services to route documents without the need to "understand" their content.
- **XCC Server** The server contains message dispatching, receipt, and security components necessary to receive document objects.
- **Communication component** The communication layer enables an XCC-based application to receive and send MarketSite messages using supported transport protocols

XML Commerce Connector (and the MarketSite architecture) relies on several basic principles:

- **Open Interface** All messages are exchanged using the Multipurpose Internet Mail Extensions (MIME) protocol. The content of the messages, or business documents, are created using eXtensible Markup Language (XML). These two standards make the MarketSite architecture open and enables you to integrate applications and data easier.
- **Secure Protocol** A MarketSite network uses a secure protocol, the Hypertext Transfer Protocol Secure (HTTPS), to exchange data, ensuring the integrity and privacy of your information.
- **Authentication** XCC authenticates the origin of messages.
- **Store and Forward** MarketSite uses a store and forward methodology for document exchange. This model is suitable for reliable transport over the Internet, but does not provide real-time characteristics.
- **Synchronous or Asynchronous Communication** Document exchange between trading partners and MarketSite can be synchronous or asynchronous. You will choose which depending on your application's requirements and characteristics.

For example, to issue a purchase order, an asynchronous exchange is most appropriate. The transactions are long and there are no requirements for immediate (real-time) response to a purchase order. In other instances, a real-time or synchronous, response is required. You determine the calling semantics when the message is created.

The XCC Architecture

The XML Commerce Connector (XCC) is a web client and optionally a web server that has been extended to support secure exchange of XML documents.

Figure 1.1 below illustrates the overall XCC architecture and how the components are related. The figure shows the XCC in two configurations.

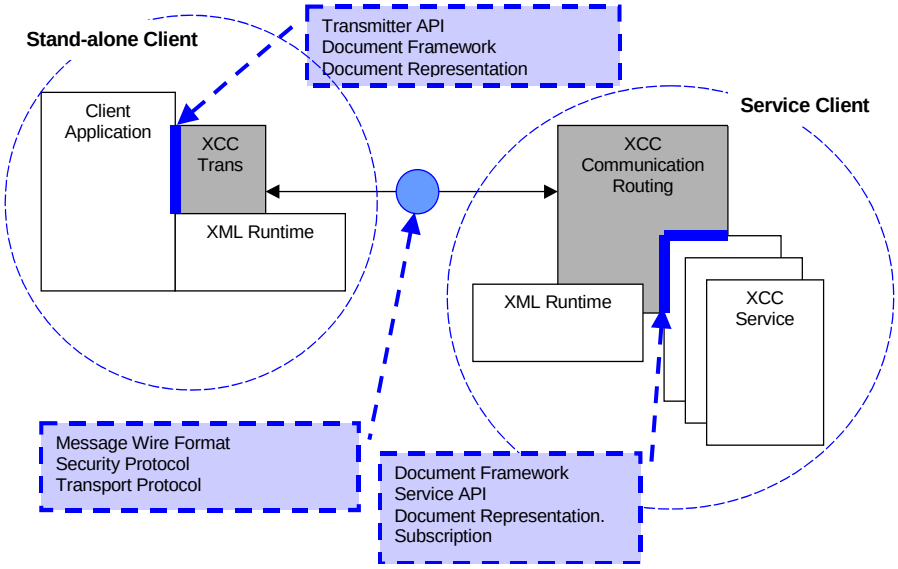

Figure 1.1 XCC Architecture

The left circle contains a stand-alone client configuration, which means that no services are used. In this configuration only the integrated application can initiate communication.

The right circle contains a service client, a full configuration using the service framework to run services. With this configuration an application integrated with XCC can both receive and send messages. This application service contains a client.

As previously described, XML Commerce Connector (XCC) has four main components:

- a service framework
- a document framework
- an XML runtime subsystem, and
- a XCC transmitter or communication subsystem.

Building Application Services and Document Services

Trading partner applications can be used to create document instances, to send them, and to receive replies to previous documents transmitted. A client can be written as a stand-alone client or run as an client service.

- **Application services** Application services reside on a XCC Servier. They receive and respond to messages from clients, subscribe to specific types of documents, and tell the XCC platform how to route its messages. When implemented, they give you the capability to create document instances, create envelopes, insert documents into envelopes, add attachments, and route those envelopes as messages to a XCC client.
- **Service client** A service client is started, managed, and stopped using the XCC Server. This service can handle many transactions on a routine, hands-off basis. Service clients have access to all of the capabilities of the XCC Server, for example, service thread pools, synchronous event logging, routing, and asynchronous reception of envelopes.
- **Stand-alone client** A stand-alone client runs outside an XCC Server. It can synchronously transmit envelopes and receive responses to them or asynchronously transmit envelopes.

  However, stand-alone clients cannot receive envelopes.

Sample XML Documents Included in XCC

The XCC Server contains the full implementation of the Commerce One Common Business Language (CBL). CBL 2.0 is a set of XML building blocks and a document framework that enables you to create robust, reusable, XML documents for electronic commerce. XCC contains the a working example of a sample XML business documents, ready for you to use or extend for your own purposes. These include a sample PriceCheck application which we recommend you use to experiment and model on.

Compile the Sample PriceCheck Service

Before beginning to write your own service, we suggest you compile and experiment with the PriceCheck application to become familiar with a working application.

Note .....This section assumes you have already completed all of the steps outlined in the *XCC Installation and Configuration Guide*. This includes installing and configuring the Microsoft Visual J++ Build Environment and the Cygnus UNIX emulation environment. See the *XCC Installation and Configuration Guide* for complete information.

Customizing Your Installation

If you have installed XCC on a drive other than "C", additional manual configuration is required. For example, you must manually edit all sample application Java files and manually change the drive from C: to the correct drive. The sample Java files are found in several directories under the following default parent directory:

```
drive-letter:\commerceone\xcc\sample\com\commerceone\sample\xcc
```

You must edit each of the following files and change all references from C:\ to the drive in which you installed XCC.

```
AsyncPurchaseOrderClient.java
AsyncPurchaseOrderClientService.java
AsyncPurchaseOrderService.java
PriceCheckRequestService.java
PriceCheckTransmitter.java
PurchaseOrderService.java
PurchaseOrderTransmitter.java
```

Start the XCC Server

You should verify that the XCC Server is running correctly. If the XCC Server is already running and you have already verified that the it is running correctly by running `pinginstall`, skip to **Step 1 on page 1-6** below.

To run the `pinginstall`, first stop and restart the XCC Server.

1. In a Cygnus window, run `pinginstall` to verify that the XCC server is running corrrectly. Type:

   ```
   cd $XCCROOT/bin
   ```

2. Then enter:

   ```
   pinginstall
   ```

A large amount of text is displayed; when complete and successfull, it finishes with pong and the name of your machine along with its IP address, as in the following example

```
Pong . [Host:mv-machinename.commerceone.com/127.0.0.1]
```

Once the `pinginstall` is running correctly, you must stop and restart the XCC Server.

3. To stop and restart the server, open the Control Panel. Select the Services applet. Scroll down, if necessary, until you find the service named "CCSNTService." Stop and then Start the service again. Exit the applet.

Now you can compile the new document services

Compile and Copy the Sample Application

You can find the sample PriceCheck service in package

```
com.commerceone.sample.xcc.pricecheck
```

The path to the class files is:

```
$XCCROOT\sample\com\commerceone\sample\xcc\pricecheck
```

1. To compile the sample applications, open a Cygnus window and change to the root sample directory. Type:

   ```
   cd $XCCROOT/sample
   ```

   Note ..... If you choose to type in the actual path, be sure to substitute your actual XCC root path for the environmental variable $XCCROOT.

