JBoss Enterprise SOA Platform 5.0

Smooks User Guide

The User Guide and Reference for the Smooks Framework.

JBoss Enterprise SOA Platform 5.0 Smooks User Guide
The User Guide and Reference for the Smooks Framework.
Edition 1.0

Editor Darrin Mison dmison@redhat.com
Editor David Le Sage dlesage@redhat.com

1801 Varsity Drive
Raleigh, NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588 Research Triangle Park, NC 27709 USA

A reference guide for using the Smooks framework.

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight key caps and key-combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a key cap, all presented in Mono-spaced Bold and all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the hyphen connecting each part of a key-combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F1`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F7`** to return to your X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`Mono-spaced Bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialogue box text; labelled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

[1] https://fedorahosted.org/liberation-fonts/

Choose **System > Preferences > Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications > Accessories > Character Map** from the main menu bar. Next, choose **Search > Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit > Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in Proportional Bold and all distinguishable by context.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether Mono-spaced Bold or Proportional Bold, the addition of Italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username*@*domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules* (*MPMs*). Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

1.2. Pull-quote Conventions

Two, commonly multi-line, data types are set off visually from the surrounding text.

Output sent to a terminal is set in `Mono-spaced Roman` and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in `Mono-spaced Roman` but are presented and highlighted as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }

}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

A note is a tip or shortcut or alternative approach to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring Important boxes won't cause data loss but may cause irritation and frustration.

Warning
A Warning should not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: *http://bugzilla.redhat.com/bugzilla/* against the product **JBoss Enterprise SOA Platform.**

When submitting a bug report, be sure to mention the manual's identifier: *Smooks_Guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Overview

Smooks is a Java framework for processing both XML and non-XML data. Non-XML data includes formats such as CSV, EDI and Java files.

Some of the features available in the current version of Smooks include:

Transformation

This feature perform a wide range of "data transforms" such as from XML to XML, CSV to XML, EDI to XML, XML to EDI, XML to CSV, Java to XML, Java to EDI, Java to CSV, Java to Java, XML to Java, EDI to Java and so forth.

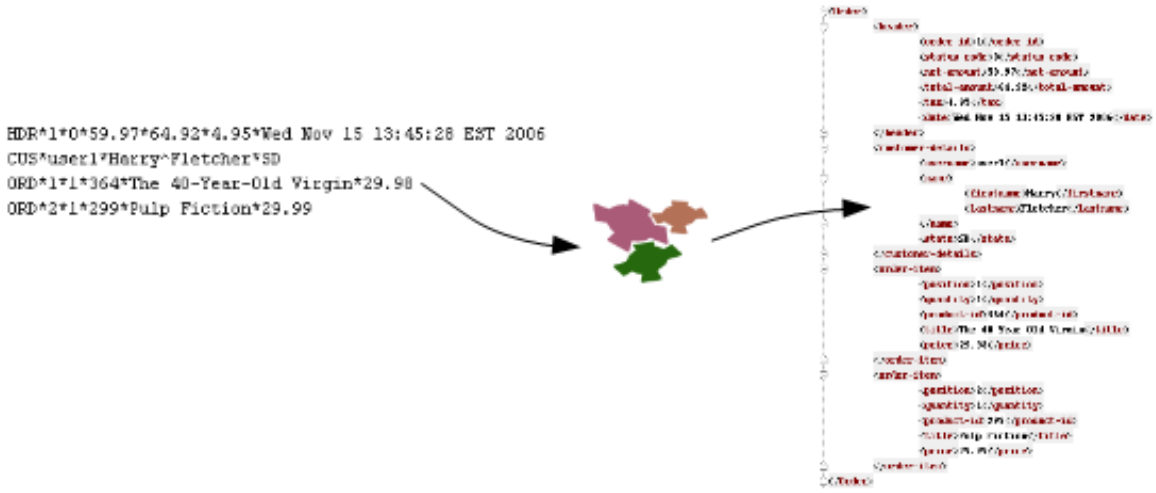

Figure 1.1. Transformation

Java Binding

This feature is used to populate a *Java Object Model* from a data source (such as a CSV, EDI, XML or Java file.) The populated object models can then be used either as transformation results in themselves or, alternatively, as a templating resources from which XML or other character-based results can be generated. This feature also supports *Virtual Object Models* (maps and lists of typed data), which can be used by both the ETL and templating functionality.

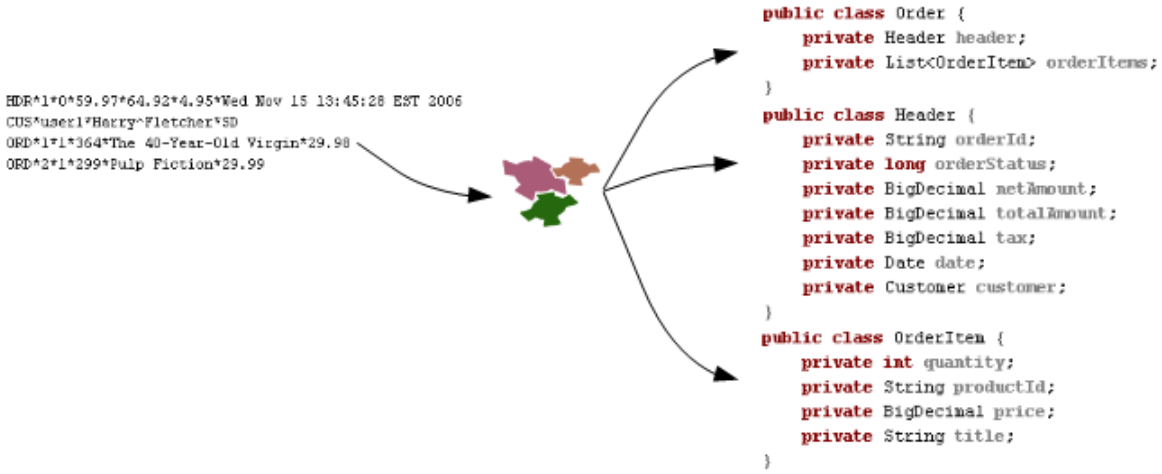

Figure 1.2. Java Binding

Huge Message Processing

This feature is used to process very large messages (which may be gigabytes in size.) It can split, transform and route message fragments to a variety of destinations, be they Java Message Service, file or database locations.

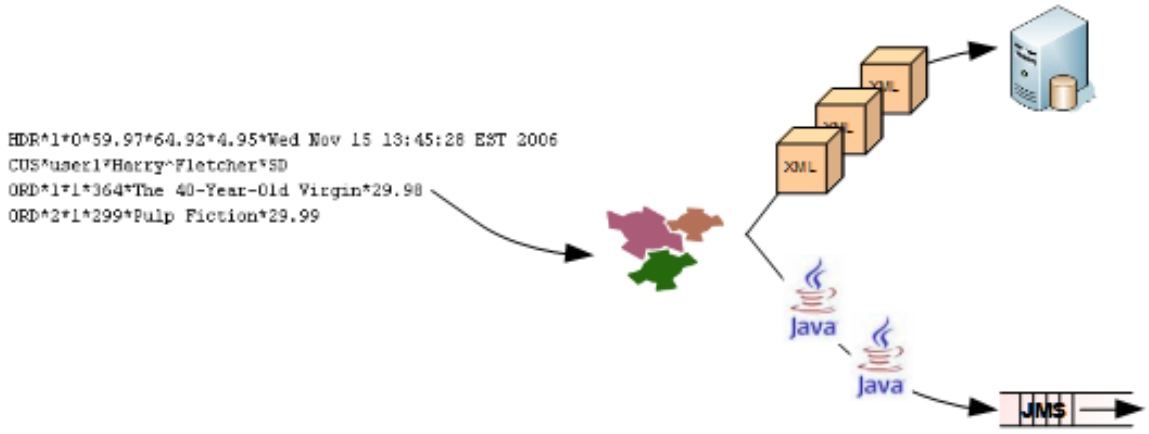

Figure 1.3. Huge Message Processing

Message Enrichment

This feature is used to "enrich" a message with information from a database or some other type of data-source.

