Sun B2B Suite HIPAA OTD Library User's Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–1279–10
December 2007

071206@19028

Contents

Preface

The*Sun B2B Suite HIPAA Library User's Guide* describes the HIPAA OTD library, how to install it, and how to use it with eGate Integrator.

Who Should Use This Book

This guide is intended for users who are designing, deploying, and managing Java Composite Application Platform Suite (Java CAPS) Projects that use HIPAA OTDs. This document assumes that you are familiar with eGate-specific procedures.

How This Book Is Organized

This book contains the following chapters:

- Chapter 1, "Overview of the HIPAA OTD Library," provides an overview of the HIPAA OTD Library as well as its support for validation.
- Chapter 2, "Installing the HIPAA OTDs," describes how to install HIPAA OTDs, and the HIPAA OTD Library documentation.
- Chapter 3, "Using HIPAA OTDs," describes how to display OTDs, and how to build Collaborations with HIPAA OTDs.
- Chapter 4, "Java Methods for HIPAA OTDs," provides the syntax for the Java methods provided with the HIPAA OTDs.
- Appendix A, "X12OTDErrors Schema File and Sample XML", provides the X12OTDErrors schema file and a sample validation output XML.

Related Books

For information on eXchange Integrator, refer to the *eXchange Integrator User's Guide*.

For information on the HIPAA Protocol Manager, refer to the *HIPAA Protocol Manager User's Guide*.

For late-breaking information on the B2B Suite, refer to the `B2B_Readme` file, located on the product media.

For information about Java CAPS products, refer to the following:

- *Java CAPS Installation Guide*
- *Java CAPS Deployment Guide*
- *eGate Integrator User's Guide*
- *eGate Integrator System Administration Guide*
- *eGate Integrator JMS Reference Guide*
- *eInsight Business Process Manager User's Guide*
- `Readme` for Java CAPS / eGate Integrator

Screen Captures

Depending on what products you have installed, and how they are configured, the screen captures in this document may differ from what you see on your system.

Related Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX® system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell	`machine_name%`
C shell for superuser	`machine_name#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell for superuser	`#`

◆ ◆ ◆ CHAPTER 1

Overview of the HIPAA OTD Library

This chapter provides an overview of the HIPAA OTD Library as well as its support for validation.

This chapter contains the following topics:

About the HIPAA OTD Library

HIPAA is an acronym for the Health Insurance Portability and Accountability Act of 1996. This Act is designed to protect patients. Among other things, it makes specifications affecting standards of treatment and privacy rights. These specifications provide a number of standardized transactions that can be used for such things as a healthcare eligibility inquiry or a healthcare claim.

For more information on HIPAA, visit the following Web sites:

- `http://www.cms.hhs.gov`
- `http://www.hipaa-dsmo.org`
- `http://www.wedi.org/`
- `http://www.claredi.com/`
- `http://aspe.os.dhhs.gov/admnsimp/`

For more information on NCPDP, visit the official NCPDP web site at this address:

- `http://www.ncpdp.org/`

The HIPAA Object Type Definition (OTD) Library provides OTDs for the 2000 Standard and 2000 Addenda HIPAA messages.

HIPAA messages have a message structure, which indicates how data elements are organized and related to each other for a particular EDI transaction. In Java CAPS, message structures are defined as OTDs. Each OTD consists of the following:

- **Physical hierarchy**: The predefined way in which envelopes, segments, and data elements are organized to describe a particular HIPAA EDI transaction.
- **Delimiters**: The specific predefined characters that are used to mark the beginning and end of envelopes, segments, and data elements.
- **Properties**: The characteristics of a data element, such as the length of each element, default values, and indicators that specify attributes of a data element, for example, whether it is required, optional, or repeating.

The HIPAA OTD Library provides HIPAA OTDs that you can use to build Java CAPS Projects for interfacing with HIPAA systems. You can use the OTDs with eGate Integrator or in combination with eXchange Integrator and eGate Integrator.

HIPAA Validation Support

Within each HIPAA OTD are Java methods and Java bean nodes for handling validation as described in "performValidation" on page 43. No pre-built translations are supplied with the OTD libraries. These translations can be built in the Java Collaboration Editor.

HIPAA OTDs have validations and translations, but a validation does not generate an acknowledgment transaction. Instead, the validation generates a string.

The output string of the validation (see"check" on page 32 and "checkAll" on page 33) is in XML format conforming to the `X12OTDErrors.xsd` file. Refer to "Contents of the `X12OTDErrors.xsd` File" on page 51 for more information. For a sample of the validation output XML, refer to "Sample of Validation Output XML" on page 53.

On Demand Parsing

For performance enhancement reasons, the `unmarshal()` method does not unmarshal the entire message to the leaf element and subelement level. Instead, this method does the following:

- Unmarshals the incoming message at the segment and composite level. In other words, the OTD checks for all relevant segments and composites and reports any missing or extra segments or composites.
- Reports excess trailing delimiter for elements and composites.

This behavior is also referred to as "parse on demand," meaning that elements within a segment or composite are not unmarshaled until an element in that segment or composite is accessed by a `getxxx()` method invoked in a Collaboration or during marshaling. The OTD may assign unmarshaled segments and composites to a pool that is ready to be freed from memory by the Java Virtual Machine software. Once these segments or composites are freed from memory, they become unparsed. If the element within the segment or composite is accessed again, the OTD reparses the segment or composite.

By default, HIPAA OTDs set no limit of parsed segments or composites held in memory. You can specify a limit for parsed and freed segments or composites by using the following methods at the OTD root levels:

- `setMaxParsedSegsComsNum()` method (see "setMaxParsedSegsComsNum" on page 46)
- `setMaxFreedSegsComsNum()` method (see "setMaxFreedSegsComsNum" on page 45)

You can use these methods to set and control the runtime memory use of the unmarshaling process.

Sample of ANSI Output

This section shows an excerpt of the XML X12 4010 850 in ANSI format:

```
ISA*00* *00* *01*9012345720000 *01*9088877320000 *011001*1718*U*
00200*000000001*0*T*:~GS*PO*901234572000*908887732000*20011001*1615*1*T*004010~ST*
850*0001~BEG*01*BK*99AKDF9DAL393*39483920193843*20011001*AN3920943*AC*IBM*02*AE*02*
BA~CUR*AC*USA*.2939*SE*USA*IMF*002*20011001*0718*021*20011001*1952*038*20011001*161
5*002*20011001*0718*021*20011001*1952~REF*AB*3920394930203*GENERAL PURPOSE*BT:123456
78900987654321768958473:CM:500:AB:3920394930203~PER*AC*ARTHUR JONES*TE*(614)555
-1212*TE*(614)555-1212*TE*(614)555-1212*ADDL CONTACT
```

Note – The HIPAA OTDs accept standard ANSI X12 format . You do not need to change the existing Business Processes or Collaborations to specify the input format. The standard data is ANSI X12. By default, the OTD output is ANSI. To change the output to XML (reserved for future use), use the `setXmlOutput (boolean)` method. For information, refer to "setXmlOutput" on page 47. To verify whether the output is XML (reserved for future use), use the `getXmlOutput()` method. For information, refer to "getXmlOutput" on page 40.