2. Then type:

   ```
   make
   ```

   If the make fails for any reason, double-check your workstation's PATH statement, as previously described, to make sure it is correct.

3. Edit the file:

```
$XCCROOT/runtime/servers/defaultserver/config/startup/service-start.prop
```

   a) To add the PriceCheckService to the startup services, find the first occurance of "`startup.application.services`". Add a comma

and Service to the end of the statement, as shown below.

```
startup.application.services=PingService,DispatchAssistantServiceImpl,PriceCheckServi
ce
```

b) In the same file, locate the next occurance of "service.PingService". Copy this and the next line immediately below. Then modify the lines as shown below, or cut and paste the two lines of code below.

```
service.PriceCheckService.code=com.commerceone.sample.xcc.pricecheck.PriceCheckServic
e

service.PriceCheckService.args=initialThreads=0,maximumThreads=9,queueCode=com.commer
ceone.ccs.kernel.queue.ThreadedEnvelopeQueue
```

Note ..... Each of the above code entries should be on one continuous line.

c) Exit the `service-start.prop` file.

4. Stop and restart the XCC server, open the Control Panel. Select the Services applet. Scroll down, if necessary, until you find the service named “CCSNTService.” Stop and then Start the service again. Exit the applet.

5. Select the open Cygnus window. Change to the XCC root directory. Type:

```
cd $XCCROOT/bin
```

6. Copy the `pingtest` file to create a new file, `pctest`. Type:

```
cp pingtest pctest
```

7. Edit pctest. Type

```
notepad pctest
```

8. Change

```
"${XCCROOT}/bin/launch" com.commerceone.ccs.service.ping.PingTransmitter $*
```

to

```
"${XCCROOT}/bin/launch" com.commerceone.sample.xcc.pricecheck.PriceCheckTransmitter
$*
```

Note ..... The above entry should be on one continuous line.

9. Exit the `pctest` file.
10. Run the PriceCheck client. In the Cygnus window, type (or cut and paste):

```
pctest -d abc -authpref uidpswd -userid abc -password abc -recipient abc -m abc -
timeout 10000
```

The above entry should be on one continuous line. Proper execution displays the message:

```
Got PriceCheckResponse from PriceCheckService
```

in the Cygnus client window. This indicates that your service is running correctly.

See Chapter 4 "Creating Clients" for information on how the client service works.

Chapter 2
Using Documents

In this Chapter

This chapter describes how to use the *Document Framework*, a collection of Java interfaces, to create, read, and update documents and envelopes.

Using the Document Framework

The *Document Framework is* a collection of Java interfaces. You use it to create, read, and update documents and envelopes. A *document* in this context contains an XML representation of information. An *envelope* contains a single document and zero to many attachments.

A document is the base abstraction in the Document Framework. Each document has type, identity and other key attributes. When you send an envelope, it becomes a message.

A message can be:

- An envelope object containing a single XML document.
- An envelope object containing a single XML document and one or more *attachments*.
- Two or more nested envelope objects containing a document or documents.

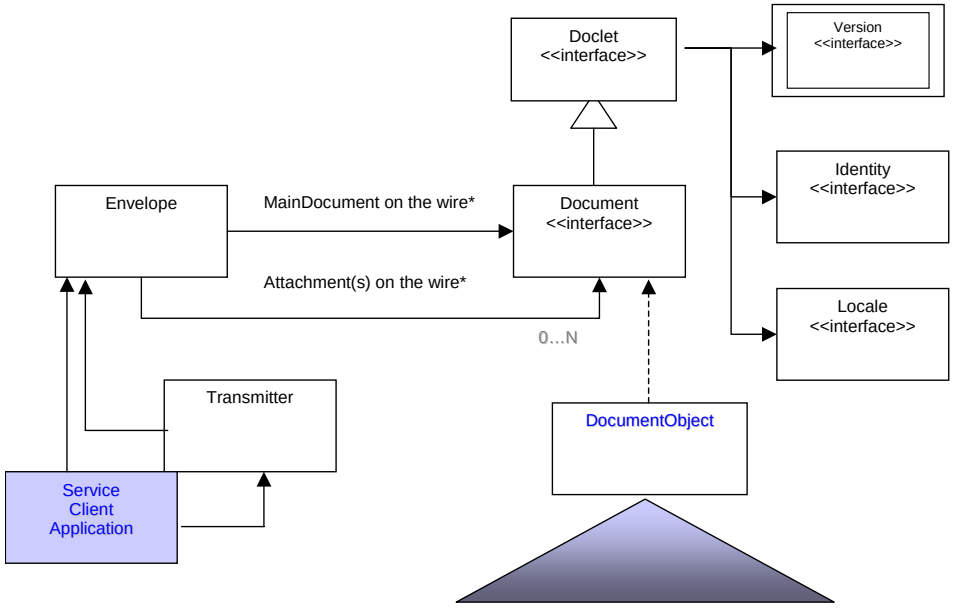

Figure 2.1 Document Framework. (* Documents on the wire are illustrated in Figure 2.2 on page 2-3.)

Using the Envelope Interface

`Envelope` is one of two main abstractions in the Document Framework. The other is described in "Using the Document Interface" on page 2-4.

An `Envelope` holds only one primary document. Other documents can be added and managed as one or more attachments. Attachments can be XML documents or other formats unknown to XCC (typically binary documents of various MIME types).

Envelopes contain a property list with key/value pairs, a context document, and a Uniform Resource Identifier (URI) catalog document. The last is used to resolve references to attachments. Finally, Envelopes can optionally contain a security credential.

Available `Envelope` methods are:

`getProperty()` `getAttachment()` `getProperties()`

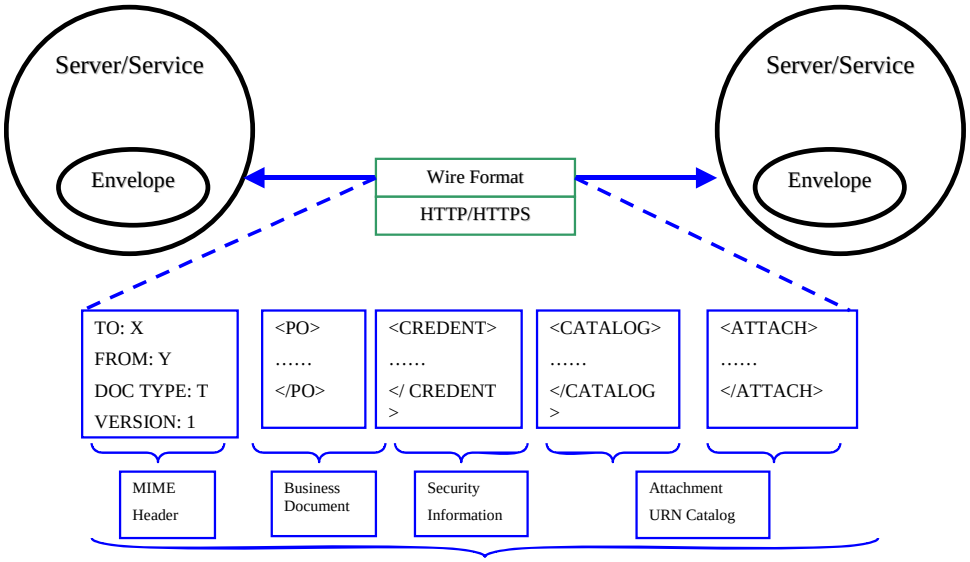

Figure 2.2 Message wire protocol

```
setProperty()       addAttachment()     getDocument()
getAttachments()    getCredential()
```

See the JavaDoc included with the installed XCC executable for more complete information.

