

A framework and code generator for processing huge XML files with complex schemata

“An MDE based approach”

Abstract

Dealing with very large XML documents with complex schemata is difficult. On the one hand because of the sheer size of the data using available technologies that are based on making an in memory representation of the data is not straight forward, on the other when the schema is complex, dealing with alternative technologies such as SAX or StAX is very labour intensive and leads to software that is hard to test (inconsistencies) and maintain. In this article we present an approach based on MDE (model driven engineering) and the SAX 2.0 API. Our design is centered on a common framework complemented by code engineering that directly maps the underlying model (XML Schema) to Java. This approach has the following advantages:

- Time to market is dramatically decreased
- Decreased cost
- It is scalable to very large XML files (low size independent memory footprint)
- It fully eliminates errors related to consistency
- It presents data in a convenient JavaBeans format fully consistent with the XML Schema
- The resulting code is highly efficient and modular
- The resulting software is dependable
- It is very easy to implement changes in the underlying model

Background

Now that application development is being outsourced often applications are built in isolated projects on an as needed basis. The problem with this approach is increased long-term risk. With a solid foundation of common components, software becomes reusable, with the following benefits:

- Decreased time to market for new products
- Decreased cost
- Increased maintainability of code base
- Increased dependability due to reuse

Building applications from scratch involves a tremendous effort from both offshore partners and local staff. Often too much time is spent on technology instead of business functionality, even when development is outsourced. One of the major reasons for this is an opportunistic approach to software development; too often the software development process is immature

resulting in poor reuse and maintainability, building similar functionality over and over again in slightly different ways; potentially common technical components are simply rebuilt as part of different projects.

Conflict of Requirements

Evidently the sizing requirement constrains our design. What it boils down to is:

We cannot use a model that makes an internal representation of all of the data in the document, but on the other hand, why care for parsing the document if we are not to do something useful with it? In other words, we'd like to have the data represented in a convenient format (JavaBeans) and be able to process it from there *without* having to bother with the technicalities of handling the XML document or the intricacies of the schema in hand. What we have here is *conflict of requirements*.

On the one end, there are quite a few technological choices that allow us to make an internal model, so we can work from the convenience of a Java model that makes sense from a semantic point of view. On the other end we find models that throw events at us (e.g. SAX API), or alternatively allow us to walk-through the document using either a cursor or event model (StAX). Neither of these models really fits because they force one of the extremes upon us. We need a way to walk the middle ground. Although lately it has become possible to use JAXB on top of StAX allowing to partially map XML, this approach requires us to mix low-level with high-level programming and still requires some serious design effort in order to make our design modular and extensible.

Making a modular design

So we want our design to be *modular, high-performance, intuitive, maintainable and easy to use* at the same time. We are going to use either the StAX or SAX API but we don't want to clutter our design with extensive control structures (making our design unreadable, poorly maintainable and hard to test). Although the StAX API may seem easier to handle than the SAX API, because of the *pull architecture*, in fact when dealing with complex XML Schemata, it is of not much help and additionally, you may be in a situation making it hard for you to use because of your technical environment. The SAX *push architecture* is fast and has low memory footprint, there's a large choice of implementations for this API available, but it maybe somewhat intimidating for developers new to *publish-subscribe* architectures.

The important challenge clearly is: **How to make a modular and maintainable design that suits our technical requirements and is easy to use at the same time?** An important clue can be found in the fact that the SAX 2.0 API allows us to *swap content handlers* on the fly as often as we like. An excellent article on the matter by Robert Hustead can be found at [1]. In this article Robert explains how to create a modular design that eliminates complex control structures by swapping content handlers.

Dealing with large and complex XML Schemata

In our XML parser framework we use a similar pattern that among others involves creating a *data* (JavaBean) and a *handler* (content handler) class per schema entity (complextype) and some other mostly private classes that glue it all together. This makes our design very modular, but when dealing with large and complex XML schemata (such as the SEPA

PAIN/PACS schemata), implementing all the JavaBeans and handler classes (or trackers as in Robert’s design) and putting it all together by hand is still a very time-consuming and error prone task, if not undoable. To solve this problem we use MDE (model driven engineering) to create all of these classes directly from the model (the ISO-20022 XSDs in this case). This way we have the benefits of efficiency, modularity and at the same time eliminate the inconsistencies that result from the manual process of creating the classes. To give you an idea, for SEPA/Payments alone, the number of data and handler classes exceeds 250 (the total number of generated classes actually exceeds 1250). All of these extend the framework, which is the engine that orchestrates it all.