Errors and Exceptions

For all HIPAA OTDs, including the envelope OTDs, when the incoming message cannot be parsed, for example, if the OTD cannot find the ISA segment, then the `unmarshal()` method generates a `com.stc.otd.runtime.UnmarshalException`.

The cause of the UnmarshalException depends on which envelope threw the exception:

You can also use the `isUnmarshalComplete()` method on an OTD to verify whether the `unmarshal()` method completed successfully. Successful completion does not guarantee that the OTD instance is free of data errors within segments and composites because elements are not unmarshaled until the first invocation of the leaf element `getElementXxxx()` method of a segment or composite. For more information, refer to "On Demand Parsing" on page 12.

Encountering this exception triggers an automatic background unmarshal of the entire segment. Note that the value returned by the `isUnmarshalComplete()` method is not influenced by the outcome of the automatic background unmarshal, instead, its value reflects what was set by the explicit invocation of the `unmarshal()` method.

ISA Segment Parsing

The ISA segment in a HIPAA message is of fixed length, and has sixteen fixed-length data fields. The fixed-length property of the ISA segment enables you to define the fixed positions for delimiters, such as segment terminator and element separator, that are critical for parsing the HIPAA message. Logic has been built into the fully-enveloped OTDs, including the HIPAA Interchange Envelope OTD to retrieve the delimiters at certain fixed positions and use these delimiters to unmarshal the message.

However, if any of the sixteen data fields is longer or shorter than defined in the specification, the positions of delimiters are nonstandard. This error will result in parsing difficulties and probable runtime failure. In some cases, data that is specified incorrectly in this way can cause conflicting delimiter values, such as element separator defined the same as segment terminator, rendering the whole message unparsable.

Steps Taken to Check for ISA Segment Errors

Assuming valid data, the logic for checking ISA segment errors is as follows:

1. Check whether the character at index=3 is a valid element separator. If not, an UnmarshalException is thrown, and the unmarshaling process is terminated.
2. Check whether the characters at index=6,17,20,31,34,50,53,69,76,81,83,89,99,101,103 is same as that at index=3. If not, an UnmarshalException is thrown with an error message that begins **"ISA Segment Error"**, and processing continues with variable length parsing.

3. Check whether the character at index=104 is a valid subelement separator. If not, an UnmarshalException is thrown with an error message that begins **"ISA Segment Error"**, and processing continues with variable-length parsing.
4. Check whether the character at index=105 is a valid segment terminator, and also verify that this character is different from the element separator and subelement separator. If not, an UnmarshalException is thrown with an error message that begins **"ISA Segment Error"**, and processing continues with variable-length parsing.
5. If the version of X12 is version 4020 or later, check whether the character at index=82 is a valid repetition separator and different from the element separator, subelement separator, and segment terminator. If not, an UnmarshalException is thrown with an error message that begins **"ISA Segment Error"**, and processing continues with variable-length parsing.
6. If variable-length parsing becomes necessary, the ISA segment is parsed using the element separator to retrieve the next sixteeen data elements. After this ISA parsing, an UnmarshalException is thrown with an error message that begins **"ISA Segment Error"**, and the OTD with ISA segment parsed may be examined later to retrieve the parsed ISA segment.

If the message does not have a terminator for its last segment , the ISA segment is also parsed before an `UnmarshalException` is thrown. You can catch this `UnmarshalException` and examine the parsed ISA segment in the OTD, for example, to generate negative TA1 acknowledgment.

◆ ◆ ◆ CHAPTER 2

Installing the HIPAA OTDs

This chapter describes how to install HIPAA OTDs, and the HIPAA OTD Library documentation.

This chapter contains the following topics:

System Requirements

Each HIPAA OTD `.sar` file requires approximately 7.5 MBytes. The combined disk space required to load the standard and Addenda `.sar` files is approximately 15 MBytes.

Due to the size of the HIPAA OTDs, increasing the heap size property of the Enterprise Designer is advisable. For information, refer to "Increasing the Enterprise Designer Heap Size" on page 19.

Other than that, the system requirements for the HIPAA OTD Library are the same as those for eGate Integrator and eXchange Integrator. For information, refer to the *Java CAPS Installation Guide*.

Supported Operating Systems

For information about supported operating systems, refer to the B2B README file.

Installing the HIPAA OTD Library

During the HIPAA OTD Library installation process, the Enterprise Manager, a Web-based application, is used to select and upload products as `.sar` files from the Java CAPS installation CD-ROM to the Repository.

The installation process includes the following steps:

- Installing the Repository
- Uploading products to the Repository
- Downloading components such as the Enterprise Designer and Logical Host
- Viewing product information home pages

Follow the instructions for installing the eGate Integrator in the *Java CAPS Installation Guide*, and include the steps below to install the HIPAA OTDs. You must have uploaded a `Product_List.sar` file to the Java CAPS Repository that includes a license for the HIPAA OTD Library.

▼ To Install the HIPAA OTD Library

1. **After uploading `eGate.sar` to the Java CAPS Repository, select and upload the following items, as described in the *Java CAPS Installation Guide*:**

   - `HIPAA_2000_Standard*.sar` for the standard 2000 HIPAA OTDs
   - `HIPAA_2000_Addenda*.sar` for the 2000 HIPAA Addenda OTDs
   - `HIPAA_OTD_Docs.sar` to install the user's guide

2. **Click the DOCUMENTATION page. Click HIPAA OTD Library in the left pane. Click HIPAA OTD Library User's Guide to download the documentation in PDF form.**

3. **Start or restart the Enterprise Designer, and click Update Center on the Tools menu.**

   The Update Center shows a list of components ready for updating.

4. **Click Add All (the button with a doubled chevron pointing to the right).**

   All modules move from the **Available/New** pane to the **Include in Install** pane.

5. **Click Next. In the license agreement window, click Accept.**

6. **When the progress bars indicate the download has ended, click Next.**

7 **Review the certificates and installed modules, and then click Finish.**

8 **When prompted to restart Enterprise Designer, click OK.**

Increasing the Enterprise Designer Heap Size

Due to the size of the HIPAA OTDs, you might need to increase the heap size property of the Enterprise Designer. If the heap size is not increased, out—of—memory errors could occur.