You create a wire message by constructing an envelope. The envelope wire protocol construction is shown in Figure 2.2 . Envelope headers typically identify the receiver, sender, and document type. Each document typically contains a version, a type, an identity, a locale, and so forth. An example of a document is a PurchaseOrder. XCC does not need to know what the content of the document is to route it to the correct application. XCC only needs to be able to read the header data in the envelope.

Figure 2.1 on page 2-2 illustrates the Document Framework. All envelopes contain a MIME header, a business document, and security information. They may contain catalog information, or null catalog content if empty, and envelopes may contain one or more attachments.

The envelope and the MIME representation must:

- Identify the individual documents represented.

- Contain a property list and an *Uniform Resource Identity* (URI) catalog. The URI catalog maps URIs in XML documents to attachments within the same MIME message.
- The document type, including the document type version.

Envelopes also contain a globally unique identifier. This is referenced as Envelope ID, Message ID, and sometimes as a Transaction ID. A Correlation ID is also used to associate request and reply documents.

Envelope Properties

Envelopes have properties that you can use for routing and bookkeeping. There are two kinds of properties:

- **Managed properties** These are created by XCC and are write-once and then read-only.
- **User-provided properties** You can use user-defined properties as normal property lists.

Attachments

Two kinds of attachments are supported:

- **Element-related attachments** Using the client application, you can add attachments using a Uniform Resource Identifier (URI). Each URI and thus the attachment is assocated with a single element in the main document. (For example, an element named “Requisition.”) The attachment is then added to the envelope. The server application extracts the URI from the element and gets the actual attachment(s).
- **Message-related attachments** The attachment(s) are added to the Envelope. The attachment(s) are extracted iteratively. No URI is used.

Using the Document Interface

`Document` is one of two main abstractions in the Document Framework. The other is described in "Using the Envelope Interface" on page 2-2. The Document Interface is used by an appliation to create and modify individual documents. It allows the appplication to transfer a document from an XLM stream representation to a Java object representation and back again. The `Document` interface is found in the following package:

```
com.commerceone.xdk.swy.metadox.type.Document
```

To retrieve document content, you need to tell your application services and clients how to identify the document's representation class. XCC uses the `Document` interface to handle a document regardless of its content and representation. Specific document representations are subtypes of `Document`.

A few of the available `Document` methods are:

```
getType()          getIdentify()       toStream()
getVersion()       getDocument()
```

See the JavaDoc included with the installed XCC executable for more complete information on the many methods included with XCC.

XCC includes a set of generated XML schemas and their java bean representations. For example, Purchase Orders and Price Check classes are included with the shipped XCC product.

The method `getDocument()` returns `this` (current representation) for documents. Document wrappers return the document.

Note .....If your application handles wrappers and documents interchangeably, using `getDocument() is` a good approach to avoid using the traditional `case` statement. Just use `getDocument()` and apply the document operations.

Type Interface

Use the `Type` interface in subscription and routing to determine the document type and the MIME type for the document instance. The `Type` is added to the message header. The `Type` interface is found in the following package:

```
com.commerceone.xdk.swy.metadox.type.Type
```

Identify Interface

Use the `Identity` interface to assign documents a globally unique identity. This is useful for tracing and auditing log keys in databases. `Identity` is normally not represented in XML, but is extracted and put in headers when sent.

`The Identity` interface is defined in the following package:

```
com.commerceone.util.identity.Identity
```

Document Representation Classes

An XML Document can be represented in several ways. Three methods are currently implemented within XCC, and others may be added in the future. Documents are usually presented to a compatible application in either of two ways, either XML-based or generic representations.

- **XML-based representations**

  `DocumentObject` representations are strongly typed Java objects created using XML to Java (X2J[1]) mapping. `DocumentObject` representations require the MIME-wrapped representation of the document to be a valid XML document, although in some cases well formed is sufficient. In the XML-based situation, the document framework can provide a programming model that is easy to use for application programmers. XCC uses XML to Java (X2J) to map documents in an XML compliant document.

- **Generic representations**

  Generic representations can be used regardless of the actual wire format of a document. The interpretation of a generic representation is done by the application, or some external application. A Microsoft Word document is a typical generic document representation.

  If the element type does not extend another element type, the mapped interface inherits from a general interface called `ElementType`. `ElementType` specifies generic methods that need to be available on all element type interfaces. If an element type extends another element type, the corresponding Java interface will extend the interface corresponding to the extended element type.

  To distinguish between these programmatically we have introduced separate classes for these programming models. Representations of a particular document either extends the corresponding Document Framework class, or is encapsulated by it.

  The generic representation can be one of either:

  - `DocumentStream` Representation as raw character stream.
  - `DocumentBytes` Representation as a binary stream.

  Documents are received as a byte-stream and XCC does not attempt to interpret the stream.

1. See Appendix B “References”.

Both the XML-based and generic representation types implement the `document` representation for the Document Framework interface, and add methods and data specific to the that type of representation.

XCC is installed with a collection of business document schemas and their associated XML to Java (X2J) representations. These are ready to integrate into your applications out-of-the box. Each of these documents, for example, a Purchase Order or a Price Check, represent a MarketSite-supported transaction.

Figure 2.3 below illustrates how each document representation has a corresponding class that implements a common interface `Document`.

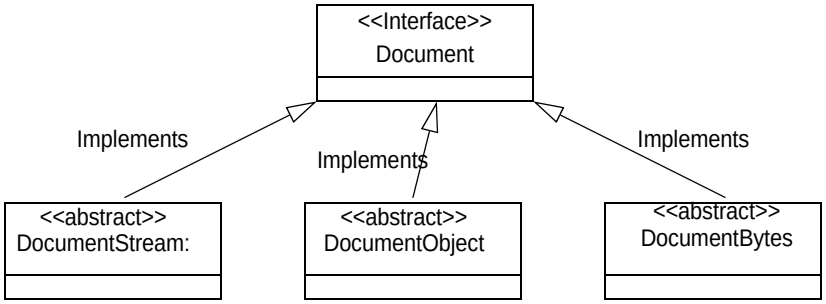

Figure 2.3 Document representation classes (UML notation used)

DocumentObject

`DocumentObject` is a representation of an XML documents. `DocumentObject` is a more strongly typed programming model than Document Object Model (DOM). The `DocumentObject` model gives programmers a more convenient and secure way to manipulate and read documents. Each element is represented by a typed object that contains the methods for accessing and manipulating the content specified for the element by the Schema.

The document object representation uses the XML to Java mapping, which enables applications to easily manipulate and process Java Bean structures that correspond to documents that are instances of a SOX Schema Definition.

Each element type in the pre-packaged document schemas included in XCC is mapped to a corresponding Java interface with the same name as the element type. The interface contains `set` and `get` methods corresponding to

the content model as well as a method for getting the attributes defined for the element type. The `set` and `get` methods are typed based on the XML schema. For example, the `set` and `get` method for the element `State` in Address will take and return a reference of type `CountryCode` respectively.

The `DocumentObject` is an abstract base type for all interfaces that implement the Java Bean representation of documents. The Java Beans mapping defines a class `ElemenTypeImpl` that extends the `DocumentObject` class. For each element `X` we create an interface `X` and a class `Ximpl` implementing `X`. Since `DocumentObject` subclasses `Document`, all the classes corresponding to Element will also implement `Document`.