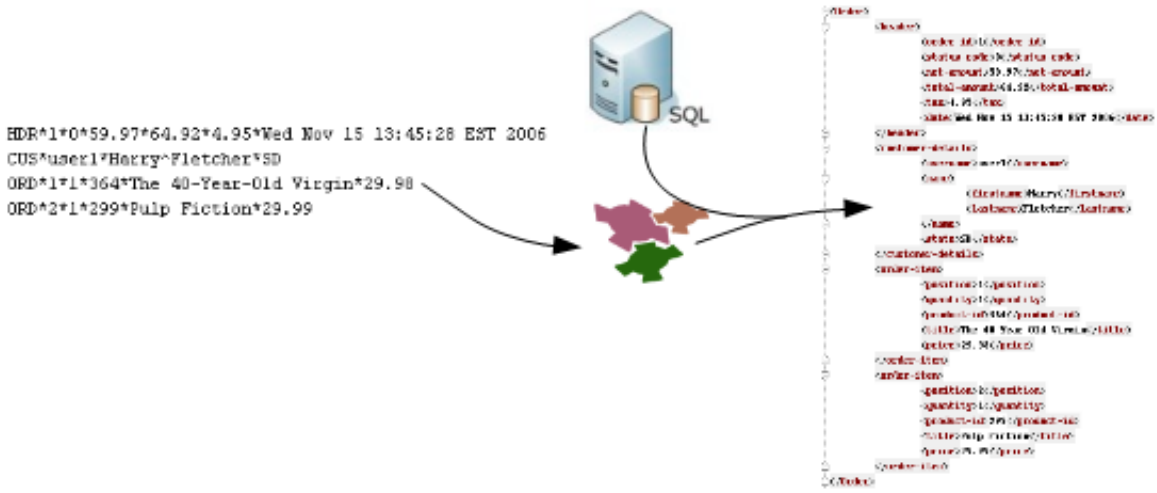

Figure 1.4. Message Enrichment

Complex Message Validation
: This is a *rules-based fragment validation* feature.

Object-Relational Mapping (ORM)-Based Message Persistence
: This feature uses a *Java Persistence*API(JPA)-compatible entity-persistence framework (like **Ibatis** or **Hibernate**) in order to access a database. It uses either the database's own query language or CRUD (Create, Read, Update and Delete) methods in order to make it read from, and write to, itself.

  This functionality can also use custom *Data Access Objects* (DAOs) to access a database and then, in the same way, use its CRUD methods to read and write to itself.

Combine
: This feature is used to perform *Extract Transform Load* (ETL) operations. It does so by leveraging Smooks' *transformation*, *routing* and *persistence* features.

1.1. Getting Started

The easiest way in which to become familiar with Smooks is to download and try out some of the examples at *http://www.smooks.org/mediawiki/index.php?title=Smooks_v1.2_Examples*. These are a good base upon which to learn how to integrate Smooks into custom applications.

1.2. Frequently Asked Questions

The answers to the most frequently asked questions about Smooks can be found at *http://www.smooks.org/mediawiki/index.php?title=FAQ*.

1.3. Maven

Detail instructions on how to integrate Smooks into a project using **Maven** are provided in the *Maven and Ant Guide* at: *http://www.smooks.org/mediawiki/index.php?title=Maven_%26_Ant*.

1.4. Ant

Detailed instruction on how to integrate Smooks into a project using **Ant** are provided in the *Maven and Ant Guide* at: *http://www.smooks.org/mediawiki/index.php?title=Maven_%26_Ant*.

Basics

The most commonly-accepted definition of Smooks, is that it is a "Transformation Engine." However, the core of Smooks does not refer to "data transformation". The **`smooks-core`** code is designed to support the "hooking" custom *Visitor Logic* into the *Event Stream* that is produced from a data-source.

`smooks-core` is a *Structured Data Event Stream Processor*.

The concept of Visitor logic is central to how Smooks works. A Visitor is a simple piece of Java logic that can perform a specific task on the message fragment at which it is targeted, such as applying an XSLT stylesheet. You have the choice of supporting your logic through either the SAX or DOM Filters by implementing one or both of the following interfaces: `org.milyn.delivery.sax.SAXElementVisitor` or `org.milyn.delivery.dom.DOMElementVisitor`.

The most common application of this functionality is the creation of transformation solutions, implementing Visitor Logic using the Event Stream produced from the *source* message to create a *result* of some other kind. However, the capabilities of the **`smooks-core`** allows much more than this common use case. Other solutions based on this processing model include:

- *Java Binding*: This is the ability to populate a *Java Object Model* from the source message.
- *Message Splitting and Routing*: This the ability to perform complex splitting and routing operations on the source message. It includes the ability to route to multiple destinations concurrently. Different data formats can also be routed concurrently. These formats include XML, EDI, CSV and Java.
- *Huge Message Processing*: This the ability to be able to declaratively "consume" (that is, transform or split-and-route) very large messages without the need for one to write substantial amounts of high-maintenance code.

2.1. Basic Processing Model

To reiterate, the basic principal of Smooks is to take a data source of some kind (such as XML) and, from it, generate an "Event Stream," to which can be applied Visitor Logic in order to produce a "result" such as an *Electronic Data Exchange* (EDI.)

Many different types of data source and results are supported, with the result that a number of different transformation types are available. Some of the more common examples include, but are in no way limited to:

- XML to XML
- XML to Java
- Java to XML
- Java to Java
- EDI to XML
- EDI to Java
- Java to EDI
- CSV to XML

Smooks supports the *Document Object Model* (DOM) and *Simple API for XML* (SAX) event models. These event models are used to map between the source and the result. The SAX event model will be given the most attention in this document. Refer to the Smooks Developer Guide at *http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_Developer_Guide* for additional information on these models.

The SAX event model is based on the hierarchical SAX events that can be generated from an XML source. E.g. **`startElement`** and **`endElement`**. This event model can be easily applied to other structured or hierarchical data sources, including EDI, CSV and Java.

Usually, the most important events are those entitled **`visitBefore`** and **`visitAfter`**. The following illustration conveys the hierarchical nature of these two events:

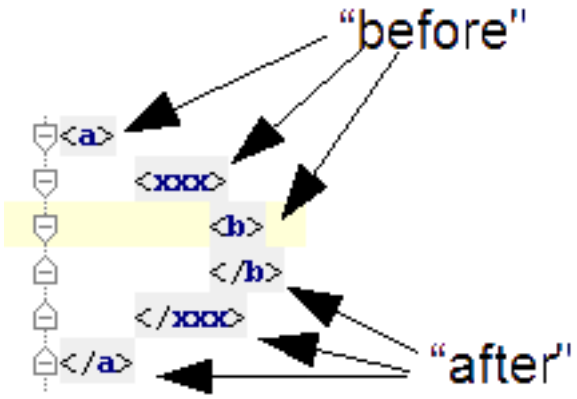

Figure 2.1. Hierarchical nature of the visitBefore and visitAfter events

2.2. Simple Example

In order to be able to "consume" the SAX Event Stream produced from the source message, you must implement one or more of the *SAXVisitor* interfaces. These are described in more detail at *http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/delivery/sax/SAXVisitor.html*. Which specific interfaces one should chose depends upon the events that need to be consumed.

The following is a very simple example that shows how to implement Visitor Logic. It demonstrates how to target that logic at the **`visitBefore`** and **`visitAfter`** events at a specific element in the overall Event Stream. In this case, the Visitor Logic is targeted at the events for the <xxx> element.

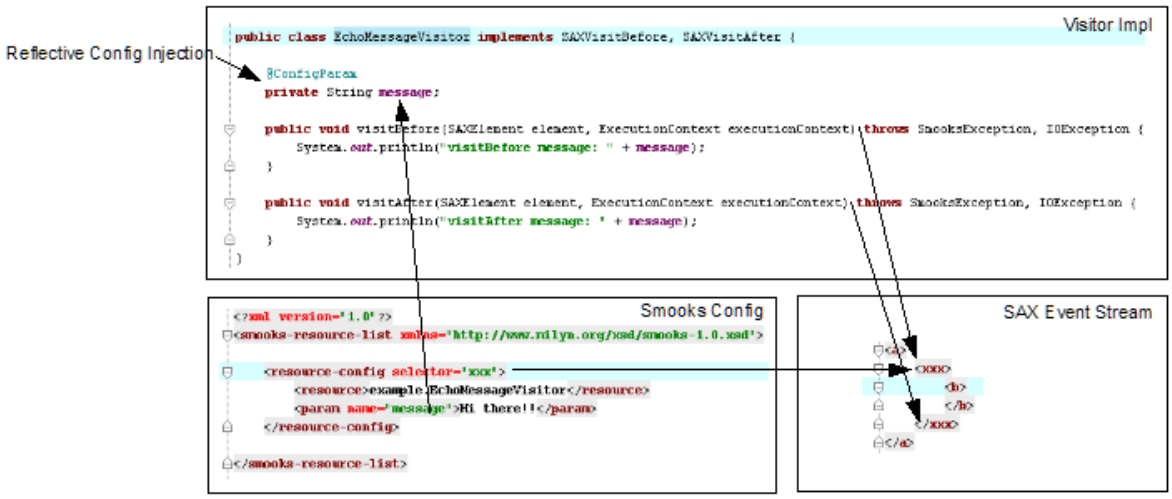

Figure 2.2. Implementing Visitor Logic

The Visitor implementation is very simple and consists of one method implementation per event. In order to target this implementation at the <xxx> element's `visitBefore` and `visitAfter` events, you need to create a Smooks configuration as shown below.

```
Smooks smooks = new Smooks("/smooks/echo-example.xml");

smooks.filterSource(new StreamSource(inputStream));
```

In this example, no result is produced. There is also no interaction with the execution of the filtering process. This is because the example didn't create an **`EvolutionContext`** and supply it to the `Smooks.filterSource` method call. See *http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/container/ExecutionContext.html*

This example demonstrated the lower-level workings of the Smooks programming model. In most cases, users do not need to write large quantities of Java code for Smooks. Smooks includes modules of pre-built functionality for many common use cases. These modules are called Cartridges. These are detailed in *Section 2.3, “Smooks Cartridges”*

2.3. Smooks Cartridges

Smooks includes pre-built and ready to use visitor logic so that users can implement solutions with minimal java code. This visitor logic is bundled into categories. These bundles are called *cartridges*.