Dealing with memory usage and implementing business rules

But, let’s go back to our conflict of requirements now. Remember the issue that brought us here is dealing with our technical requirements effectively. To get the picture I’ll use an example taken from the SEPA payments domain (ISO-20022: PAIN.001.001.03). The basic structure of the XML file is as follows: It comprises of a *header*, which contains general information about the message, followed by one or more *batches*, each of which may contain many *transactions*. It’s the transactions that we expect many of, but according to the schema the number of batches is also unbounded.

Type of data	XML element	Path
Header	GrpHdr	Document/CstmrCdtTrfInItm/GrpHdr
Batch	PmtInf	Document/CstmrCdtTrfInItm/PmtInf
Transaction	CdtTrfTxInf	Document/CstmrCdtTrfInItm/PmtInf/CdtTrfTxInf

To complete the picture we also have to add the types (schema types) each of these XML elements is an instantiation of:

Element	Type
GrpHdr	GroupHeader32
PmtInf	PaymentInstructionInformation3
CdtTrfTxInf	CreditTransferTransactionInformation10

When you examine these types (PAIN.001.001.03, see: [2]), you’ll find that for instance *CreditTransferTransactionInformation10* is pretty complex. We *definitely* need to have a convenient representation in Java allowing us to access its data in an intuitive way. The same is true for the *PaymentInstructionInformation3* and *GroupHeader32*. Now we begin to fully understand the problem: We have to deal with complex types, but on the other hand need a way to *prevent* having to store all contained objects within their parent objects if we expect a great many of them. This is why we made the framework’s behaviour configurable through properties: The property *@process* instructs the framework to send a JavaBean containing the element in scope to the registered processor component, the property *@detach* tells the framework it should *not* store the element inside its parent’s container, thus preserving memory (making it short-lived).

The following picture shows the runtime configuration (properties) file for a parser that handles **PAIN.001.001.03** messages:

```
# process header
Document/CstmrCdtTrfInitn/GrpHdr/@process=true Process GrpHdr Element

# process BATCH level and detach from parent
Document/CstmrCdtTrfInitn/PmtInf/@process=true
Document/CstmrCdtTrfInitn/PmtInf/@detached=true Detach from Parent

# process TX level and detach from parent
Document/CstmrCdtTrfInitn/PmtInf/CdtTrfTxInf/@process=true
Document/CstmrCdtTrfInitn/PmtInf/CdtTrfTxInf/@detached=true
```

Figure 1 - Configuration for PAIN.001.001.03 messages

In plain English this configuration tells the framework to:

- Process the *GrpHdr* element (there is only one)
- Process each *PmtInf* element and *not* store it in the parent container
- Process each *CdtTrfTxInf* and *not* store it in the parent container

At this time you may be wondering what happens to all the other XML elements: The nice thing is, the property file *overrides the defaults*, which are:

`@process = false` and `@detach = false`

In other words, the other items (not being detached) are simply *available through getters at their parent level*. For clarity I'll show you a snippet (truncated) of the generated Javadoc for *CdtTrfTxInf*:

AmountType3Choice	<code>getAmt()</code> Get the embedded Amt element.
PartyIdentification32	<code>getCdtr()</code> Get the embedded Cdtr element.
CashAccount16	<code>getCdtrAcct()</code> Get the embedded CdtrAcct element.
BranchAndFinancialInstitutionIdentification4	<code>getCdtrAgt()</code> Get the embedded CdtrAgt element.
CashAccount16	<code>getCdtrAgtAcct()</code> Get the embedded CdtrAgtAcct element.
Cheque6	<code>getChqInstr()</code> Get the embedded ChqInstr element.
java.lang.String	<code>getChrgBr()</code> Get the embedded ChrgBr element.
java.util.List< InstructionForCreditorAgent1 >	<code>getInstrForCdtrAgts()</code> Get the embedded list of InstrForCdtrAgt elements.
java.lang.String	<code>getInstrForDbtrAgt()</code> Get the embedded InstrForDbtrAgt element.
BranchAndFinancialInstitutionIdentification4	<code>getIntrmyAgt1()</code> Get the embedded IntrmyAgt1 element.
CashAccount16	<code>getIntrmyAgt1Acct()</code> Get the embedded IntrmyAgt1Acct element.
BranchAndFinancialInstitutionIdentification4	<code>getIntrmyAgt2()</code> Get the embedded IntrmyAgt2 element.
CashAccount16	<code>getIntrmyAgt2Acct()</code> Get the embedded IntrmyAgt2Acct element.