Figure 2.4 below illustrates element mapping. The created Entities are shaded. For a more complete description of this mapping, see the Java Beans mapping specification available from Commerce One. (See [X2JDOC] on page Appendix B-2.)

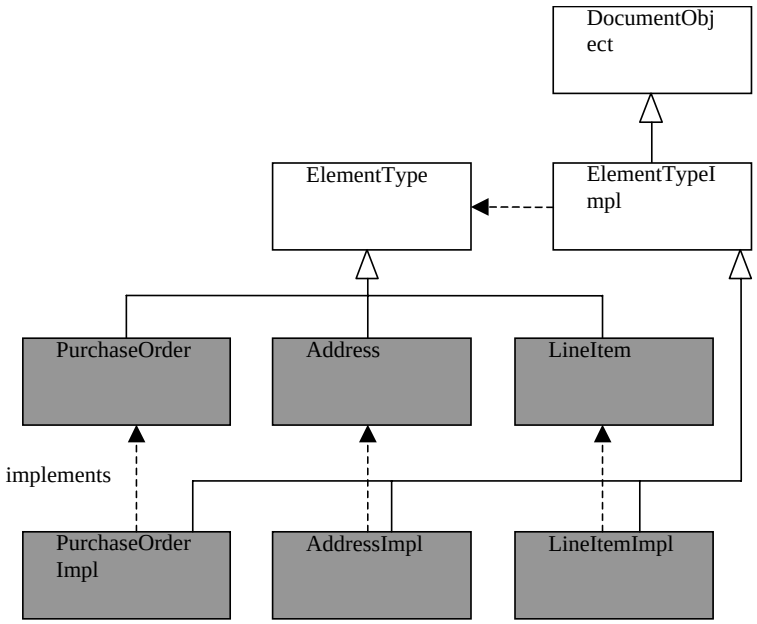

Figure 2.4 JavaBeans class mapping (UML notation used)

DocumentStream

A service can be implemented to a particular programming model, or take advantage of the benefits of different programming models at different states of the processing. In some cases it's beneficial to defer explicit processing of incoming documents. For these situations, the `DocumentStream` representation is particularly useful.

The `DocumentStream` is an abstract base class for all stream based document representations.

The `DocumentStream` programming model allows an application or XCC to keep a document as a raw character stream. The stream can be an XML document or a document in some other format. You can design the application to retrieve the stream and process it in any method necessary.

The `DocumentStream` representation is particularly useful for non-XML document 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.

EntityManager

The `EntityManager` resolves the contents of the URI to an `ExternalSource` and is responsible for opening resources represented by the URI values. This can be used, for example, to look up in a file system a SOX schema for validating a XML document.

URN Naming Conventions and Schema Paths

All Schema names are represented by URN identifiers. The SOX 2.0 schema identifier always starts with a `soxtype` declaration. The `soxtype` declaration contains the URI of the schema to which the instance document claims to conform. XCC automatically determines the physical location a schema using the URI. For this reason, you must follow strict rules for writing URIs.

In the following example, the schema is contained in the file `PO.sox`.

```
urn:x-commerceone:document:com:commerceone:marketsite:svcs:businessvcs:PO.sox$1.0
```

`urn`

The *service* name. Does not change.

`x-commerceone`

The *Namespace Identifier* for a Commerce One XML document. Does not change.

`document:com:commerceone:marketsite:svcs:businessvcs:PO.sox$1.0`

The *Namespace Specific String* describes the "logical directory" plus the version. It is a hierarchical structure, beginning with `document` and using colons (:) as the delimiter.

In this example the remaining portion of the URI is

`marketsite:svcs:businessvcs:PO.sox$1.0`

The portion of the URI before `PO.sox` is a representation of a partial path to the file `PO.sox`, with the file separator replaced by colons. This path is followed by the name of the schema file, in this case `PO.sox`, and then the version token.

To determine what the root of your schema tree is on your file system.

a) Identify the location in the file hierarchy underneath which all your schemas are located. The root is represented as (ROOT) in the example.

b) Exactly underneath the root, the path to the schema file in this example has to start with `svcs/businessvcs/`. Notice that the part of the path following the root is exactly the same as the URN fragment specified in the previous step, up to the file name `PO.sox`, with the colons replaced by file separators.

c) 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 `PO.sox` is located in the directory `(ROOT)/marketsite/svcs/n1_0`.

`com:commerceone`

The top-level directory is the HTTP domain name of the organization providing the resource, with the domains flipped. For example, the domain name "commerceone.com" becomes `com:commerceone.`

Colons (:) are escaped using semicolon followed by colon (;:). Semicolons themselves are represented using two semicolons (;;).

`marketsite`

Represents the subdirectory structure is up to that particular organization. In our build environment, it mirrors the Java packages structure.

`businessvcs`

The last part of the name between the last colon (:) and before the dollar sign ($) is the actual document name.

`$1.0`

A dollar sign ($) precedes the version number. For MarketSite 3.0, the version number must be “1.0”.

Note ..... Versioning is not yet implemented in XCC. This is reserved for future releases.

The complete physical path to the file represented by the URN

```
urn:x-commerceone:document:com:commerceone:marketsite:svcs:businessvcs:PO.sox$1.0
```

is therefore:

```
(ROOT)/marketsite/svcs/n1_0/PO.sox
```

where:

`urn`

The service name. Does not change.

`x-commerceone`

The `namespace` identifier. Does not change.

`document:...$1.0`

The `namespace specific string` identifying the particular schema and the path to the schema.

Chapter 3
Writing an Application Service

In this Chapter

Use the *Service Framework* to integrate applications with XCC. You can use the classes provided or extend them as needed. You then install this service on an XCC Server, enabling you to receive and send documents within your market place.

Application Service Prerequisites

Before your application service will run, it needs to be able to find the SOX schemas included with the XCC. They are located in your file system installed with the executables. The full path is specified in the `server-start.prop` file which you must configure before you install the service. See the ***XCC Installation and Configuration Notes*** for complete information.

Using the Service Framework

The *Service Framework* is a collection of Java abstract classes and interfaces necessary for implementing an application. It contains the components and programming interfaces you use to integrate applications with XCC. The `AbstractDocumentService` is the base class for all services that receive and process documents.

When using the Service Framework to develop an application, you create a new Java subclass as shown in Figure 3.1 on page 3-2. This subclass represents a specific service that handles business documents of a specific type. After you install the service on a server, the subclass implements specific methods supporting your new application.

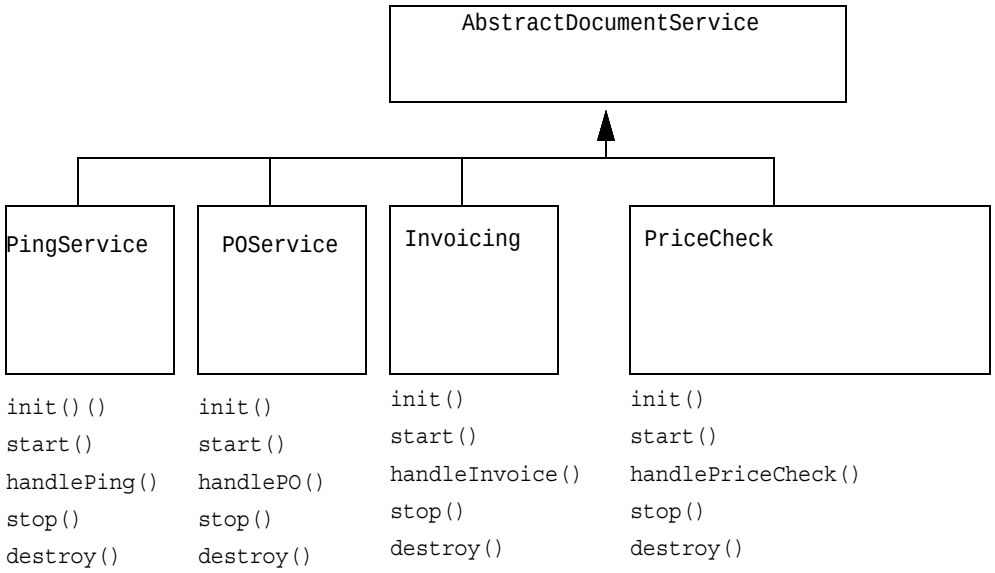

Figure 3.1 PingService is an simplified example. POService, Invoicing and PriceCheck are included in XCC.