A cartridge is a *Java Archive* (JAR) that contains reusable *Content Handlers*. The basic functionality of the **`smooks-core`** can also be extended through the creation of new cartridges. A Smooks cartridge should provide "ready-to-use" support for either a specific type of XML analysis or a transformation.

A full list of the cartridges supported by Smooks is available at: *http://www.smooks.org/cartri*.

2.4. Filtering Process Selection (Should DOM or SAX be Used?)

Smooks performs *Filtering Process Selection* based on the following criteria:

This does not include non-element Visitor resources, like, for example, *readers*.

- If all of the Visitor resources implement only DOM Visitor interfaces (`DOMElementVisitor` or `SerializationUnit`), then the DOM processing model is selected.
- If all of the Visitor resources implement only the SAX Visitor interface (`SAXElementVisitor`), then the SAX processing model would be selected.
- If all of the Visitor resources implement both the DOM and SAX interfaces, then the DOM processing model is selected by default, unless overridden by the *`stream.filter.type`* global configuration parameter.

The *`stream.filter.type`* can be set to either DOM or SAX. The following is an example that shows how to set it to SAX:

```
<params>
    <param name="stream.filter.type">SAX</param>
</params>
```

More information about global parameters can be found in *Section 15.2, "Global Configuration Parameters"*.

2.4.1. Mixing the DOM and SAX Models

The Document Object Model has advantages and disadvantages.

It has the advantage of being easier to use at the level of coding, as it allows *node traversal* and other features. Using the Document Object Model also allows one to take advantage of some pre-existing scripting and templating engines, such as FreeMarker and Groovy, have built-in support of DOM structures.

However, the Domain Object Model has the disadvantage of being constrained by memory. This limits its usefulness for dealing with very large messages.

Support for mixing these two models was added in **Smooks v1.1** via the **`DomModelCreator`** class. When it is used with SAX filtering, this Visitor will construct a DOM fragment from the visited element. This allows one to use DOM utilities within a streaming environment.

When more than one model is nested inside each other, the outer models will never contain data from the inner models; in other words, the same fragment will never co-exist inside two models. The following example message demonstrates this principle:

```
<order id="332">
    <header>
        <customer number="123">Joe</customer>
    </header>
    <order-items>
        <order-item id='1'>
            <product>1</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
```

```
        <order-item id='2'>
            <product>2</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
        <order-item id='3'>
            <product>3</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
    </order-items>
</order>
```

The **`DomModelCreator`** can be configured within Smooks to be able to create models for both the "order" and "order-item" message fragments, as follows:

```
<resource-config selector="order,order-item">
    <resource>org.milyn.delivery.DomModelCreator</resource>
</resource-config>
```

In this case, the "order" model will never contain model data for the "order-item" (because the order-item elements are nested inside the order element.) The in-memory model for the "order" will be:

```
<order id='332'>
     <header>
         <customer number="123">Joe</customer>
     </header>
     <order-items />
</order>
```

There will never be more than one "order-item" model in memory at any given time. Each new model overwrites the previous one. The system was designed this way in order to ensures that the amount of memory being used is kept to a minimum.

The Smooks processing model is event-driven (that is, one can "hook in" the Visitor logic that will be applied at different points of the process of filtering and streaming the message. It is applied via the message's own content.) Due to this fact, one can take advantage of the mixed DOM and SAX processing model.

See the following examples, as they utilize this mixed DOM and SAX approach:

- Groovy Scripting: *http://www.smooks.org/mediawiki/index.php?title=V1.2:groovy*
- FreeMarker Templating: *http://www.smooks.org/mediawiki/index.php?title=V1.2:xml-to-xml*

2.5. Checking the Smooks Execution Process

As Smooks performs the process of *filtering* (in other words, as it processes the event stream generated from the source), it publishes those events that can be captured and programmatically-

analyzed during and after execution. The easiest way to obtain an execution report from Smooks is to configure the **`ExecutionContext`** to generate it. Smooks also supports the generation of an HTML report via the **`HtmlReportGenerator`** class.

The following example demonstrates how to configure Smooks to generate an HTML report:

```
Smooks smooks = new Smooks("/smooks/smooks-transform-x.xml");
ExecutionContext execContext = smooks.createExecutionContext();

execContext.setEventListener(new HtmlReportGenerator("/tmp/smooks-
report.html"));
smooks.filterSource(execContext, new StreamSource(inputStream), new
 StreamResult(outputStream));
```

The **`HtmlReportGenerator`** is a tool which one will find very useful whilst undertaking development work with Smooks. It is the closest thing that Smooks has, at present, to an IDE-based debugger. (A debugger will be included in a future release). The **`HtmlReportGenerator`** tool is very useful when one is trying to diagnose issues. It is also helpful when one is simply trying to comprehend an aspect of a particular Smooks transformation.

An example of an **`HtmlReportGenerator`** report is provided on this web page: *http://www.milyn.org/docs/smooks-report/report.html*

Of course, a custom **`ExecutionEventListener`** implementation can also be created. Read this page for more information: *http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/event/ExecutionEventListener.html*

Smooks Resources

3.1. An Introduction to Smooks Resources

Smooks operates by taking a data stream of one form or another (such as XML, EDI, Java, JSON or CSV) and generating an event stream from it. This event stream is then used by it to fire different types of Visitor Logic (which might be Java, **Groovy**, **FreeMarker** or XSLT.) The result of this process can take a number of forms. It can result in:

- the production of a new data stream in a different format (this is a traditional *transformation*);
- the binding of data from the source message data stream to Java objects in order to produce a populated *Java Object Graph* (by *Java Binding*);
- the production of many smaller messages (by *message splitting*.)

Smooks, at its core, simply regards all of the Visitor Logic as Smooks Resources (**`SmooksResourceConfiguration`**), ready-configured to be applied and based on an event selector (from the source data event stream.) This is a very generic processing model. It made a lot of sense to develop the **`smooks-core`** and its maintenance architecture in this way. However, one may find that it is a little too generic from a usability perspective. This is because everything looks very similar in the configuration. To resolve this issue, a feature known as the *Extensible Configuration Model* was introduced in **Smooks v1.1**. This feature allows for specific resource types (such as Java Bean bindings and **FreeMarker** template configurations) to be specified and to have dedicated XSD name-spaces of their own.

```
<jb:bean beanId="lineOrder" class="example.trgmodel.LineOrder"
 createOnElement="example.srcmodel.Order">
    <jb:wiring property="lineItems" beanIdRef="lineItems" />
    <jb:value property="customerId" data="header/customerNumber" />
    <jb:value property="customerName" data="header/customerName" />
</jb:bean>
```

Example 3.1. Example (Java Binding Resource)

```
<ftl:freemarker applyOnElement="order-item">
    <ftl:template>
    </ftl:template>
</ftl:freemarker>
```

Example 3.2. Example (FreeMarker Template Resource)

When one compares the above examples to their equivalents in pre-**Smooks v1.1** releases, it can be seen that:

- The user now has a more strongly-"typed" configuration that is, in each case, domain-specific (and so, therefore, easier to read.)

- Because the configurations in v1.1 and onwards are XSD-based, the user also gains auto-completion support within his or her IDE.
- There is no longer any need to define the actual handler for the given resource type (such as the **`BeanPopulator`** for Java bindings.)

Java Binding

The Smooks *JavaBean Cartridge* facilitates the creation and population of Java objects from both message and *bind* data.

4.1. Java Binding Overview

This Smooks feature can be used in its own right as a Java Binding framework for XML, EDI, CSV and so forth. However, it is very important to remember that the *Java Binding* functionality is the cornerstone of many of the other capabilities that Smooks provides. This is because Smooks makes the Java Objects that it creates and binds data into available through the BeanRepository class. (Please see this website for more information: *http://www.milyn.org/javadoc/v1.2/smooks-cartridges/javabean/org/milyn/javabean/repository/BeanRepository.html*.) This is, essentially, a Java Bean Context that is made available to any Smooks Visitor implementation via the ExecutionContext. (Please see this website for more information: *http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/container/ExecutionContext.html*.)