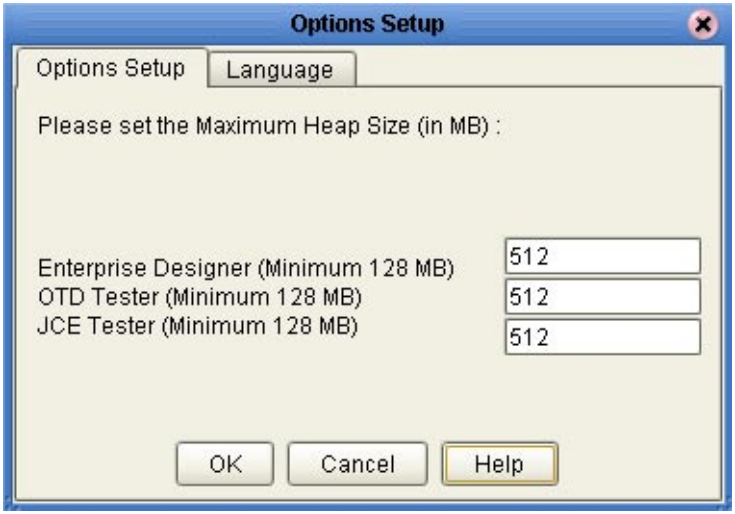
▼ To Increase the Enterprise Designer Heap Size

1 **Start the Enterprise Designer.**

2 **On the Tools menu in Enterprise Designer, click Options.**

The **Options Setup** dialog box appears.

3 **Set the configured heap size for the Enterprise Designer, OTD Tester, and JCE Tester to no less than 512 MBytes, and click OK.**



4 **Restart Enterprise Designer.**

Resolving Memory Errors at Enterprise Designer Startup

If an out of memory error occurs at Enterprise Designer startup, change the setting in the `heapSize.bat` file. This file is resides in the folder *jcaps51*`\edesigner\bin`, where *jcaps51* is the folder where Enterprise Designer is installed.

▼ To Resolve Memory Errors

1. **Open the `heapSize.bat` file in a text editor.**
2. **Change the heap size settings to no less than 512MBytes.**
3. **Save the file.**
4. **Restart the Enterprise Manager.**

◆ ◆ ◆ CHAPTER 3

Using HIPAA OTDs

This chapter describes how you use HIPAA OTDs provided in the HIPAA OTD Library, such as building HIPAA Collaborations.

This chapter contains the following topics:

Displaying HIPAA OTDs

After installing the HIPAA OTDs, you can view the OTDs in the OTD Editor.

▼ To Display HIPAA OTDs

1. **If you are not already in Enterprise Designer, access it by running the following script:**

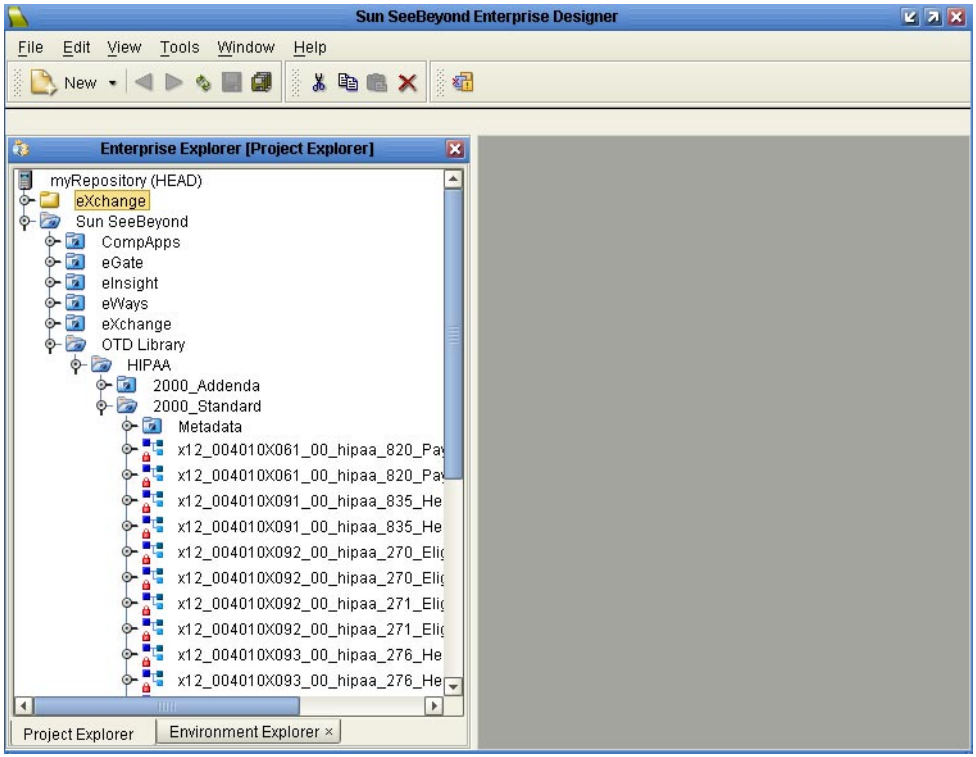
   ```
   c:\...\edesigner\bin\runed.bat
   ```

2. **Click the Project Explorer tab.**

3. **Expand the following folders:**

   - **Sun SeeBeyond**
   - **OTD Library**
   - **HIPAA**
   - **2000_Standard** or **2000_Addenda**

   The Project Explorer tab displays the available OTDs.



The following table below describes the OTD naming conventions.

x12_	X12 protocol name
v*nnnn*_	X12 version
00_	2000 HIPAA
hipaa_	Protocol name
q*n*_	Optional; indicating quarter
820_PayMOrdeAdvi	Transaction code and transaction name abbreviation
_Full	Fully enveloped OTD version that includes the inner and outer envelopes

The folder also includes a **Metadata** folder, which holds a `.zip` file that contains all HIPAA SEF files for the OTDs..

Building HIPAA OTD Collaborations

This section describes how you build Java Collaborations that use the HIPAA OTDs provided in the HIPAA OTD Library.