When you install the new service, you must configure it to run on a specific server and tell the MarketSite platform how to route its messages.There are a specific set of files you must modify to configure your application. These are covered in detail in the ***XCC Installation and Configuration Notes.***

Service Framework

The different categories of services have well-defined interfaces and abstract service implementations for developers to extend. Not all of the services are exposed in XCC. A Service is Manageable, which means it can be initialized, started, stopped, and destroyed. A `Service` is the basis for most functionality in the Service Framework.

Caution.....Be sure to implement both the `start` and `stop` methods as part of a clean shutdown. Your service will fail if you do not invoke these.

Document Services

Document Services are services that accept documents from the document router through the `DocumentListener` interface and from other services through the `DocumentResponder` interface. Most core services and application services are Document Services. They need to get Envelopes containing documents before they can be processed.

The `AbstractDocumentService` implements the `DocumentService` and the `DocumentListener` interfaces.

Subscribing to Document Types

Specific document services, for example, `AvailabilityCheck`, subscribe to documents of a certain type. The subscriptions occur when a service is initialized. This is a very common pattern for `init()` implementations. You must always first call `super.init()`, as in the example from the PriceCheck service below.

```
public void init()
{
       super.init();
       this.setDelay();
       this.subscribe(PriceCheckRequest.DOC_TYPE);
}
```

As soon as the service is started it must be ready to accept documents from the router. This example uses the convenience method `subscribe()`. This method is inherited from `AbstractDocumentService`.

Services include, but are not limited to, Document Services (sometimes called application services) and Auxiliary Services.

For more advanced applications, you can subscribe to documents using three key properties:

- SenderID
- ReceiverID
- Document Type

You can set up your service to subscribe on these three properties. The subscription mechanism supports an asterisk (*) as a wild card for all three properties.

`AbstractDocumentService` normally have queued input, which means a service implementor can treat his input as a serial stream of envelopes. There can be no concurrent invocations.

The envelope queue makes it possbile to stop a service even though the queue keeps receiving envelopes. When the services is started again, it starts receiving envelopes in a sequence from the queue. No envelopes are lost.

The method `DocumentListener` operation `handleDocument()` is implemented in this `AbstractDocumentServices` to dispatch on document type. A subclass has the option to override the method `handleAll()`, which allows the service to use one method to get all subscribed documents.

If you expect more than one document type, you might usually think of testing for each document type using `instanceOf`. The dispatch approach instead allows you to implement methods for each document type, eliminating the need for `doctype` tests. Subclasses implement the methods `handle${doctyp}` for each doctype they subscribe to.

In short, you just need to implement a single method, as shown below.

```
this.subscribe(PriceCheck.DOCTYPE)
```

Use the `publishReply()` method to route a reply back to the waiting sender. For example (from the `PriceCheckService`):

```
       public void handlePriceCheckRequest(Envelope envelope) throws
DocumentExchangeException
       {
           System.err.print("--------------------");
           this.waitDelay();
           System.err.print("--------------------\n");

           PriceCheckRequest pc =
(PriceCheckRequest)envelope.getDocument(DocumentObject.REPRESENTATION);
           System.err.println("From PriceCheckRequestService: Received
PriceCheckRequest ... ");

           //Create output file to dump document sent by the client
```

```
            File outFile = new File("c:\\commerceone\\xcc\\bin\\pc.out");
            try
            {
                FileOutputStream fout = new FileOutputStream(outFile);
                pc.toStream(fout);
                fout.flush();
                fout.close();
            }
            catch (Exception ex)
            {
                ex.printStackTrace(System.err);
            }

            // Populate response document
            ManagedProperties prop = envelope.getProperties();
            Credential cred = this.getCredential(envelope);
            PriceCheckResult pcr =
PriceCheckResultHelper.createPriceCheckResult(prop,pc,cred);

            // Synchronously publish the reply
            this.publishReply(pcr, envelope);
        }
```

For a complete example of a service, see the sample PriceCheckHelper in package

`com.commerceone.sample.xcc.pricecheck`

The path to the class files is:

`$XCCROOT\sample\com\commerceone\sample\xcc\pricecheck`

Chapter 4
Understanding Clients

In this Chapter

This chapter describes how to create both stand-alone clients and service clients for creating and receiving XCC documents.

Choose a XCC Client

There are two kinds of clients you can use to interact with a remote application.

- **Stand-alone client** A stand-alone client can synchronously transmit envelopes and receive responses to the envelope, and asynchronously transmit envelopes. A stand-alone client is external to the XCC server and cannot receive and process asynchronous replies.
- **Service client** A service client can handle many transactions on a routine, hands-off basis. It can use properties in its configuration files to automatically receive and reply to certain documents, for example, price queries or inventory alerts. Service clients are hosted by a XCC Server, giving the application access to server thread pools, asynchronous event logging, routing, and asynchronous envelope reception. See "Creating a Service Client" on page 4-3.

A XCC client can communicate with a MarketSite server three ways:

- **Synchronous** For each transmitted message, the calling thread within the client blocks until either a response envelope is received or a time-out occurs. If no response is received, the connection times-out and an error is generated. The PriceCheck class is a good example of a synchronous client. It is found in:

  ```
  com.commerceone.sample.xcc.PriceCheck
  ```

- **Peer-to-peer** A reply is anticipated by the client or an agent for the client but is not guaranteed within any fixed time interval. The message

is sent asynchronously; the client is not blocked. For an example of a peer-to-peer service see the AsyncPurchaseOrderClient class in the package:

```
com.commerceone.sample.xcc.AsyncPurchaseOrderClient
```

- **One-way** The message is sent asynchronously. No reply is expected and the client is not blocked.

Using a Stand-alone Client

A stand-alone client can synchronously transmit envelopes and receive responses to the envelope, and asynchronously transmit envelopes. A stand-alone client is external to the XCC server and cannot perform all of the functions available in a service client.

The simplest way to create a working stand-alone client is to clone the existing PriceCheck service included in the XCC Server. See the instructions on "Compile and Copy the Sample Application" on page 1-6

Configuring a Client

When a message is received, your application service must validate the object to verify that the schema is correct. XCC looks for the necessary the SOX schema path and other necessary properties in the `client.prop` file in a search list. The default location for `client.prop` is the current directory. If it is not found there, the compiler searches for it in the home directory for the document being compiled.

See the ***XCC Installation and Configuration Notes*** for complete details on configuring the transmitter.