Some of the features that build on the existing functionality provided in the Java Bean Cartridge include:

- *Templating*: templating, as its name implies, typically involves applying a template (**FreeMarker** or other) to the objects that are in the **`BeanRepository`**.
- *Validation*: business rules validation (such as that via the MVEL) involves applying a rule or expression to the objects that are in the **`BeanRepository`**.
- *Message Splitting and Routing*: this feature works by generating split messages from the objects in the **`BeanRepository`**, either by using and routing the objects themselves, or by applying a template to them and routing the result of that operation (which might, for instance, be a new XML or CSV file.)
- *Persistence* (Database Reading and Writing): the "persistence" features depend upon the *Java Binding* functions for the creation and population of the Java objects such as entities that are to be persisted. Data that has been read from a database is normally bound into the **`BeanRepository`**.
- *Message Enrichment*: as stated above, "enriched" data (for example, that which is read from a database) is typically bound to the **`BeanRepository`**. From there, it is available to all other features, including the Java Binding functionality itself. From there, it might be used by such things as *expression-based bindings*. Therefore, messages generated by Smooks can be enriched.

4.2. Situations in Which to Use Smooks Java Binding

Users are often unsure of when to Smooks, rather than an alternative tool, to bind data to a Java Object model. There are certain situations for which Smooks is ideal and others in which it should be avoided. These will now be examined.

It would make sense to use Smooks in these scenarios:

- When binding non-XML data to a Java Object model, such as an EDI, CSV or JSON.
- When binding data (be it XML or some other type), the data model or hierarchical structure of which does not match that of the target Java Object model. An alternative program, JiBX (*http://jibx.sourceforge.net/*) also supports this, but only for XML.

- When one is binding data from an XML data structure for which there is no defined schema (or XSD.) Some frameworks effectively require a well-defined XML data model via such a schema.
- When binding data from a number of existing but different data formats into a single pre-existing Java Object model. This is related to the points outlined above.
- When binding data into existing third-party object models that one cannot modify by, for example, a post-compile step.
- In situations where the data and Java Object models may vary in isolation from each other. Because of Point Two above, Smooks can handle this simply by modifying the binding configuration. Alternative frameworks often require a binding/schema regeneration and redeployment and so forth (see Point Three above.)
- In situations whereby one needs to execute additional logic in parallel to the binding process. This could include, for example, Validation, Split Message Generation (via Templates), Split Message Routing, Fragment Persistence, or any other form of custom logic that one may wish to implement. This is often a very powerful capability in situations such as that in which one is processing huge message streams.
- When dealing with huge message streams by splitting them into a series of many small object models and routing these to other systems for processing.
- When using other Smooks features that rely on the Smooks Java Binding capabilities.

Situations in which the use of Smooks may not make sense include the following:

- When there is a well-defined data model (via schema/XSD) and all one needs to do is bind data into an object model (without the need for any validation, or persistence.)
- When the object model is isolated from other systems and can, as a result, change without impacting said systems.
- In situations in which the processing of XML and high-level performance is paramount over all other considerations (in other words, in situations where even mere nanoseconds matter), frameworks such as JiBX (*http://jibx.sourceforge.net/*) are definitely worth the giving of consideration over Smooks. This is not to imply that the performance of the Smooks Java Binding is, in any way, poor but it does acknowledge the fact that frameworks that utilise post-compile optimizations targeted at a specific data format such as XML will always have the advantage in the right conditions.

4.3. Basics Of Java Binding

As the reader will be well aware by now, Smooks supports a wide range of source data formats but, for the sake of simplicity, this chapter will always illustrate topics with XML examples. In fact, most of the following examples will relate to this one sample XML message:

```
<order>
    <header>
        <date>Wed Nov 15 13:45:28 EST 2006</date>
        <customer number="123123">Joe</customer>
    </header>
    <order-items>
        <order-item>
            <product>111</product>
```

```
            <quantity>2</quantity>
            <price>8.90</price>
        </order-item>
        <order-item>
            <product>222</product>
            <quantity>7</quantity>
            <price>5.20</price>
        </order-item>
    </order-items>
</order>
```

Note

A few of the examples do use different XML samples. The data is explicitly defined in these cases.

The *JavaBean* cartridge is used via the configuration name-space defined on this website: *http://www.milyn.org/xsd/smooks/javabean-1.2.xsd*. Once the schema is installed in an IDE, one can avail oneself of auto-complete functionality.

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.2.xsd">

    <jb:bean beanId="order" class="example.model.Order"
 createOnElement="#document" />

</smooks-resource-list>
```

Example 4.1. An Example Configuration

This configuration simply creates an instance of the **`example.model.Order`** class and binds it into the Bean Context under the beanId "order." The instance is created at the very start of the message on the #document element (this is at the start of the root <order> element.)

- beanId: this is the bean's identification. Please see the section of this book on the "The Bean Context" for more details.
- class: this is the bean's fully-qualified classname.
- createOnElement: this attribute dictates the time at which the bean instance is to be created. Population of the bean properties is controlled through the *binding configurations* (child elements of the <jb:bean> element).
- createOnElementNS: the name space of the createOnElement can be specified via this attribute.

4.3.1. The Bean Context

A *bean context* (also known as the *Bean Map*) is an important part of a cartridge. One bean context is created per execution context (in other words, per **`Smooks.filterSource`** operation). Every

bean, created by the cartridge, is put into this context under its beanId. If one wants the contents of the bean context to be returned at the end of the **`Smooks.filterSource`** process, one should supply a **`org.milyn.delivery.java.JavaResult`** object in the call to **`Smooks.filterSource`** method. The following example illustrates this principal:

```
//Get the data to filter
StreamSource source = new
 StreamSource(getClass().getResourceAsStream("data.xml"));

//Create a Smooks instance (cachable)
Smooks smooks = new Smooks("smooks-config.xml");

//Create the JavaResult, which will contain the filter result after
 filtering
JavaResult result = new JavaResult();

//Filter the data from the source, putting the result into the JavaResult
smooks.filterSource(source, result);

//Getting the Order bean which was created by the JavaBean cartridge
Order order = (Order)result.getBean("order");
```

If access to the beans within the context is needed at run-time (from within a customer's Visitor implementation), use the **`BeanRepository`** class.

The cartridge dictates the following conditions for Java Beans. There must be:

- A public *no-argument constructor*;
- *Public property setter* methods. The do not need to follow any specific name formats but it would be better if they do follow that which exists for standard property setter methods.

Note

The ability to set Java Bean properties directly is not supported.

4.4. Java Binding Configuration Details

The configuration shown above simply created the **`example.model.Order`** instance and bound it into the bean context. This section will describe how to bind data into that bean instance.

The Java Bean Cartridge provides support for thee types of data binding. These are added as child elements of the <jb:bean> element:

- jb:value: this is used to bind the data values from the source message event stream into the target bean.
- jb:wiring: This is used to "wire" another bean instance from the context into a property on the target bean. This is the configuration that allows one to construct an object graph (as opposed to a loose collection of Java object instances.)

- jb:expression: As its name suggests, this configuration is used to bind in a value calculated from an expression. A simple example is that of binding an order item's total cost into an OrderItem bean. This would be based on the result of an expression that calculates the total cost by multiplying the item's price by the quantity ordered.

The full XML-to-Java binding configuration for the order XML message can now be studied. Firstly, one shall examine the Java Objects with which the XML message shall be populated. (Note that the "getters" and "setters" are not shown):

```
public class Order {
    private Header header;
    private List<OrderItem> orderItems;
}

public class Header {
    private Date date;
    private Long customerNumber;
    private String customerName;
    private double total;
}

public class OrderItem {
    private long productId;
    private Integer quantity;
    private double price;
}
```

The Smooks configuration that is required to bind the data from the order XML to this object model is as follows:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.2.xsd">

(1)   <jb:bean beanId="order" class="com.acme.Order"
 createOnElement="order">
(1.a)     <jb:wiring property="header" beanIdRef="header" />
(1.b)     <jb:wiring property="orderItems" beanIdRef="orderItems" />
      </jb:bean>

(2)   <jb:bean beanId="header" class="com.acme.Header"
 createOnElement="order">
(2.a)     <jb:value property="date" decoder="Date" data="header/date">
              <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z yyyy</
jb:decodeParam>
          </jb:value>
(2.b)     <jb:value property="customerNumber" data="header/customer/
@number" />
(2.c)     <jb:value property="customerName" data="header/customer" />
(2.d)     <jb:expression property="total" execOnElement="order-item" >
```

```
                header.total + (orderItem.price * orderItem.quantity);
            </jb:expression>
        </jb:bean>

(3)     <jb:bean beanId="orderItems" class="java.util.ArrayList"
 createOnElement="order">
(3.a)       <jb:wiring beanIdRef="orderItem" />
        </jb:bean>

(4)     <jb:bean beanId="orderItem" class="com.acme.OrderItem"
 createOnElement="order-item">
(4.a)       <jb:value property="productId" data="order-item/product" />
(4.b)       <jb:value property="quantity" data="order-item/quantity" />
(4.c)       <jb:value property="price" data="order-item/price" />
        </jb:bean>

</smooks-resource-list>
```