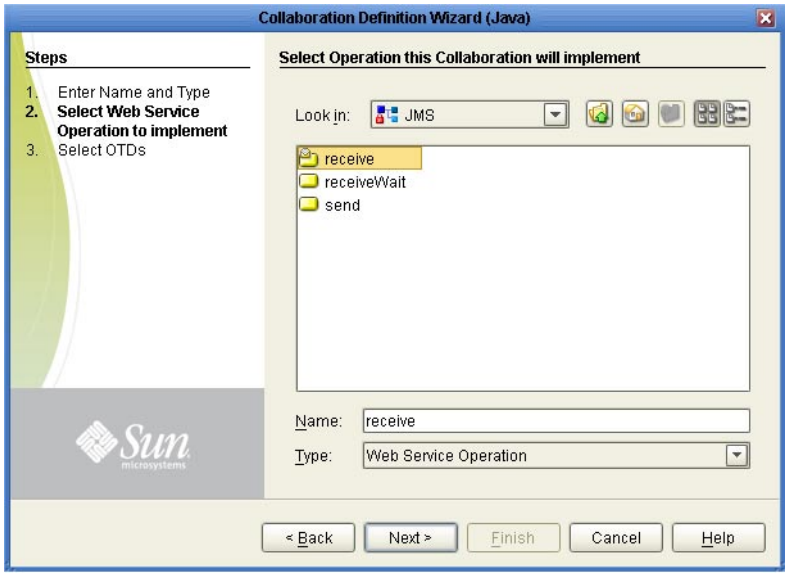
Before you can build the Collaboration, you must have installed the `.sar` file for the particular OTD to be used. For information, see "Installing the HIPAA OTD Library" on page 18.

▼ To Build HIPAA OTD Collaborations

1. **Access the Enterprise Designer.**
2. **Click the Project Explorer tab.**
3. **Click mouse button three on the Project name for which you want to create a Collaboration. Click New. Click Collaboration Definition (Java).**

   The **Collaboration Definition Wizard** dialog box appears.
4. **Type the name of the Collaboration and click Next.**

   The **Select Web Service Operation** page appears.
5. **Select the Web service operation to be implemented by this collaboration, which will trigger the execution of this Java Collaboration Definition, for example, SeeBeyond⇒eGate⇒JMS⇒receive, and click Next.**



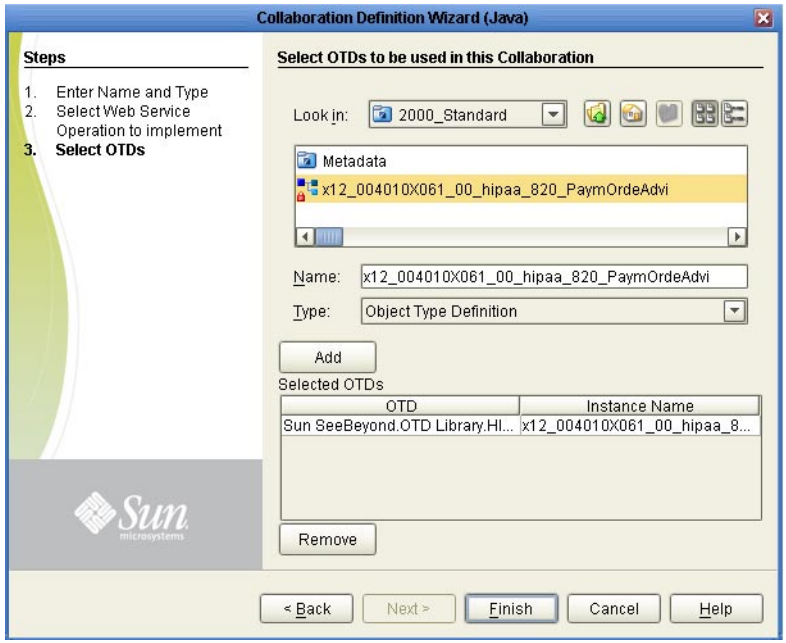
The **Select OTDs** page appears.

6 Under Look In, navigate to the OTDs by double-clickingSeeBeyond⇒OTD Library⇒HIPAA⇒2000_Standard or 2000_Addenda.

The **Look In** area displays the OTDs for the selected HIPAA directories. The following table describes the naming convention for the OTDs.

x12_	X12 protocol
v*nnnn*_	X12 version
00_	2000 HIPAA
hipaa_	Protocol name
q1_	Optional; indicating quarter
820_PayMOrdeAdvi	Transaction code and transaction name abbreviation
_Full	Fully enveloped OTD version that includes the inner and outer envelopes

7 Double-click the OTDs to be used. The OTDs are added to the Selected OTDs list.



8 Click Finish.

The Collaboration appears in the Collaboration Editor. You can now use the eGate and OTD methods to build the business logic for the Collaboration. For information about the HIPAA OTD methods, refer to Chapter 4, "Java Methods for HIPAA OTDs."

Possible Differences in Output When Using Pass-Through

If you are using pass-through, the output file contains essentially the same data as the input file.

Certain differences in output, based on variations in acceptable interpretation of the information, are acceptable, provided that the data conforms to the formats specified for the elements. For example:

- If the input file includes a six-digit date, the output file might represent this as an eight-digit value. For example, 070205 in the input file might be represented as 20070205 in the output file.
- The number of trailing zeros after a decimal point might vary. For example, an input value of 10.000 might be represented as 10 in the output file.

The reason these changes occur is that, during pass-through, certain data fields are parsed and stored as Java objects other than strings, for example, Date or Double.

The actual value of all the information must remain the same.

◆ ◆ ◆ CHAPTER 4

Java Methods for HIPAA OTDs

This chapter describes the Java methods available for HIPAA OTDs.

This chapter contains the following topics:

Get and Set Methods

The OTDs in the HIPAA OTD Library contain the Java methods that enable you to set and get the delimiters, which in turn extend the functionality of the HIPAA OTD Library.

The following get and set methods are available under the root node and at the xxx_Outer, xxx_Inner, and xxx levels:

- "getXmlOutput" on page 40

The following methods are available from the loop elements:

- "get*Loopxxx*" on page 36 and "setLoopxxx" on page 45
- "getSegmentCount" on page 38
- "setXmlOutput" on page 47

Note – The get and set methods are automatically generated from the bean nodes. On occasion, this means get and set methods may be available that are not beneficial, such as setFGValidationResult.

Setting Delimiters

The OTDs must include some way for delimiters to be defined so that they can be mapped successfully from one OTD to another. The HIPAA delimiters are as follows:

- Data element separator. Default is an asterisk.
- Subelement separator/component element separator. Default is a colon.
- Repetition separator. Default is a plus sign.
- Segment terminator. Default is a tilde.

The repetition separator and subelement separator are explicitly specified in the interchange header segment. The other two delimiters are implicitly defined within the structure of the interchange header segment, by their first use. For example, after the fourth character defines the data element separator, the same character is used subsequently to delimit all data elements; and after the 107th character defines the segment terminator, the same character is used subsequently to delimit all segments.