Example Clients

The XCC Server includes examples of working applications. The PriceCheck service is a simple example for creating a stand-alone client to communicate with a `PriceCheck` service. You can find the sample `PriceCheckHelper` in package

```
com.commerceone.sample.xcc.pricecheck
```

The path to the class files is:

```
$XCCROOT\sample\com\commerceone\sample\xcc\pricecheck
```

Using a Service Client

XCC contains the ability to transmit documents using a stand-alone client to transmit documents directly, or from a Document Service using the TransmitterService. The second method allows an XCC-based application to securely transmit documents to and from a MarketSite server.

A Document Service client can handle many transactions on a routine, hands-off basis. It can use properties in its configuration files to automatically receive and reply to certain documents, for example, price queries or inventory alerts. Service clients are hosted by a XCC Server, giving the application access to server thread pools, asynchronous event logging, routing, and asynchronous envelope reception.

Note .....Each Service Client has it's own configuration. The configuration, set up during installation, is available at runtime by calling `this.getConfiguration()`.

The Transmitter Service is used in two ways by Document Services:

- Implicitly, as a result of publishing a Reply envelope in response to a peer-to-peer request.
- Explicitly, by requesting the transmission of a synchronous, peer-to-peer or one-way transmittal.

Implicit Usage

Implicit usage is invoked by calling the `publishReply()` method of a service's `AbstractDocumentService` superclass, with the envelope argument containing an envelope whose request mode is set to peer-to-peer, as illustrated in the following code example from `AsyncPurchaseOrderService`:

```
// Sync or peer-to-peer; publish reply
this.publishReply(reply, envelope);
```

Internally, the `publishReply()` method routes the reply to the `TransmitterService` for transmission. If transmission fails, the `TransmitterService` forwards the Reply Envelope to the MarketSite `Lost&Found` service.

Explicit Usage

Explicit usage differs depending on whether the request mode of the envelope to transmit is synchronous, peer-to-peer or one-way. Usage for each mode is described below.

Synchronous

Synchronous mode transmissions are invoked by directly calling the `processDocument()` method of the `TransmitterService`. This involves first obtaining a reference to the `TransmitterService` object, and then invoking the `processDocument()` method of the object.

An example from `PriceCheckTransmitter`:

```
ParameterList modeParams = new ParameterList();
EnvelopePropertyValue ep;
...
//set sync mode
ep = new EnvelopePropertyValue(SYNC_MODE_VALUE, modeParams);
request_env.setRequestMode((PropertyValue)ep);
...
//get reply envelope
reply_env = responder.processDocument((Envelope)request_env);
```

Peer-to-peer and One-way

Because of their asynchronous characteristic, peer-to-peer and one-way transmissions are routed to the `TransmitterService` rather than invoked by directly calling a `TransmitterService` method. This routing is achieved by wrapping the document to transmit in a Transmit wrapper, and then publishing the envelope by calling the `publishReply()` method of the service's `AbstractDocumentService` superclass.

For example, from the Async`PurchaseOrderService`:

```
// get envelope properties
PurchaseOrder in = (PurchaseOrder)
```

```
envelope.getDocument(DocumentObject.REPRESENTATION);
...
ManagedProperties prop = envelope.getProperties();
Credential cred = this.getCredential(envelope);
PurchaseOrderResponse reply =
AsyncPurchaseOrderResponseHelper.createPurchaseOrderResponse(prop, in, cred);

// Sync or peer-to-peer; publish reply
this.publishReply(reply, envelope);
```

Transmitter Parameters

For each transmission method used in a stand-alone client or in a service client, the following specific keys are valid. You can obtain the values for the constant values from:

`com.commerceone.xdk.swi.metadox.property.PropertiesConstants`

Synchronous

`TIMEOUT_PARAM_KEY`

Request time-out in seconds or `INFINITE_TIMEOUT_VALUE`.

`ACK_PARAM_KEY`

Must be set to `ACK_NO_PARAM_VALUE`.

For example, from the `PriceCheckTransmitter`:

```
String timeout = cxt.getProperty(TIMEOUT);
...
modeParams.set(TIMEOUT_PARAM_KEY, timeout);
```

Peer-to-peer and One-way

`ACK_PARAM_KEY`

Must be set to `ACK_YES_PARAM_VALUE`.

For example, from the `AsyncPurchaseOrderClient`:

```
modeParams.set(ACK_PARAM_KEY, ACK_YES_PARAM_VALUE);
```

Exception Handling

A client can experience four types of exceptions.

- **EstablishError** An error occurs while establishing connection
- **TransferError** A connection is established, and an error occurs during transmission
- **ProcessingError** The message was transferred, but something goes wrong in processing the message
- **ServiceError** The message reached the application service, but a failure occurred

Catching Exceptions in a Stand-alone Client

Client-side errors in using a transmitter are always seen as exceptions. The error happens before or during connection time. All Exceptions are derived from `DocumentExchangeException`.

```
DocumentExchangeException
EstablishException
TransmitterPropertyException
TransferException
IllegalDocumentTypeException
ProcessingException
XMLConversionException
RepresentationConversionException
```

The exceptions that can be thrown for any transmitter include but is not limited to:

Exception	Description
`IllegalDocumentTypeException`	Thrown when an unexpected Document type was sent or received. The client application would normally not see this exception.
`TransmitterPropertyException`	The provided properties to the TransmitterFactory are bad. Some values come from client.prop, some are created in your program.
`TransferException`	Could not transfer the message to the other end. Most likely an IO exception.

`EstablishException`	Could not establish connection with the other end. Bad address, IO problem or even authentication failures at the server.
`XMLConversionException`, `RepresentationConversionException`	The XML processing on the client went wrong. Either the document you sent out, or the reply document (for synchronous requests) was bad. Also, check the parameters for schema resolution in client.prop.

The application can choose the granularity of the control it requires. Fro example,

```
try {
tx.handleDocument() …
}
catch (DocumentExchangeException ex) {}
```

Catches all exceptions, but any control available is very high-level. To obtain a very fine-grained level of control, you could use the following code:

```
try {
Tx.handleDocument() …
}
catch (XMLConversionException tex) {}
catch (TransferException tex) {}
catch (ProcessingException pex) {}
catch (DocumentExchangeException ex) {}
```

Catching Exceptions in a Service Client

A service client experiences two kinds of exceptions:

- As an exception of a blocking call,
- As an error document.

Most errors are returned as exceptions in the client call. The error document is only sent for asynchronously peer-peer requests when something goes wrong in the business service, after the client connection is closed and acknowledgment is sent.

Connection Related Errors

Connection-related errors occur when the message does not reach a business service. They are always returned as exceptions The client connection is still open and the error document is sent back as return value. The client always sees this error as an exception thrown on the `transmitter.handleDocument()` call.

To inspect the server information, query the exception for the error document, as shown below.

```
com.commerceone.ccs.doclet.error.Error_sox.Error;
com.commerceone.ccs.excp.comm.sender.ServerException;


ServerException ex
Error error = ex.getServerErrorDocument();
// The error document also hold a default message and a severity code,
// see Release Notes
String event_code = error.getCode();
```

Service Related Errors

The message reaches the service, either the business level of the service or the service framework level. Depending on the transmission method used, the error is detected and processed differently.

Transmission Method	Error Reporting Method
synchronous	An error is reported back to the client as a client-side exception
peer-to-peer or one-way	The error is sent as an error document, to be picked up and processed by an Error service in the client installation.

Business services often have business level Error handling schemes and should use that if needed. If you want your client service to use the Error document approach for business level errors, you should extend the `ServiceException` class.

Your service should throw those exceptions to be caught by the Service framework (`CCS_BLOX_DOC_9002_ServiceError`). Any other exceptions leaking from the Business service are returned as a generic error (`CCS_BLOX_DOC_9001_DocumentReceiveFailed`).