1	Configuration "'(1)'" defines the creation rules for the "com.acme.Order" bean instance (top level bean). We create this bean instance at the very start of the message i.e. on the <order> element . In fact, we create each of the beans instances ("'(1)'", "'(2)'", "'(3)'" - all accepts "'(4)'") at the very start of the message (on the <order> element). We do this because there will only ever be a single instance of these beans in the populated model. Configurations "'(1.a)'" and "'(1.b)'" define the "'wiring'" configuration for wiring the "Header" and "List <OrderItem>" bean instances ("'(2)'" and "'(3)'") into the Order bean instance (see the "'beanIdRef'" attribute values and how the reference the "'beanId'" values defined on "'(2)'" and "'(3)'"). The "'property'" attributes on "'(1.a)'" and "'(1.b)'" define the "Order" bean properties on which the wirings are to be made.
2	Configuration "'(2)'" creates the "com.acme.Header" bean instance. Configuration "'(2.a)'" defines a "'value'" binding onto the "Header.date" property. Note that the "'data'" attribute defines where the binding value is selected from the source message; in this case it is coming from the header/date element. Also note how it defines a "'decodeParam'" sub-element. This configures the *http://www.milyn.org/javadoc/v1.2/commons/org/milyn/javabean/decoders/DateDecoder.html*. Configuration "'(2.b)'" defines a "'value'" binding configuration onto "Header.customerNumber" property. What should be noted here is how to configure the "'data'" attribute to select a binding value from an element attribute on the source message. Configuration "'(2.b)'" also defines an "'expression'" binding where the order total is calculated and set on the "Header.total" property. The "'execOnElement'" attribute tells Smooks that this expression needs to be evaluated (and bound/rebound) on the <order-item> element. So, if there are multiple <order-item> elements in the source message, this expression will be executed for each <order-item> and the

	new total value rebound into the "Header.total" property. Note how the expression adds the current orderItem total to the current order total (header.total).
3	Configuration "'(3)"' creates the "List OrderItem" bean instance for holding the "OrderItem" instances. Configuration "'(3.a)"' wires in the orderItem bean ("'(4)"') instances into the list. Note how this wiring does not define a "'property"' attribute. This is because it wires into a Collection (same applies if wiring into an array.)
4	Configuration "'(4)"' creates the "OrderItem" bean instances. Note how the "'createOnElement"' is set to the <order-item> element. This is because we want a new instance of this bean to be created for every <order-item> element (and wired into the "List <OrderItem>" "'(3.a)"'.) If the "'createOnElement"' attribute for this configuration was not set to the order-item element (e.g. if it was set to one of the <order>, <header> or <order-items> elements), then only a single "OrderItem" bean instance would be created and the binding configurations ("'(4.a)"' etc) would overwrite the bean instance property bindings for every <order-item> element in the source message i.e. you would be left with a "List<OrderItem>" with just a single "OrderItem" instance containing the <order-item> data from the last <order-item> encountered in the source message.

Binding Tips:

- jb:bean createOnElement

  Set it to the root element (or **`#document`**) for bean instances in which only a single instance will exist in the model.

  Set it to the recurring element for collection bean instances. If one does not specify the correct element in this case, data could be lost.

- jb:value decoder

  In most cases, Smooks will automatically detect the data-type decoder to be used for a jb:value binding. However, some decoders require configuration, an example being the date decoder. In these cases, the decoder attribute should be defined on the binding, as well as on the <jb:decodeParam> child elements (to specify the decode parameters.) A full list of the data decoders that are available "out-of-the-box" can be found here: *http://www.milyn.org/javadoc/v1.2/commons/org/milyn/javabean/decoders/package-summary.html*.

- jb:wiring property

  This is not required when binding into collections.

- jb:expression execOnElement

  This explicitly tells Smooks when the expression is to be evaluated and the result bound. If it is not defined, the expression is executed based on the value of the parent <jb:bean createOnElement>.

- Collections

  One must just define the **`jb:bean`** class for the required collection type and wire in the collection entries.

For arrays, one should just post-fix the **`jb:bean`** class attribute value with square brackets, so that, for instance, it looks like this: `class="com.acme.OrderItem[]"`.

4.4.1. Extended Life-Cycle Binding

4.4.1.1. Binding Key Value Pairs into Maps

If a binding's <jb:value property> attribute is either undefined or empty, then the name of the selected node will be used as the map entry key (where the beanClass is a map.)

There is one other way in which to define the map key. The value of the <jb:value property> attribute can start with the @ character. The rest of the value then defines the selected node's attribute name, from which the map key is selected. The following example demonstrates this:

```
<root>
    <property name="key1">value1</property>
    <property name="key2">value2</property>
    <property name="key3">value3</property>
</root>
```

Here is the configuration:

```
<jb:bean beanId="keyValuePairs" class="java.util.HashMap"
 createOnElement="root">
    <jb:value property="@name" data="root/property" />
</jb:bean>
```

This would create a hash map with three entries against the keys that have been set (key1, key2 and key3.)

Of course the @ character notation does not work for bean wiring. The cartridge will simply use the value of the property attribute, including the @ character, as the map entry key.

4.4.2. Virtual Object Models (Maps and Lists)

It is possible to create a complete object model without writing one's own Bean classes. This virtual model is created using only maps and lists. This is very convenient if one is using the Java Bean cartridge in between two processing steps, such as during part of a model-driven transformation (XML->Java->XML or XML->Java->EDI.)

The following example demonstrates the principle:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/
javabean-1.2.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/
freemarker-1.1.xsd">
```

```
    <!--
    Bind data from the message into a Virtual Object model in the bean
 context....
    -->
    <jb:bean beanId="order" class="java.util.HashMap"
 createOnElement="order">
        <jb:wiring property="header" beanIdRef="header" />
        <jb:wiring property="orderItems" beanIdRef="orderItems" />
    </jb:bean>
    <jb:bean beanId="header" class="java.util.HashMap"
 createOnElement="order">
        <jb:value property="date" decoder="Date" data="header/date">
            <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z yyyy</
jb:decodeParam>
        </jb:value>
        <jb:value property="customerNumber" decoder="Long" data="header/
customer/@number" />
        <jb:value property="customerName" data="header/customer" />
        <jb:expression property="total" execOnElement="order-item" >
            header.total + (orderItem.price * orderItem.quantity);
        </jb:expression>
    </jb:bean>
    <jb:bean beanId="orderItems" class="java.util.ArrayList"
 createOnElement="order">
        <jb:wiring beanIdRef="orderItem" />
    </jb:bean>
    <jb:bean beanId="orderItem" class="java.util.HashMap"
 createOnElement="order-item">
        <jb:value property="productId" decoder="Long" data="order-item/
product" />
        <jb:value property="quantity" decoder="Integer" data="order-item/
quantity" />
        <jb:value property="price" decoder="Double" data="order-item/
price" />
    </jb:bean>

    <!--
    Use a FreeMarker template to perform the model driven transformation on
 the Virtual Object Model...
    -->
    <ftl:freemarker applyOnElement="order">
        <ftl:template>/templates/orderA-to-orderB.ftl</ftl:template>
    </ftl:freemarker>

</smooks-resource-list>
```

Note that one should always define the decoder attribute for a Virtual Model (map). This is because Smooks has no way of auto-detecting the decode type for the purpose of data binding to a map. So, if one needs typed values bound into a Virtual Model, one has to specify an appropriate decoder. If the decoder is not specified then Smooks will simply bind the data into the Virtual Model as a string.

Study the examples found at this web address: *http://www.smooks.org/mediawiki/index.php?title=Smooks_v1.2_Examples*.

4.5. Merging Multiple Data Entities Into a Single Binding

This can be achieved by using *Expression Based Bindings* (<jb:expression>.)

4.6. Generating the Smooks Binding Configuration

The Javabean Cartridge contains the **`org.milyn.javabean.gen.ConfigGenerator`** utility class. This can be used to generate a *binding configuration template*. As its name implies, this template can be used as the basis from which a binding can be defined.

From the command-line:

```
$JAVA_HOME/bin/java -classpath <classpath>
 org.milyn.javabean.gen.ConfigGenerator -c <rootBeanClass> -o
 <outputFilePath> [-p <propertiesFilePath>]
```

- The **`-c`** command-line argument specifies the root class of the model whose binding configuration is to be generated.
- The **`-o`** command-line argument specifies the path and filename for the generated configuration output.
- The **`-p`** command-line argument specifies the path and filename optional binding configuration file that specifies additional binding parameters.

The optional **`-p`** properties file parameter allows specification of additional configuration parameters:

- **`packages.included`**: This is a semi-colon separated list of packages. Any fields in the class that match these packages will be included in the binding configuration that is generated.
- **`packages.excluded`**: This is a semi-colon separated list of packages. Any fields in the class that match these packages will be excluded from the binding configuration that is generated.

After running this utility against the target class, one needs to perform the following tasks in order to make the binding configuration work with one's source data model.

- For each jb:bean element, set the createOnElement attribute to the event that should be used. This is in order to create the bean instance.
- Update the jb:value data attributes to select the event element or attribute that supplies the binding data for that bean property.
- Check the jb:value decoder's attributes. It may be that not all of these will be set, depending on the actual property type. These must be configured by hand as one may need to configure <jb:decodeParam>'s sub-elements for the decoder on some of the bindings, such as that for a date field.
- Double-check the binding configuration's elements (<jb:value> and <jb:wiring>), in order to make sure that all of the Java properties have been accounted for by the configuration that has been generated.

Determining the selector values can sometimes be difficult, especially for non-XML sources such as Java. The **HTML Reporting Tool** can be a great help in this situation, because it assists one in visualising the input message model against which the selectors will be applied. It is this model which is seen by Smooks. To commence, one must generate a report using the Source data, albeit with an empty transformation configuration file. In the report, one can see the model against which there is a need to add the configurations. Do so one at a time, re-running the report each time in order to check taht they are being applied.

The following is an example of a generated configuration. Please note the `$TODO$` tokens.

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.2.xsd">

    <jb:bean beanId="order" class="org.milyn.javabean.Order"
 createOnElement="$TODO$">
        <jb:wiring property="header" beanIdRef="header" />
        <jb:wiring property="orderItems" beanIdRef="orderItems" />
        <jb:wiring property="orderItemsArray" beanIdRef="orderItemsArray" /
>
    </jb:bean>

    <jb:bean beanId="header" class="org.milyn.javabean.Header"
 createOnElement="$TODO$">
        <jb:value property="date" decoder="$TODO$" data="$TODO$" />
        <jb:value property="customerNumber" decoder="Long" data="$TODO$" />
        <jb:value property="customerName" decoder="String" data="$TODO$" />
        <jb:value property="privatePerson" decoder="Boolean" data="$TODO
$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

    <jb:bean beanId="orderItems" class="java.util.ArrayList"
 createOnElement="$TODO$">
        <jb:wiring beanIdRef="orderItems_entry" />
    </jb:bean>

    <jb:bean beanId="orderItems_entry" class="org.milyn.javabean.OrderItem"
 createOnElement="$TODO$">
        <jb:value property="productId" decoder="Long" data="$TODO$" />
        <jb:value property="quantity" decoder="Integer" data="$TODO$" />
        <jb:value property="price" decoder="Double" data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

    <jb:bean beanId="orderItemsArray"
 class="org.milyn.javabean.OrderItem[]" createOnElement="$TODO$">
        <jb:wiring beanIdRef="orderItemsArray_entry" />
    </jb:bean>
```

```
    <jb:bean beanId="orderItemsArray_entry"
 class="org.milyn.javabean.OrderItem" createOnElement="$TODO$">
        <jb:value property="productId" decoder="Long" data="$TODO$" />
        <jb:value property="quantity" decoder="Integer" data="$TODO$" />
        <jb:value property="price" decoder="Double" data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

</smooks-resource-list>
```

4.7. Programmatic Configuration

Java Binding Configuratons can be added to Smooks "programmatically" by using using the configuration class that can be found at: *http://www.milyn.org/javadoc/v1.2/smooks-cartridges/javabean/org/milyn/javabean/Bean.html*.

This can be used to configure a Smooks instance for the purpose of performing Java Bindings on a specific class. To populate a graph, one simply creates a chart of Bean instances. Do so by binding Beans onto other Beans. The Bean class uses a *Fluent* application programming interface (for which all methods return the Bean instance). This makes it easy to "string" configurations together. In this way, one can build up a graph of the Bean configuration.

4.7.1. An Example

This example takes the "classic" Order message and binds it into a corresponding Java Object model.

```
<order xmlns="http://x">
    <header>
        <y:date xmlns:y="http://y">Wed Nov 15 13:45:28 EST 2006</y:date>
        <customer number="123123">Joe</customer>
        <privatePerson></privatePerson>
    </header>
    <order-items>
        <order-item>
            <product>111</product>
            <quantity>2</quantity>
            <price>8.90</price>
        </order-item>
        <order-item>
            <product>222</product>
            <quantity>7</quantity>
            <price>5.20</price>
        </order-item>
    </order-items>
</order>
```

Example 4.2. The Message

```
public class Order {
    private Header header;
    private List<OrderItem> orderItems;
}

public class Header {
    private Long customerNumber;
    private String customerName;
}

public class OrderItem {
    private long productId;
    private Integer quantity;
    private double price;
}
```

Example 4.3. The Java Model (Not Including "Getters" and "Setters")

```
Smooks smooks = new Smooks();

Bean orderBean = new Bean(Order.class, "order", "/order");

orderBean.bindTo("header",
    orderBean.newBean(Header.class, "/order")
        .bindTo("customerNumber", "header/customer/@number")
        .bindTo("customerName", "header/customer")
    ).bindTo("orderItems",
    orderBean.newBean(ArrayList.class, "/order")
        .bindTo(orderBean.newBean(OrderItem.class, "order-item")
            .bindTo("productId", "order-item/product")
            .bindTo("quantity", "order-item/quantity")
            .bindTo("price", "order-item/price"))
    );

smooks.addVisitors(orderBean);
```

Example 4.4. The Configuration Code

```
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(orderMessageStream), result);
Order order = (Order) result.getBean("order");
```

Example 4.5. The Execution Code

4.8. Notes on "JavaResult"

Readers should note that there is "no guarantee" as to the exact contents of a *JavaResult* (*http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/payload/JavaResult.html*) instance after having called the **`Smooks.filterSource`** method. Rather, the **`JavaResult`** instance will contain the final contents of the bean context, to which further data can be added by any Visitor implementation.

One can restrict the Bean Set returned by a **`JavaResult`** by setting the Smooks configuration to use a <jb:result>. In the following example, one "tells" Smooks to retain only the "order" bean in the **`ResultSet`**:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/
javabean-1.2.xsd">

    <!-- Capture some data from the message into the bean context... -->
    <jb:bean beanId="order" class="com.acme.Order" createOnElement="order">
        <jb:value property="orderId" data="order/@id"/>
        <jb:value property="customerNumber" data="header/customer/@number"/
>
        <jb:value property="customerName" data="header/customer"/>
        <jb:wiring property="orderItems" beanIdRef="orderItems"/>
    </jb:bean>
    <jb:bean beanId="orderItems" class="java.util.ArrayList"
 createOnElement="order">
        <jb:wiring beanIdRef="orderItem"/>
    </jb:bean>
    <jb:bean beanId="orderItem" class="com.acme.OrderItem"
 createOnElement="order-item">
        <jb:value property="itemId" data="order-item/@id"/>
        <jb:value property="productId" data="order-item/product"/>
        <jb:value property="quantity" data="order-item/quantity"/>
        <jb:value property="price" data="order-item/price"/>
    </jb:bean>

    <!-- Only retain the "order" bean in the root of any final JavaResult.
 -->
    <jb:result retainBeans="order"/>

</smooks-resource-list>
```

After having applied this configuration, calls to the **`JavaResult.getBean(String)`** method for anything other than the "order" bean will return <null>. This will work correctly in cases such as that of the above example, because the other bean instances are "wired" into the "order" graph.

Note

Note that as of Smooks v1.2, if a *http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/payload/JavaSource.html*JavaSource instance is supplied to the **`Smooks.filterSource`** method (as the filter Source instance), Smooks will use the

JavaSource to construct the bean context associated with the *http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/container/ExecutionContext.html* for that method's invocation. As a result, some of the JavaSource bean instances may be visible in the JavaResult.

Templating

Smooks provides two main "templating" options:

- *http://freemarker.org* **FreeMarker** Templating
- *http://www.w3.org/Style/XSL/* XSL Templates

The distinguishing feature added here is the ability to use these templating technologies within the context of a Smooks filtering process. This means that they:

- can be applied to a source message on a per-fragment basis rather than to the whole message as is the case with Fragment-Based Transformations. This is useful in situations in where, for example, one only wishes to insert a piece of data into a message at a specific point. One such situation might be that of adding headers to a SOA-P message in which one does not wish to interfere with the rest of the message stream. In this case, the template can be aimed at the fragment of interest.
- can take advantage of other Smooks cartridge technologies such as the Javabean Cartridge. The latter can be used to decode and bind data from the message into the Smooks bean context. It can then use (reference) that decoded data from inside the **FreeMarker** template (as Smooks makes this data available to **FreeMarker**.)
- can be used to process huge message streams measuring gigabytes in size whilst, at the same time, maintaining a relatively simple processing model, with a low memory footprint. (See *http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Processing_Huge_Messages_.28GBs.29* for more information.)
- can be used to generate *split message fragments* that can then be routed (*http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing*.) to physical (file, Java Message Service) or logical end-points on an Enterprise Service Bus.

Note that Smooks can (and will) be extended in order to add support for other templating technologies.

Note

Be sure to read the section on Java Binding.

5.1. FreeMarker Templating

FreeMarker is a very powerful *Templating Engine*. Smooks allows **FreeMarker** to be used as a means of generating text-based content. This content can subsequently be inserted into a message stream (a *Fragment Transform*), or used as a Split Message Fragment for routing to another process. (See *http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing*.)

Smooks' **FreeMarker** templates are configured via the configuration name-space (*http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd*.) One simply needs to configure this XSD in one's IDE in order to use it.

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">
    <ftl:freemarker applyOnElement="order">
        <ftl:template><!--<orderId>${order.id}</orderId>--></ftl:template>
    </ftl:freemarker>
</smooks-resource-list>
```

Example 5.1. Example - Inline Template

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">
    <ftl:freemarker applyOnElement="order">
        <ftl:template>/templates/shop/ordergen.ftl</ftl:template>
    </ftl:freemarker>
</smooks-resource-list>
```

Example 5.2. Example - External Template Reference

Smooks allows one to perform a number of operations with the Templating result. This is achieved by the "'<use>'" element, which must be added to the <ftl:freemarker> configuration.