Incoming Message

Because the fully-enveloped OTD automatically detects delimiters in an incoming message that has the interchange header segment while unmarshaling, do not specify delimiters for the incoming message. Any delimiters that are set before unmarshaling are ignored, and the `unmarshal()` method picks up the delimiter used in the ISA segment of the incoming message.

For non-enveloped OTDs, if the incoming message uses non-standard delimiters, set the delimiters on the OTD instance before the `unmarshal()` method is invoked.

Setting Delimiters

You can set the delimiters in the Java Collaboration Editor using the methods or bean nodes that are provided in the OTDs. Use the following methods to specify delimiters:

- "setDefaultX12Delimiters" on page 44
- "setElementSeparator" on page 44
- "setSegmentTerminator" on page 46
- "setSubelementSeparator" on page 47
- "setRepetitionSeparator" on page 46
- "setSubelementSeparator" on page 47

If the input data is already in HIPAA format, you can use the get methods to retrieve the delimiters from the input data. For information, refer to "Get and Set Methods" on page 27.

Outgoing Message

If an OTD outputs ANSI X12 data rather than XML, you must specify the delimiters only if non-standard delimiters are used. If the delimiters are not specified, the industry standard delimiters are used. (For information about which methods to use for delimiter setting, refer to "Setting Delimiters" on page 28)

Note that the interchange-level object is called a fully-enveloped OTD; the message-level object is called a non-enveloped OTD.

For fully-enveloped OTDs, you can also set the subelement separator and repetition separator from the corresponding elements within the ISA segment.

To add the support of serialization i.e. marshalling of non-root level objects such as segments, composites, and segment loops in addition to root level objects i.e. interchange level objects, group level objects and message level objects; each of the non-root level objects now contains a set of delimiters appropriate to its message type which are accessible through a few methods/APIs mentioned below to the user.

Delimiter Set

The delimiter set for the HIPAA message type consists of the following:

- segmentTerminator
- elementSeparator
- subelementSeparator
- repetitionSeparator
- segmentCodeSeparator equivalent to elementSeparator
- decimalMark

Accessor Methods

The methods to access the delimiters inside a root-level or non-root-level object are:

- `getSegmentTerminator`
- `setSegmentTerminator`

- `getElementSeparator`
- `setElementSeparator`
- `getSubelementSeparator`
- `setSubelementSeparator`
- `getRepetitionSeparator`
- `setRepetitionSeparator`
- `getSegmentCodeSeparator`
- `setSegmentCodeSeparator`
- `getDecimalMark`
- `setDecimalMark`
- `setDefaultDelimiters`

Initialization of the Delimiter Set

The delimiters of a root-level or non-root-level object are defaulted to their industry standard values when the delimiter set is created. Each individual delimiter can be changed by the corresponding setter method during initialization or later invocation by the user.

The initialization of the delimiter set can be triggered either by any of the "Accessor Methods" on page 29, or by any the following additional methods:

- All segments: `marshalToBytes`
- ISA segment only: `setEI10_11_InteContStanIden` (X12 version 4010 or lower, or interchange envelope OTD)
- ISA segment only: `setEI65_11_RepeSepa` (X12 version 4020 or higher)

Precedence of Delimiters

When a fully-enveloped OTD or interchange envelope OTD is used to marshal its content into an outgoing message, the delimiter values in the first delimiter field-containing segment (the ISA segment) can sometimes conflict with the delimiter values specified at the interchange level (that is, at the OTD level). This occurs because the interchange level objects and non-root-level objects can separately allow a user to set delimiters independently in a fully-enveloped OTD.

The delimiter values in the ISA segment, if initialized, take precedence over the delimiter values set for the fully-enveloped OTD or interchange envelope OTD. The precedence order can therefore be represented as follows:

Delimiter set in the ISA segment (if initialized) > Delimiter set in the OTD

Example Showing Delimiter Precedence

The following example method illustrates the precedence of delimiters set in the X12 ISA segment over delimiters set in the root level "com.stc.x12env.runtime.ic.ICEnv" object in an X12 interchange envelope OTD:

```
public String generateOutput() throws Exception {

     String encoding = "utf-8";

    // (1) Create a new instance of X12 Interchange Envelope OTD
    com.stc.x12env.runtime.ic.ICEnv icEnvOtd = new com.stc.x12env.runtime.ic.ICEnv();

    // (2) Set delimiters in the Interchange Envelope OTD
    icEnvOtd.setSegmentTerminator('~');
    icEnvOtd.setElementSeparator('+');
    icEnvOtd.setSubelementSeparator('^');

    // (3) Create a new ISA segment object
    com.stc.x12env.runtime.ic.ISA isaSegment = new com.stc.x12env.runtime.ic.ISA();

    // (4) Populate the fields inside the ISA
    isaSegment.setEI01_1_AuthInfoQual("00");
    isaSegment.setEI02_2_AuthInfo("          ");
    isaSegment.setEI03_3_SecuInfoQual("01");
    isaSegment.setEI04_4_SecuInfo("          ");
    isaSegment.setEI05_5_InteIDQual("13");
    isaSegment.setEI06_6_InteSendID("3105451234");
    isaSegment.setEI05_7_InteIDQual("16");
    isaSegment.setEI07_8_InteReceID("123456789");
    com.stc.runtime.dt.Date date = com.stc.otd.runtime.edi.EdiDate.parse8("20070115");
    isaSegment.setEI08_9_InteDate(date);
    com.stc.runtime.dt.Time time = com.stc.otd.runtime.edi.EdiTime.parse4("1647");
    isaSegment.setEI09_10_InteTime(time);
    isaSegment.setEI10_11_InteContStanIden("U");
    isaSegment.setEI11_12_InteContVersNumb("00301");
    isaSegment.setEI12_13_InteContNumb(905);
    isaSegment.setEI14_15_UsagIndi("T");
    isaSegment.setEI13_14_AcknRequ("1");
    isaSegment.setEI15_16_CompElemSepa(":");

    // (5) Set the rest of delimiters inside the ISA segment object
    isaSegment.setSegmentTerminator('!');
    isaSegment.setElementSeparator('*');

    // (6) Set the populated ISA segment object to the Interchange
    //     Envelope OTD
    icEnvOtd.setISA_InteContHead(isaSegment);

    // (7) Get the IEA segment object inside the Interchange
    //     Envelope OTD; also creates the IEA segment instance
    com.stc.x12env.runtime.ic.IEA ieaSegment = icEnvOtd.getIEA_InteContTrai();

    // (8) Populate the fields inside the IEA
```

```
    ieaSegment.setEI12_2_InteContNumb(905);
    ieaSegment.setEI16_1_NumbOfInclFuncGrou(1);

    // (9) Provide the Functional Group data
    String funcGrp = " GS*FA*123*321*927003*1203*1112*T*004010!ST*997*0001!AK1*FA*1!
    AK2*997*0001!AK3*BEG*31**1!AK4*12:4*479*1*98382LKA!AK5*A*1*3*5*1*3!
    AK9*A*0*433*500006*1*2*5*1*2!SE*8*0001!GE*1*1112!";

    // (10) Set the functional group data inside the Interchange
    //      Envelope OTD
     icEnvOtd.setFunctionalGroup(0, funcGrp.getBytes(encoding));

    // (11) Invoke API to generate serialized byte array output
    //      of the Interchange Envelope OTD
    byte[] results = icEnvOtd.marshalToBytes();

    // (12) return as a string
    return new String(results, encoding);

     }
```