Figure 2 - JavaBean specification for CdtTrfTxInf

Here you see the *getters* for its contained elements. The use of *java.util.list* is consistent with

the multiplicity for *InstrForCdtrAgts* defined in the schema.

The processor component

Now about the business: Thus far we only talked about the framework and the *Reader Component*. What we want to do *with* the data is implemented by the provided *Processor Component* (*APPLICATION PROCESSOR*). The application programmer will create a *specific processor* by implementing the *required interface* (*DataProcessor*) specified by the framework.



Figure 3 - Application provided Processor Component

The *Reader Component* delivers the JavaBeans to the *Processor Component* in accordance with the instructions in the properties file. Following our example the processor will receive every instance of the *GrpHdr*, the *PmtInf* complete with all of its contained elements (except for the *detached* *CdtTrfTxInf* items) and the *CdtTrfTxInf*, with all of its contained elements (see Fig.2). For flexibility the framework sends two notifications to the processor per element, one when it first encounters the element in the XML document and the second when it encounters its closing tag. This gives the processor maximum flexibility. Let's have a look at the definition of *PaymentInstructionInformation3*:

```

<xs:complexType name="PaymentInstructionInformation3">
  <xs:sequence>
    <xs:element name="PmtInfId" type="Max35Text"/>
    <xs:element name="PmtMtd" type="PaymentMethod3Code"/>
    <xs:element maxOccurs="1" minOccurs="0" name="BtchBookg" type="BatchBookingIndicator"/>
    <xs:element maxOccurs="1" minOccurs="0" name="NbOfTxs" type="Max15NumericText"/>
    <xs:element maxOccurs="1" minOccurs="0" name="CtrlSum" type="DecimalNumber"/>
    <xs:element maxOccurs="1" minOccurs="0" name="PmtTpInf" type="PaymentTypeInformation19"/>
    <xs:element name="ReqdExctnDt" type="ISODate"/>
    <xs:element maxOccurs="1" minOccurs="0" name="PoolgAdjstmntDt" type="ISODate"/>
    <xs:element name="Dbtr" type="PartyIdentification32"/>
    <xs:element name="DbtrAcct" type="CashAccount16"/>
    <xs:element name="DbtrAgt" type="BranchAndFinancialInstitutionIdentification4"/>
    <xs:element maxOccurs="1" minOccurs="0" name="DbtrAgtAcct" type="CashAccount16"/>
    <xs:element maxOccurs="1" minOccurs="0" name="UltmtDbtr" type="PartyIdentification32"/>
    <xs:element maxOccurs="1" minOccurs="0" name="ChrgBr" type="ChargeBearerType1Code"/>
    <xs:element maxOccurs="1" minOccurs="0" name="ChrgsAcct" type="CashAccount16"/>
    <xs:element maxOccurs="1" minOccurs="0" name="ChrgsAcctAgt" type="BranchAndFinancialInstitutionIdentification4"/>
    <xs:element maxOccurs="unbounded" minOccurs="1" name="CdtTrfTxInf" type="CreditTransferTransactionInformation10"/>
  </xs:sequence>

```

Figure 4 - Definition of PaymentInstructionInformation3

It immediately occurs to us it is defined as a sequence. We also find that the contained transactions are last in the sequence. Consequently we know that when we encounter the first transaction, the batch level information is complete and thus can be processed. Of course this can be very convenient if for instance we want to store the data in a database (since it enables us setting the FK relationship with the contained transactions).

Application Development with the Framework

With the LDX framework the team can focus entirely on the design and development of the *Processor Component*. The design of the processor component is not constrained by the

Reader Component; it's up to the team to decide whether they want to implement the *interface DataProcessor* synchronously or asynchronously.