```
<ftl:freemarker applyOnElement="order">
    <ftl:template>/templates/shop/ordergen.ftl</ftl:template>
    <ftl:use>
        <ftl:inline directive="insertbefore" />
    </ftl:use>
</ftl:freemarker>
```

Example 5.3. Example - "Inlining" the Template Result

Inlining, as its name implies, allows one to "inline" the templating result to a **`Smooks.filterSource`** result. A number of so-called "directives" are supported:

- addto: this adds the templating result to the targeted element.
- replace (default): this uses the templating result to replace the targeted element. This is the default behavior for the <ftl:freemarker> configuration when the <use> element is not configured.
- insertbefore: this adds the templating result prior to the targeted element.
- insertafter: this adds the templating result after the targeted element.

By using <ftl:bindTo>, one can bind the templating result to the Smooks bean context. The templating result can then be accessed by other Smooks components, such as those used for routing. This can be especially useful for splitting huge messages into smaller ones (see *)http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing*[1].) They can

[1] http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing

then be routed to another process for handling (see *)http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing*[2].)

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jms="http://www.milyn.org/xsd/smooks/jms-
routing-1.2.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/
freemarker-1.1.xsd">

    <jms:router routeOnElement="order-item" beanId="orderItem_xml"
 destination="queue.orderItems" />

    <ftl:freemarker applyOnElement="order-item">
        <ftl:template>/orderitem-split.ftl</ftl:template>
        <ftl:use>
            <!-- Bind the templating result into the bean context, from
 where
                 it can be accessed by the JMSRouter (configured above). --
>
            <ftl:bindTo id="orderItem_xml"/>
        </ftl:use>
    </ftl:freemarker>

</smooks-resource-list>
```

Example 5.4. Example - Binding the Templating Result to the Smooks Bean Context

Note

See full example in the tutorial at *http://www.smooks.org/mediawiki/index.php?title=Smooks_v1.2_Examples*.

By using <ftl:outputTo>, one can direct Smooks to write the templating result directly to an **`OutputStreamResource`**. This is another useful mechanism for splitting (*)http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing*[3]) huge messages into smaller ones that are more "consumable." They can then be processed individually.

[2] http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing
[3] http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/
javabean-1.2.xsd"
                      xmlns:file="http://www.milyn.org/xsd/smooks/file-
routing-1.1.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/
freemarker-1.1.xsd">

    <!-- Create/open a file output stream. This is written to by the
 freemarker template (below).. -->
    <file:outputStream openOnElement="order-item"
 resourceName="orderItemSplitStream">
        <file:fileNamePattern>order-${order.orderId}-
${order.orderItem.itemId}.xml</file:fileNamePattern>
        <file:destinationDirectoryPattern>target/orders</
file:destinationDirectoryPattern>
        <file:listFileNamePattern>order-${order.orderId}.lst</
file:listFileNamePattern>

        <file:highWaterMark mark="3"/>
    </file:outputStream>

    <!--
    Every time we hit the end of an <order-item> element, apply this
 freemarker template,
    outputting the result to the "orderItemSplitStream" OutputStream, which
 is the file
    output stream configured above.
    -->
     <ftl:freemarker applyOnElement="order-item">
        <ftl:template>target/classes/orderitem-split.ftl</ftl:template>
        <ftl:use>
            <!-- Output the templating result to the "orderItemSplitStream"
 file output stream... -->
            <ftl:outputTo outputStreamResource="orderItemSplitStream"/>
        </ftl:use>
    </ftl:freemarker>

</smooks-resource-list>
```

Example 5.5. Example - Writing the Template Result to an **`OutputStreamSource`**

Note

See full example in the tutorial at *http://www.smooks.org/mediawiki/index.php?title=Smooks_v1.2_Examples*.

5.2. FreeMarker Transforms using NodeModels

The easiest way in which to construct message transformations in **FreeMarker** is to use the latter's *NodeModel* functionality (see *http://freemarker.org/docs/xgui_expose_dom.html*.) This is where **FreeMarker** uses a W3C DOM for its templating model, as it references the DOM nodes directly from within the **FreeMarker** template.

Smooks adds two additional capabilities:

- The ability to perform this on a "fragment" basis, meaning that just the targeted fragment, rather than the full message, can be used as the DOM model.
- The ability to use the NodeModel (*http://freemarker.org/docs/xgui_expose_dom.html*) in a streaming filter process (*http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Mixing_DOM_and_SAX*.)
- The ability to use it on non-XML messages (CSV, EDI and so forth.)

To use this facility in Smooks, one needs to define an additional resource that declares the NodeModels to be captured (or created, in the case of SAX streaming):

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/
javabean-1.2.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/
freemarker-1.1.xsd">

    <!--
    Create 2 NodeModels. One high level model for the "order"
    (header etc) and then one per "order-item".

    These models are used in the FreeMarker templating resources
    defined below. You need to make sure you set the selector such
    that the total memory footprint is as low as possible. In this
    example, the "order" model will contain everything accept the
    <order-item> data (the main bulk of data in the message). The
    "order-item" model only contains the current <order-item> data
    (i.e. there's max 1 order-item in memory at any one time).
    -->
     <resource-config selector="order,order-item">
        <resource>org.milyn.delivery.DomModelCreator</resource>
    </resource-config>

    <!--
    Apply the first part of the template when we reach the start
    of the <order-items> element. Apply the second part when we
    reach the end.

    Note the <?TEMPLATE-SPLIT-PI?> Processing Instruction in the
    template. This tells Smooks where to split the template,
    resulting in the order-items being inserted at this point.
    -->
```

```
    <ftl:freemarker applyOnElement="order-items">
        <ftl:template><!--<salesorder>
    <details>
        <orderid>${order.@id}</orderid>
        <customer>
            <id>${order.header.customer.@number}</id>
            <name>${order.header.customer}</name>
        </customer>
    <details>
    <itemList>
    <?TEMPLATE-SPLIT-PI?>
    </itemList>
</salesorder>--></ftl:template>
    </ftl:freemarker>

    <!--
    Output the <order-items> elements. This will appear in the
    output message where the <?TEMPLATE-SPLIT-PI?> token appears in the
    order-items template.
    -->
     <ftl:freemarker applyOnElement="order-item">
        <ftl:template><!-- <item>
    <id>${.vars["order-item"].@id}</id>
    <productId>${.vars["order-item"].product}</productId>
    <quantity>${.vars["order-item"].quantity}</quantity>
    <price>${.vars["order-item"].price}</price>
<item>--></ftl:template>
    </ftl:freemarker>

</smooks-resource-list>
```

Note

See full example in this tutorial: *http://www.smooks.org/mediawiki/index.php?title=V1.2:xml-to-xml*.

5.2.1. FreeMarker and the Javabean Cartridge

The **FreeMarker** NodeModel is very powerful and easy to use. The trade-off is obviously that of performance. It is not cheap to construct W3C DOMs. It also may be the case that the required data has already been extracted and populated into a Java Object model anyway, an example being where the data also needs to be routed to a JMS end-point as a set of Java Objects.

For situations in which using the NodeModel is impractical, Smooks allows one to use the Javabean Cartridge to populate a proper Java Object (or Virtual) Model. This model can then be used in the **FreeMarker** templating process. See the document on the Javabean Cartridge (*)http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Java_Binding*[4]) for more details.