EventCode	Severity	Description
`CCS_BLOX_DOC_2112_DocumentReceiveFailed`		
	Error	A generic error in the service.
`CCS_BLOX_DOC_9010_XMLProcessingError`		
	Error	The received document could not be processed by the XML infrastructure.
`CCS_BLOX_SVC_9000_ServiceError`		
	Error	A Business service exception was thrown.
`CCS_BLOX_PROT_2115_ServiceAuthorizationFailed`		
	Error	The client does not have the right to send this document to the service. Authorization failed. Not in MS 3.0.
`CCS_BLOX_SVC_8000_NO_MESSAGE_STORE_FOUND`		
	Error	Outbound replies to asynchronous requests can be archived to allow them to be replayed later if lost. The storage failed.
`CCS_BLOX_PROT_2114_BadProtectionRef`		
	Critical	Server configuration problem.

Appendix A
Glossary

? occurrence

A Schema for Object-Oriented XML element, nested sequence or nested choice specifying an occurs value of "?" is optional and may appear once, or not at all.

* occurrence

A Schema for Object-Oriented XML element, nested sequence or nested choice specifying an occurs value of "*" is optional and may appear multiple times.

+ occurrence

A Schema for Object-Oriented XML element, nested sequence or nested choice specifying an occurs value of "+" must appear, but may appear more than once.

camel case

A naming style from java and C++, where a name will contain of one or several words. Each new word is started with a capital letter. An example would be:

ThisIsACamelCasedName. It is a recommended naming style for Schema for Object-Oriented XML elements and datatypes.

choice content model

A content model of a Schema for Object-Oriented XML elementtype that specifies a number of elements, one of which may appear in an instance of the elementtype.

current namespace

The namespace of the current schema

CXP

Acronmyn for Commerce One's Schema for Object-Oriented XML schema and XML parser.

Daemon

A daemon is a process running virtually all the time. This is accomplished with the means of the operating system: NT service, Inetd, Init.d. The Daemon is responsible for the life cycle and management of one or many CCS servers.

datatype definition

A Schema for Object-Oriented XML element in a schema that defines a user-defined datatype. The datatype can be used in attribute definitions or element definitions.

decimals

A Schema for Object-Oriented XML attribute used in scalar to specify the maximum number of allowed decimals

default

A presence specified for a Schema for Object-Oriented XML attribute that has a default value. If the attribute is not present in the instance, the default value will be used instead.

default namespace

A namespace declared for A Schema for Object-Oriented XML element in an instance, which defines the namespace for that element and all of its contents. This enables a user to avoid prefixes in cases where large elements are not in the current namespace.

digits

A Schema for Object-Oriented XML attribute used in scalar to specify the maximum number of allowed digits (not including decimals)

DOCTYPE declaration

A tag identifies the document type of the current document. Must be present in all Schema for Object-Oriented XML schemas containing the keyword schema and the URI to the current schema.dtd

Document

Base abstraction in the Document Framework. Documents have type and identity and more.

Document Content

The actual content of a document, e.g. a PurchaseOrder document, holds information about item, price, references and more. In a Schema for Object-Oriented XML world content is structured according to a schema.

Document Format

In a normalized world we would only assume XML derived from Schema for Object-Oriented XML. That will not be the case! There will be other XML, raw EDI, attachments of any binary format. This doesn't mean that we need to understand how to parse and interpret any format. It only means that CCS need to be able to pass any document through the system to a service which claim to understand the format. Formats are meaningful concepts for users of services as well as service developers. It is part of the contract, so both Service Description and Service Specification documents list format requirements.

Document Identity

A document needs to have a unique identity. The scope of the uniqueness is very application specific, but still CCS need the guarantees that two documents that are different do not have the same identity. The Commerce Platform system Util provide an Interface Identity with implementations for UUID and Long. It is believed that 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 have a locale associated. 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 Representation

Without compromising with the content a document can have one (and only one at a time) representation out of several possible. Known representations are DocumentObject (Bean, COM), DocumentTree (DOM), DocumentEvents (Event Listeners) and DocumentStream (raw byte stream). Representations are only meaningful for Service implementations; they are not exposed as part of a service description.

Document Type

An XML document has a document type. In Schema for Object-Oriented XML derived XML we talk about Schema for Object-Oriented XML 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. A new field was added to a PurchaseOrder, it is still the same type, but we need to bump up the version. A service accepting PurchaseOrder can reason about versions, and decide what versions it accepts, if external mapping is required etc. The current strategy is to make the version information explicit, but not assign too much semantics and expectations on versioning schemes. Don't confuse Document Version (instance) with the Document Type Version (type).

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 bookkeeping document has changed, ergo new version, but its identity is still the same. For obvious reasons this will not be supported for size/hash value identities. Don't confuse Document Version (instance) with the Document Type Version (type).

Document Wrapper

A class which contain Documents. Examples 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.

DTD

Document Type Definition. Defines a set of structures for XML documents.

elementtype

A Schema for Object-Oriented XML element in a schema that defines an element structure to be used in an instance document.

element content model

A content model of a Schema for Object-Oriented XML elementtype that specifies that only one type of element may be present in the instance of the elementtype.

element element

A Schema for Object-Oriented XML element used in a content model to specify a content element. The type attribute on element specifies what type the element should be of in the instance, and can be of a datatype or element type.

empty content model

A content model of a Schema for Object-Oriented XML elementtype that specifies that no content may be present in the instance of the elementtype.

enumeration

A Schema for Object-Oriented XML element that specifies a user defined datatype that consist of an enumeration of valid values.

Envelope

Base abstraction in the Document Framework. Envelopes hold one and only one document. It can also hold attachments. Properties are kept to facilitate routing and application service code. An Envelope is a DocumentWrapper and ergo a Document.

fixed

A presence specified for a Schema for Object-Oriented XML attribute that has a fixed value. No other value may be specified in the instance.

implied

A presence specified for a Schema for Object-Oriented XML attribute that is optional in the instance.

inheritance

A feature of Schema for Object-Oriented XML allowing a SOX elementtype to derive structure from a previously defined elementtype.

instance document

An XML instance of a Schema for Object-Oriented XML schema. The instance is written in XML format and must conform to a schema or a set of schemas. It may only have one root element.

Logical Directory

The part of an URN that we consider application specific.

MarketSite Domain

A data center which provides a common set of supply-side business and infrastructure services.Each domain includes supplier-buyer data that is specific to a particular region. For example, a MarketSite Domain in England would include data for suppliers and buyers from the British Isles and other neighboring countries.

maxlength

A Schema for Object-Oriented XML attribute used in varchar to specify the maximum length of a string in the instance.

maxexclusive

A Schema for Object-Oriented XML attribute used in scalar to specify if the maximum value specified is or is not allowed in the instance

maxvalue

A Schema for Object-Oriented XML attribute used in scalar to specify the maximum value allowed in the instance.

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.

minvalue

A Schema for Object-Oriented XML attribute used in scalar to specify the minimum value allowed in the instance.

minexclusive

A Schema for Object-Oriented XML attribute used in scalar to specify if the minimum value specified is or is not allowed in the instance.

namespace

A unique identifier of a schema. Since all schemas reside in their own namespace that is separate from all other namespaces, they can refer to other namespaces without any name collisions.

Namespace Identifier

The first part of the URN that uniquely identifies the name space.

nested choice

A choice element that is nested inside another sequence or choice element.

nested sequence

A sequence element that is nested inside another sequence or choice element.