The foregoing example method returns the following string as output:

```
ISA*00*          *01*          *13*3105451234     *16*123456789
*051215*1647*U*00301*000000905*1*T*:!
 GS*FA*123*321*927003*1203*1112*T*004010!ST*997*0001!AK1*FA*1!AK2*9
97*0001!AK3*BEG*31**1!AK4*12:4*479*1*98382LKA!AK5*A*1*3*5*1*3!AK9*
A*0*433*500006*1*2*5*1*2!SE*8*0001!GE*1*1112!IEA*1*000000905!
```

Available Methods

This section describes the signature and description for each available HIPAA OTD method.

check

Signature

```
public java.lang.String[] check()
```

Description

Validates the content of the OTD data tree at runtime and returns a string array of validation errors for the message body only; validation errors for envelope segments are not included. To include envelope segments, see the `checkAll()` method below.

The method returns null if there are no validation errors.

Exceptions

None.

checkAll

Signature

```
public java.lang.String[] checkAll()
```

Description

Validates the content of the OTD data tree at runtime and returns a string array of validation errors for the message body and the envelope segments. The `checkAll()` method is only available for fully-enveloped OTDs.

The method returns null if there are no validation errors.

Exceptions

None.

clone

Signature

```
public java.lang.Object clone()
```

Description

Creates and returns a copy of this OTD instance.

Exceptions

java.lang.CloneNotSupportedException

count*XXX*

Signature

```
public int countxxx()
```

where *xxx* is the bean name for repeatable nodes.

Description

Counts the repetitions of the node at runtime.

Exceptions

None.

countLoop*xxx*

Signature

```
public int countLoopxxx()
```

where *xxx* is the bean node for a repeatable segment loop.

Description

Counts the repetitions of the loop at runtime.

Exceptions

None.

get*xxx*

Signature

```
public item getxxx()
```

where *xxx* is the bean name for the node and where *item* is the Java type for the node.

```
public item[] getxxx()
```

where *xxx* is the bean name for the repeatable node and where *item[]* is the Java type for the node.

Description

Returns the node object or the object array for the node.

Exceptions

None.

getAllErrors

Signature

```
public java.lang.String[] getAllErrors()
```

Description

Returns all the validation errors as a string array. These validation errors include errors encountered during unmarshaling input data and the validation results from both the message and the envelope segments.

Exceptions

None.

getElementSeparator

Signature

```
public char getElementSeparator()
```

Description

Gets the element separator character.

Exceptions

None.

getFGValidationResult

Signature

```
public com.stc.otd.runtime.edi.FGError[] getFGValidationResult()
```

Description

Returns the validation errors for the functional group envelope in the format of an FGError array. This method is available only at the Outer and Inner root levels in fully-enveloped OTDs.

Exceptions

None.

getICValidationResult

Signature

```
public com.stc.otd.runtime.edi.ICError[] getICValidationResult()
```

Description

Returns the validation errors for the interchange envelope in the format of an ICError array. This method is available only at the Outer and Inner root levels in fully-enveloped OTDs.

Exceptions

None.

getInputSource

Signature

```
public byte[] getInputSource()
```

Description

Returns the byte array of the original input data source.

Exceptions

None.

get*Loopxxx*

Signature

`public` **`item`** `get`*`Loopxxx`*`()`

where *Loopxxx* is the bean name for the segment loop and where *item* is the Java type for the segment loop.

`public` *`item[]`* `get`*`Loopxxx`*`()`

where *Loopxxx* is the bean name for the repeatable segment loop and where *item[]* is the Java type for the repeatable segment loop.

Description

Returns the segment loop object or the object array for the segment loop.

Exceptions

None.

getMaxDataError

Signature

```
public int getMaxDataError()
```

Description

Returns the maximum number of message validation errors held in the *msgValidationResult* bean node. If this method returns -1 there is no limit of how many errors can be reported.

Exceptions

None.

getMaxFreedSegsComsNum

Signature

```
public int getMaxFreedSegsComsNum()
```

Description

Returns the maximum number of segment and composite objects marked to be freed from memory. For more information, refer to "On Demand Parsing" on page 12.

Exceptions

None.

getMaxParsedSegsComsNum

Signature

```
public int getMaxParsedSegsComsNum()
```

Description

Returns the maximum number of segments and composite objects to be parsed. For more information, refer to "On Demand Parsing" on page 12.

Exceptions

None.

getMsgValidationResult

Signature

```
public com.stc.otd.runtime.check.sef.DataError[] getMsgValidationResult()
```

Description

Returns the validation errors for the message body. Use this method after the `performValidation()` method. For information, refer to "performValidation" on page 43.

This method is only available at the Outer, Inner, and transaction set levels. It is also available at the top root level of non-enveloped OTDs.

Exceptions

None.

getRepetitionSeparator

Signature

```
public char getRepetitionSeparator()
```

Description

Returns the repetition separator character.

Exceptions

None.

getSegmentCount

Signature

```
public int getSegmentCount()
```

Description

Returns the segment count at the current level.

Exceptions

None.

getSegmentTerminator

Signature

```
public char getSegmentTerminator()
```

Description

Returns the segment terminator character.

Exceptions

None.

getSubelementSeparator

Signature

```
public char getSubelementSeparator()
```

Description

Returns the subelement/composite element separator character.

Exceptions

None.

getTSValidationResult

Signature

```
public com.stc.otd.runtime.edi.TSError[] getTSValidationResult()
```

Description

Returns the validation errors for the message envelope (segments UNH/UIH and UNT/UIT) in the format of an TSError array. This method is available only at the Outer, Inner, and transaction set root levels of fully enveloped OTDs. It is also available at the root level of non-enveloped OTDs.