There are a couple of things worth mentioning:

- The implementation of the *org.xml.sax.ErrorHandler* interface, which is good practice allowing your application to respond to error conditions during parsing. This class provides an excellent place to hookup Logging.
- There is a class called *Pain001V03MessageHandler*, which is the entry point for applications dealing with pain.001.001.03 type of messages. This class is generated and available as part of the framework.
- The *ProcessorException* is part of the Processor Component interface. It is in fact the only exception that is allowed to leak through the interface and should be used to report error conditions within the Processor Component that are unrecoverable (e.g. you may want to abort processing upon some special condition).

```
/**
 * ErrorHandler to capture errors during XML processing.
 */
final class MyErrorHandler implements ErrorHandler {

    public void error(SAXParseException exception) throws SAXException {
        // TODO Auto-generated method stub
    }

    public void fatalError(SAXParseException exception)
        throws SAXException {
        // TODO Auto-generated method stub
    }

    public void warning(SAXParseException exception)
        throws SAXException {
        // TODO Auto-generated method stub
    }
}
```

Figure 5 - Hookup points for Logging

Now let's have a look at the application main module. This module is largely boilerplate code, just requiring you to make some changes to adapt it to your case (for example add logging). In this sample we've used the command line arguments to get the runtime configuration file (*args[2]*), the XML Schema (*args[1]*) and the XML document (*args[0]*). The inline comments should suffice to understand the code.

```
try {
    // initialize Pain001 configuration
    ParserConfiguration.instance().load(new FileInputStream(args[2]));

    // create the XMLReader and optionally set the ErrorHandler..
    XMLReader reader = XMLReaderFactory.createXMLReader();
    reader.setErrorHandler(new MyErrorHandler());

    // initialize Pain001V03 message handler..
    Pain001V03MessageHandler handler = new Pain001V03MessageHandler(reader);

    // connect schema file..
    handler.setSchema(new StreamSource(new FileInputStream(args[1])));

    // validate XML prior to processing..
    System.out.println("Validating..");
    handler.validate(new StreamSource(new FileInputStream(args[0])));

    // connect processor to reader component..
    Pain001V03DataProcessor processor = new Pain001V03DataProcessor();
    handler.connectProcessor(processor);

    // connect input source..
    handler
        .setInputSource(new InputSource(
            new FileInputStream(args[0])));

    // start processing..
    System.out.println("Processing..");
    handler.process();
    System.out.println("Done. No of transactions: " + processor.getTxCount());

} catch (SAXException e) {
    System.out.print(e.getMessage());
} catch (IOException e) {
    System.out.print(e.getMessage());
} catch (ProcessorException e) {
    System.out.println(e.getMessage());
}
```

Figure 6 - The Main module

Validation

When looking at validation, there are a couple of cases that we need to distinguish:

- **Reject XML message** (the complete XML) when validation fails
- **Reject individual transaction** when not valid but process the valid ones

In the *first case* we can use pre-validation as shown in the sample code. When the document is not valid (according to the provided XML Schema) a *SAXException* is thrown and the application may report it and exit.

In the *latter case* the XML message is partly valid but some mandatory information may be missing. In this case pre-validation may be skipped or a validation against a relaxed schema (defining minimum requirements) may be done. It is then up to the Processor Component to decide what to do with individual transactions that are incomplete (e.g. log and skip).

Conclusion

The LDX Parser Framework for Java dramatically reduces development time. In fact the team can focus entirely on the design and development of the *business*.

For handling large XML documents with complex schemata we clearly need a technology that automatically maps the XML data into Java allowing us to use a convenient model but *without* the overhead of loading the complete document into memory. The framework we presented here has shown to handle large transaction files with over 1,000,000 transactions with ease whilst providing the convenience of an internal object representation in Java. Its *runtime configurability* (per application instance) allows tuning memory utilization and enabling processing per type of element. The MDE approach automatically generates the Java classes allowing the application programmer to access the data in a convenient and consistent manner and enforces consistency with the model (the XML Schema).

References

- 1 <http://java.sun.com/developer/technicalArticles/xml/mapping/>
- 2 <http://iso20022.org>