N,M occurrence

A Schema for Object-Oriented XML element, nested sequence or nested choice specifying an occurs value of N,M specifies the valid range of occurrences of that object.

Node

A node is closely bound to a physical machine deployed with one CCS Daemon, and one or multiple CCS Servers.

occurs

A way to specify how many times a Schema for Object-Oriented XML element, nested sequence or nested choice may appear in a content model.

option

A Schema for Object-Oriented XML element used in an enumeration element to specify a valid value for the enumeration.

parser

A processor that parses documents.

polymorphism

Subtypes of elements can appear in an instance whenever the presence of their supertype is specified.

prefix

A name associated with a namespace, used with an object in a schema or an instance to specify that that object is from the associated namespace. In the schema the prefix is used as the value for the prefix attribute, in the instance the prefix is pre-pended to the name of a Schema for Object-Oriented XML element, separated from the element with a colon, ":".

required

A presence specified for a Schema for Object-Oriented XML attribute that must appear in the instance

root element

The outermost set of tags in an XML document that contains all other tags. There may only be one root element in each document, except for any version and document type elements.

scalar

A Schema for Object-Oriented XML element that specifies a user defined datatype that defines a number type datatype with various constraints.

Schema

A document describing the valid structure of a set of XML instance document. A purchase order schema would be an example.

schema.dtd

A DTD to which all Schema for Object-Oriented XML schemas must conform.

schema element

The root element inside a Schema for Object-Oriented XML schema that defines that schema's namespace, as well as wraps all definitions in that schema.

sequence content modelA content model of a Schema for Object-Oriented XML elementtype that constrains the content of the elementtype's instance to the specified sequence. The elements must appear in the correct sequence, with all required elements present.

sequence content model

A content model of a Schema for Object-Oriented XML elementtype that constrains the content of the elementtype's instance to the specified sequence. The elements must appear in the correct sequence, with all required elements present.

Server

A server in this document means a CCS Server. A CCS Daemon is a CCS Server, so where it is understood from the context, we might not spell out Daemon. A server is an execution container for services. A Server is today limited to one operating system process, that will not be the case in later CCS releases.

Service

A service in this document means a CCS Service. It is the granularity of configurable components. Communication Services, Routing Services, Mapping Services and Document Services together make up a server for a specific ecommerce task.

Service Category

A service fall into one of the Service Categories. Most common categories are Document Services , Communication Services, AuxiliaryServices, Routing Services, Mapping Services. For example: `DocumentService`.

Service Full Name

A service full name is a '.' separated concatenation of category, type, name and version. For example:

```
DocumentService.PurchaseOrderService.POService_OfficeDepot.1_1
```

Service Name

A service has a name chosen by the operator at startup. Where only one service instance of a certain service type is started, it is common to let the Service Type also become the Service name, but that is not required. For example:

```
POService_OfficeDepot
```

Service Type

A service type is alway the same as the name of the implementing Service class (without package name). For example:

```
PurchaseOrderService
```

Service Version

A service in this document means a CCS Service. It is the granularity of configurable components. Communication Services, Routing Services, Mapping Services and Document Services together make up a server for a specific ECommerce task.

Service Framework

A Service Framework makes it easy for service implementers to focus on the business problem, not infrastructure. How do I plug in my service? How do I find other services? How do I get my configuration?

SOX

Acronym for Schema for Object-oriented XML. A XML schema language that provides a more powerful way to define a XML structure than a Document Type Definition (DTD) permits. Some of the features that make it more advantageous are: a large number of datatypes that can be used both in attributes; element content; inheritance; and polymorphism.

SOX 2.0 specification

The current Schema for Object-Oriented XML language specification. The SOX Schema is documented at ***http://www.w3.org/TR/NOTE-SOX/***.

SOX schema

Defines structure rules in the form of elementtype definitions and datatype definitions. The schema is written in XML format and conforms to a DTD called "schema.dtd".

SOXtype declaration

A tag that must be present in all XML instances of Schema for Object-Oriented XML schemas. It consists of a processing instruction that identifies the document as being an instance of a sox schema.

string content model

A content model of a Schema for Object-Oriented XML elementtype that specifies that only text content may be present in the instance of the elementtype

URI

Uniform Resource Identifier. Consists of an address to a resource. How that address is resolved depends on the specific scheme. See the current URI working draft for more information (***http://www.w3.org/Addressing/URL/URI_Overview.html***).

URI Catalog

Catalog to map URNs to a URI, containing attachment transport and location information.

URN

Uniform Resource Name. A 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. For example,

```
urn:x-someidentifier:somestring-19980101231145,
urn:isbn:1-56592-169-0
```

Note 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.

UUID

A UUID is a 16 byte globally unique identity. Note 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.

valid

Meaning that the document is conforms to the constraints specified in the document it claims to conform to. That document could for example be a DTD or a Schema for Object-Oriented XML schema. See the XML 1.0 specification or the sox 2.0 specification for more details on validity.

varchar

A Schema for Object-Oriented XML element that specifies a user defined datatype that specifies a string datatype with a maximum length.

well-formed

Meaning that an XML document conforms to the well-formedness constraints set forth in the XML 1.0 specification.

XML

Extensible Mark-up Language. A mark-up language ideal for storing data in a human readable form. Provides means to define tags and structures which enables a highly customizable way of storing data.

XML 1.0 specification

A language specification of the current version of the XML language. You can find the XML language specification at ***http://www.w3.org/TR/REC-xml/*** .

XML Instance Document

An XML document that represent a document containing data. A specific purchase order would be an example of an instance document of the type purchase order.

XML Version Tag

A tag that identifies the current version of XML. Sample format is:

```
<?xml version="1.0"?>
```

This tag must be present in all documents that consist of XML content that should be parseable by an XML processor. It must be present in Schema for Object-Oriented XML schemas.

Appendix B
References

[DOM]

Document Object Model. See http://www.w3.org/.

[SAX]

Simple API for XML. See http://www.megginson.com/SAX and http://www.megginson.com/SAX/SAX2.

[SOX]

Andrew Davidson, Matthew Fuchs, Mette Hedin, Mudita Jain, Jari Koistinen, Chris Lloyd, Murray Maloney, and Kelly Schwarzhof. *Schema for Object-Oriented XML 2.0*. July 1999. See http://www.w3.org/TR/NOTE-SOX

[EXT]

Matthew Fuchs and Jari Koistinen. *Extending and Evolving Electronic Commerce Communities*. Commerce One Technical White Paper. 1999.

[X2J]

Jari Koistinen and Mudita Jain. XML Programming Models for Electronic Commerce Systems. Commerce One Technical White Paper. 1999.

[DCD]

Document Content Description for XML (DCD), Tim Bray et. al. W3C, 10 August 1998. See http://www.w3.org/TR/NOTE-dcd

[XML-Data]

XML-Data, Andrew Layman, et. al. W3C, 05 January 1998. See http://www.w3.org/TR/1998/NOTE-XML-data-0105

[XML]

Extensible Markup Language (XML) 1.0, Tim Bray, et al. W3C, 10 February 1998. See http://www.w3.org/TR/REC-xml

[XSDL]

XML Schema Part 1: Structures, David Beech et al. See http://www.w3.org/TR/xmlschema-1/

[X2JDOC]

Jari Koistinen, Matthew Fuchs, Mudita Jain and Kelly Schwartzhoff. *XML Schema to Java Mapping*. Commerce One Inc. March 1999. Product documentation.

[UML]

G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language*. Rational Software Corporation, January 1997.

Index

Symbols

A

C

D

E

F

G

I

J

L

M

N

O

P

R

S

T

U

V

W

X