Exceptions

None.

getUnmarshalError

Signature

```
public com.stc.otd.runtime.check.sef.DataError[] getUnmarshalError()
```

Description

Returns the unmarshal errors as an array of the DataError objects. The unmarshal errors are reported from an `UnmarshalException` generated during unmarshaling. Usually these errors are associated with otd.isUnmarshalComplete=false.

Exceptions

None.

getXmlOutput

Note – This method is reserved for future use.

Signature

```
public boolean getXmlOutput()
```

Description

Verifies whether the X12 OTD will output data in XML format.

Exceptions

None.

Example

```
x12_4010.x12_4010_850_PurcOrde_Outer myOTD=new x12_4010.x12_4010_850_PurcOrde_Outer();
......
......
boolean isXml=myOTD.getXmlOutput();
```

has*xxx*

Signature

```
public boolean has xxx()
```

where *xxx* is the bean name for the node.

Description

Verifies if the node is present in the runtime data.

Exceptions

None.

has*Loopxxx*

Signature

```
public boolean has Loopxxx()
```

where *Loopxxx* is the bean name for the segment loop.

Description

Verifies if the segment loop is present in the runtime data.

Exceptions

None.

isUnmarshalComplete

Signature

```
public boolean isUnmarshalComplete()
```

Description

Flag for whether or not unmarshaling completed successfully. For more information, see "On Demand Parsing" on page 12 and "Errors and Exceptions" on page 14.

Exceptions

None.

marshal

Signature

```
public void marshal(com.stc.otd.runtime.OtdOutputStream)
```

Description

Marshals the internal data tree into an output stream.

Exceptions

`java.io.IOException` for output problems

`com.stc.otd.runtime.MarshalException` for an inconsistent internal tree

marshalToBytes

Signature

```
public byte[] marshalToBytes()
```

Description

Marshals the internal data tree into a byte array.

Exceptions

`jjava.io.IOException` for output problems

`com.stc.otd.runtime.MarshalException` for an inconsistent internal tree

marshalToString

Signature

```
public java.lang.String marshalToString()
```

Description

Marshals the internal data tree into a String.

Throws

`java.io.IOException` for input problems

`com.stc.otd.runtime.MarshalException` for an inconsistent internal tree

performValidation

Signature

```
public void performValidation()
```

Description

Performs validation on the OTD instance unmarshaled from input data.

You can access the validation results from a list of nodes, such as allErrors, msgValidationResult, and the node for reporting envelope errors (such as ICValidationResult, FGValidationResult, and TSValidationResult). For more information, refer to "HIPAA Validation Support" on page 12.

Exceptions

None.

reset

Signature

```
public void reset()
```

Description

Clears out any data and resources held by this OTD instance.

Exceptions

None.

set*xxx*

Signature

```
public void setxxx(item)
```

where *xxx* is the bean name for the node and where *item* is the Java type for the node.

```
public void setxxx(item[])
```

where *xxx* is the bean name for the repeatable node and where *item[]* is the Java type for the node.

Description

Sets the node object or the object array for the node.

Exceptions

None.

setDefaultX12Delimiters

Signature

```
public void setDefaultX12Delimiters()
```

Description

Sets the current delimiters to the default HIPAA delimiters:

- segment terminator = ~
- element separator = *
- subelement separator = :
- repetition separator = +

For more information ,refer to "Setting Delimiters" on page 28.

Exceptions

None.

setElementSeparator

Signature

```
public void setElementSeparator(char arg0)
```

Description

Sets the element separator character. For more information ,refer to "Setting Delimiters" on page 28.

Exceptions

None

setLoopxxx

Signature

```
public void setLoopxxx(item)
```

where *Loopxxx* is the bean name for the segment loop and where *item* is the Java type for the segment loop.

```
public void setLoopxxx(item[])
```

where *Loopxxx* is the bean name for the repeatable segment loop and where *item[]* is the Java type for the repeatable segment loop.

Description

Sets the segment loop object or the object array for the segment loop.

Exceptions

None.

setMaxDataError

Signature

```
public void setMaxDataError(int)
```

Description

Returns the maximum number of message validation errors held in the *msgValidationResult* bean node. If this method returns -1 there is no limit of how many errors can be reported.

Exceptions

None.

setMaxFreedSegsComsNum

Signature

```
public void setMaxFreedSegsComsNum(int)
```

Description

Sets the maximum number of segment and composite objects marked to be freed from memory. For more information, refer to "On Demand Parsing" on page 12.

Exceptions

None.

setMaxParsedSegsComsNum

Signature

```
public void setMaxParsedSegsComsNum(int)
```

Description

Sets the maximum number of segments and composite objects to be parsed. For more information, refer to "On Demand Parsing" on page 12.

Exceptions

None.

setRepetitionSeparator

Signature

```
public void setRepetitionSeparator(char)
```

Description

Sets the repetition separator character. For more information, refer to "Setting Delimiters" on page 28.

Exceptions

None.

setSegmentTerminator

Signature

```
public void setSegmentTerminator(char)
```

Description

Sets the segment terminator character. For more information, refer to "Setting Delimiters" on page 28.

Exceptions

None.

setSubelementSeparator

Signature

```
public void setSubelementSeparator(char)
```

Description

Sets the subelement separator character. For more information, refer to "Setting Delimiters" on page 28.

Exceptions

None.

setXmlOutput

Note – This method is reserved for future use.

Signature

```
public void setXmlOutput(boolean)
```

Description

When used with the parameter set to true, this method causes the HIPAA OTD involved to output XML.

When used with the parameter set to false, this method causes the HIPAA OTD to output ANSI (which is the default output if this method is not used at all).

Use this method when the HIPAA OTD is set to automatic output (the default). If the Collaboration is set to manual output, use marshal (boolean) to achieve the same result.

Exceptions

None.

unmarshal

Signature

```
public void unmarshal(com.stc.otd.runtime.OtdInputStream)
```

Description

Unmarshals the given input into an internal data tree.

For more information, refer to in "On Demand Parsing" on page 12 and "Errors and Exceptions" on page 14.

Exceptions

`java.io.IOException` for output problems

`com.stc.otd.runtime.UnmarshalException` for a lexical or other mismatch

unmarshalFromBytes

Signature

```
public void unmarshalFromBytes(byte[])
```

Description

Unmarshals the given input byte array into an internal data tree.

Exceptions

`java.io.IOException` for input problems

`com.stc.otd.runtime.UnmarshalException` for an inconsistent internal tree

unmarshalFromString

Signature

```
public void unmarshalFromString(java.lang.String)
```

Description

Unmarshals (deserializes, parses) the given input string into an internal data tree.

Exceptions

`java.io.IOException` for input problems

`com.stc.otd.runtime.UnmarshalException` for an inconsistent internal tree. This typically occurs when the OTD does not recognize the incoming message as X12.

◆ ◆ ◆ APPENDIX A

X12OTDErrors Schema File and Sample XML

This appendix provides the contents of the `X12OTDErrors.xsd` file, which is the schema file the validation output string conforms to. This appendix also provides a sample of validation output XML.

For more information, refer to "HIPAA Validation Support" on page 12 and "performValidation" on page 43.

This appendix contains the following topics:

Contents of the `X12OTDErrors.xsd` File

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.4 U (http://www.xmlspy.com) by Tony (TechLeader) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="X12OTDErrors">
    <xs:annotation>
      <xs:documentation>Validation Errors from an HIPAA OTD validation</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="X12ICError" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="X12FGError" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="X12TSError" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="X12DataError" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="X12ICError">
```

```
    <xs:annotation>
      <xs:documentation>Interchange Envelope Validation Error Structure. For TA1 generations</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="InteContNumb" type="xs:string"/>
        <xs:element name="InteContDate" type="xs:string"/>
        <xs:element name="InteContTime" type="xs:string"/>
        <xs:element name="InteNoteCode" type="xs:string"/>
        <xs:element name="ICErrorDesc" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="X12FGError">
    <xs:annotation>
      <xs:documentation>Functional Group Envelope Validation Error Structure.
 For AK1AK9 generations</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="FuncIdenCode" type="xs:string"/>
        <xs:element name="GrouContNumb" type="xs:string"/>
        <xs:element name="NumbOfTranSetsIncl" type="xs:string"/>
        <xs:element name="FuncGrouSyntErroCode" type="xs:string"/>
        <xs:element name="FGErrorDesc" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="X12TSError">
    <xs:annotation>
      <xs:documentation>Transaction Set Envelope Validation Error Structure.
 For AK2AK5 generations</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="TranSetIdenCode" type="xs:string"/>
        <xs:element name="TranSetContNumb" type="xs:string"/>
        <xs:element name="TranSetSyntErroCode" type="xs:string"/>
        <xs:element name="TSErrorDesc" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="X12DataError">
    <xs:annotation>
      <xs:documentation>Transaction Set (excluding envelopes) Validation Error Structure.
 For AK3AK4 generations</xs:documentation>
    </xs:annotation>
    <xs:complexType>
```

```
      <xs:sequence>
        <xs:element name="Level" type="xs:short" minOccurs="0"/>
        <xs:element name="SegmIDCode" type="xs:string"/>
        <xs:element name="SegmPosiInTranSet" type="xs:int"/>
        <xs:element name="LoopIdenCode" type="xs:string" minOccurs="0"/>
        <xs:element name="SegmSyntErroCode" type="xs:short" minOccurs="0"/>
        <xs:element name="ElemPosiInSegm" type="xs:short"/>
        <xs:element name="CompDataElemPosiInComp" type="xs:short" minOccurs="0"/>
        <xs:element name="DataElemRefeNumb" type="xs:string" minOccurs="0"/>
        <xs:element name="DataElemSyntErroCode" type="xs:short"/>
        <xs:element name="CopyOfBadDataElem" type="xs:string" minOccurs="0"/>
        <xs:element name="RepeatIndex" type="xs:short" minOccurs="0"/>
        <xs:element name="ErrorCode" type="xs:int"/>
        <xs:element name="ErrorDesc" type="xs:string" minOccurs="0"/>
        <xs:element name="Severity" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Sample of Validation Output XML

```
<X12OTDErrors>
  <X12ICError>
    <InteContNumb>000000001</InteContNumb>
    <InteContDate>041102</InteContDate>
    <InteContTime>1441</InteContTime>
    <InteNoteCode>021</InteNoteCode>
    <ICErrorDesc>Invalid Number of Included Groups Value</ICErrorDesc>
  </X12ICError>
  <X12FGError>
    <FuncIdenCode>PO</FuncIdenCode>
    <GrouContNumb>1</GrouContNumb>
    <NumbOfTranSetsIncl>2</NumbOfTranSetsIncl>
    <FuncGrouSyntErroCode>5</FuncGrouSyntErroCode>
  </X12FGError>
  <X12FGError>
    <FuncIdenCode>PO</FuncIdenCode>
    <GrouContNumb>1</GrouContNumb>
    <NumbOfTranSetsIncl>2</NumbOfTranSetsIncl>
    <FuncGrouSyntErroCode>4</FuncGrouSyntErroCode>
    <FGErrorDesc>Number of Included Transaction Sets Does Not Match Actual Count</FGErrorDesc>
  </X12FGError>
  <X12TSError>
    <TranSetIdenCode>850</TranSetIdenCode>
    <TranSetContNumb>0001</TranSetContNumb>
    <TranSetSyntErroCode>4</TranSetSyntErroCode>
```

```
    <TSErrorDesc>Number of Included Segments Does Not Match Actual Count</TSErrorDesc>
  </X12TSError>
  <X12DataError>
    <Level>1</Level>
    <SegmIDCode>MEA</SegmIDCode>
    <SegmPosiInTranSet>21</SegmPosiInTranSet>
    <LoopIdenCode/>
    <SegmSyntErroCode>8</SegmSyntErroCode>
    <ElemPosiInSegm>4</ElemPosiInSegm>
    <DataElemRefeNumb>C001</DataElemRefeNumb>
    <DataElemSyntErroCode>10</DataElemSyntErroCode>
    <ErrorCode>15025</ErrorCode>
    <ErrorDesc>MEA_4 at 21: [Syntax rule E-Exclusion: One or None]
 Exclusion condition violated because E0412</ErrorDesc>
    <Severity>ERROR</Severity>
  </X12DataError>
  <X12DataError>
    <Level>1</Level>
    <SegmIDCode>N4</SegmIDCode>
    <SegmPosiInTranSet>195</SegmPosiInTranSet>
    <LoopIdenCode>N1</LoopIdenCode>
    <SegmSyntErroCode>8</SegmSyntErroCode>
    <ElemPosiInSegm>7</ElemPosiInSegm>
    <DataElemRefeNumb>1715</DataElemRefeNumb>
    <DataElemSyntErroCode>10</DataElemSyntErroCode>
    <CopyOfBadDataElem>CNT</CopyOfBadDataElem>
    <ErrorCode>15025</ErrorCode>
    <ErrorDesc>N1_N4_7 at 195 [CNT]: [Syntax rule E-Exclusion: One or None]
 Exclusion condition violated because E0207</ErrorDesc>
    <Severity>ERROR</Severity>
  </X12DataError>
</X12OTDErrors>
```

Index