[4] http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Java_Binding

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/
javabean-1.2.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/
freemarker-1.1.xsd">

    <!-- Extract and decode data from the message. Used in the freemarker
 template (below). -->
    <jb:bean beanId="order" class="java.util.Hashtable"
 createOnElement="order">
        <jb:value property="orderId" decoder="Integer" data="order/@id"/>
        <jb:value property="customerNumber" decoder="Long" data="header/
customer/@number"/>
        <jb:value property="customerName" data="header/customer"/>
        <jb:wiring property="orderItem" beanIdRef="orderItem"/>
    </jb:bean>
    <jb:bean beanId="orderItem" class="java.util.Hashtable"
 createOnElement="order-item">
        <jb:value property="itemId" decoder="Integer" data="order-item/
@id"/>
        <jb:value property="productId" decoder="Long" data="order-item/
product"/>
        <jb:value property="quantity" decoder="Integer" data="order-item/
quantity"/>
        <jb:value property="price" decoder="Double" data="order-item/
price"/>
    </jb:bean>

    <ftl:freemarker applyOnElement="order-item">
        <ftl:template><!--<orderitem id="${order.orderItem.itemId}"
 order="${order.orderId}">
    <customer>
        <name>${order.customerName}</name>
        <number>${order.customerNumber?c}</number>
    </customer>
    <details>
        <productId>${order.orderItem.productId}</productId>
        <quantity>${order.orderItem.quantity}</quantity>
        <price>${order.orderItem.price}</price>
    </details>
</orderitem>-->
        </ftl:template>
    </ftl:freemarker>

</smooks-resource-list>
```

Example 5.6. Example (using a Virtual Model)

Note

The full example can be seen at *http://www.smooks.org/mediawiki/index.php?title=Smooks_v1.2_Examples*.

5.2.2. Programmatic Configuration

FreeMarker templating configurations can be added to a Smooks instance programmatically by simply configuring and adding a **`FreeMarkerTemplateProcessor`** (*http://www.milyn.org/javadoc/v1.2/smooks-cartridges/templating/org/milyn/templating/freemarker/FreeMarkerTemplateProcessor.html*) instance to it. The following example configures a Smooks instance with configurations for a Java Binding and **FreeMarker** templating:

```
Smooks smooks = new Smooks();

smooks.addVisitor(new Bean(OrderItem.class, "orderItem", "order-
item").bindTo("productId", "order-item/product/@id"));
smooks.addVisitor(new FreeMarkerTemplateProcessor(new
 TemplatingConfiguration("/templates/order-tem.ftl")), "order-item");

// And then just use Smooks as normal... filter a Source to a Result etc...
```

5.2.3. XSL Templating

The process of configuring XSL templates in Smooks is almost identical to that of configuring **`FreeMarkerTemplateProcessor`** (*http://www.milyn.org/javadoc/v1.2/smooks-cartridges/templating/org/milyn/templating/freemarker/FreeMarkerTemplateProcessor.html*). It is achieved via the Just configure this XSD in an IDE (*http://www.milyn.org/xsd/smooks/xsl-1.1.xsd*) in order to begin to use it immediately.

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:xsl="http://www.milyn.org/xsd/smooks/xsl-1.1.xsd">

    <xsl:xsl applyOnElement="#document">
        <xsl:template><!--<xxxxxx/>--></xsl:template>
    </xsl:xsl>

</smooks-resource-list>
```

Example 5.7. Example

As is the case with **FreeMarker**, external templates can be configured via a URI reference in the <xsl:template> element.

As already stated, the process of configuring XSL templates in Smooks is almost identical to that of configuring **FreeMarker** templates (see above.) For this reason, please consult the **FreeMarker**

configuration documents. Translating them to XSL equivalents is simply a matter of changing the configuration name-space. However, please do read the following sections.

5.2.3.1. Points to Note Regarding XSL Support

It does not make sense to use Smooks for the purpose of executing XSLT, unless:

- There is a need to perform a fragment transformation, as opposed to the transformation of a whole message.
- There is a need to use other Smooks functionality to perform additional operations on the message, such as those for splitting or persistence.
- XSL templating is only supported through the DOM filter. It is not supported through the SAX filter. This can (depending on the XSL being applied) result in lower performance when compared to a SAX-based application of XSL.
- Smooks applies XSLs on a per-message fragment basis (DOM Element Nodes) as opposed to the whole document (DOM Document Node.) This can be very useful for fragmenting or modularizing XSLs, but do not assume that an XSL written and working for a stand-alone context (that is, external to Smooks and on the whole document) will automatically work through Smooks without modification. For this reason, Smooks does handle XSLs targeted at the document root node differently in that it applies the XSL to the DOM Document Node (rather than the root DOM Element). The basic point to note here is that if there already are XSLs being ported to Smooks, some "tweaks" to the stylesheet may be required.
- XSLs typically contain a template that is matched with the root element. Because Smooks applies XSLs on a per-fragment basis, matching against the "root element" is no longer valid. Ensure that the stylesheet contains a template that matches against the context node (that is, the targeted fragment.)

5.2.3.2. Potential Issue: XSLT Works Outside Smooks but Not Within

This can happen on occasions and is most likely the result of one of the following:

- The *Fragment-Based Processing Model*: when the stylesheet contains a template that is using an absolute path reference to the document root node, issues will occur in the Smooks Fragment-Based Processing Model. This is because the element being targeted by Smooks is not the document root node. One's XSLT needs to contain a template that matches against the context node being targeted by Smooks.
- *SAX versus DOM Processing*: "like" is not being compared with "like." Smooks currently only supports a DOM-based processing model for XSL. In order to undertake an accurate comparison, one needs to use a *DOMSource* (that is namespace-aware) when executing the XSLT outside Smooks. It has been noticed that a given XSL Processor does not always produce the same output when applying a given XSLT using SAX or DOM.

"Groovy" Scripting

Support for *Groovy* scripting (*http://groovy.codehaus.org*) is made available through the *configuration namespace* (*http://www.milyn.org/xsd/smooks/groovy-1.1.xsd*.) The namespace adds support for DOM- or SAX-based scripting.

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:g="http://www.milyn.org/xsd/smooks/groovy-1.1.xsd">

    <g:groovy executeOnElement="xxx">
        <g:script>
        <!--
        //Rename the target fragment element from "xxx" to "yyy"...
        DomUtils.renameElement(element, "yyy", true, true);
        -->
        </g:script>
    </g:groovy>

</smooks-resource-list>
```

Example 6.1. Example Configuration

Here are some tips on usage.

- *Imports*: These can be added via the appropriately-named "imports" element. A number of classes are automatically imported:
- • **`org.milyn.xml.DomUtils`**
  - **`org.milyn.javabean.repository.BeanRepository`**
  - **`org.w3c.dom.*`**
  - **`groovy.xml.dom.DOMCategory`**
  - **`groovy.xml.dom.DOMUtil`**
  - **`groovy.xml.DOMBuilder`**
- *Visited Element*: the visited element is available to the script through the appropriately named variable called "element." It is also available under that variable name which is equal to the element name but only if the latter name contains only alpha-numeric characters.
- *Execute Before*/*Execute After*: by default, the script is executed on the visitAfter event. One can direct it to be executed on the visitBefore by setting the executeBefore attribute to `true`.
- *Comment*/*CDATA Script Wrapping*: if the script contains special XML characters, it can be wrapped in an `XML Comment` or `CDATA` section.

6.1. Mixed DOM and SAX with Groovy

Because Groovy has a number of very useful DOM processing features, support has been added for the mixed DOM and SAX models.

This means that one can use Groovy's DOM utilities to process the message fragment that is being targeted. The "element" received by the Groovy script will be a DOM one, even when the SAX filter is being used. This makes Groovy scripting via the latter filter much easier, whilst, at the same time, maintaining the ability to process huge messages in a streamed fashion.

Things of Which to be Wary with Mixed DOM and SAX:

- It is only available in default mode (that is, when executeBefore equals `false`. If executeBefore is configured `true`, then this facility will not be available and the Groovy script will only have access to the element as a SAXElement.
- In order to write the DOM fragment to a **`Smooks.filterSource StreamResult`**, writeFragment must be called. (See the example below.)
- There is obviously performance overhead to be incurred by using this DOM construction facility. Nevertheless, it can still be used to process huge messages because of the way in which the **`DomModelCreator`** works in conjunction with SAX. Thus, it can still process huge messages but it might take a slightly longer period of time. The trade-off is that between "usability" and performance.

6.1.1. Mixed DOM and SAX Example

Take an XML message such as the following:

```
<shopping>
    <category type="groceries">
        <item>Chocolate</item>
        <item>Coffee</item>
    </category>
    <category type="supplies">
        <item>Paper</item>
        <item quantity="4">Pens</item>
    </category>
        <category type="present">
        <item when="Aug 10">Kathryn's Birthday</item>
    </category>
</shopping>
```

One will want to modify the "supplies" category in the shopping list, by adding two to the quantity of "Pens." To do this, one must write a simple Groovy script and target it at the <category> elements in the message.

The script simply iterates over the <item> elements in the category and, where the category type is "supplies" and the item is "Pens", it increments the quantity by two:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:g="http://www.milyn.org/xsd/smooks/groovy-1.1.xsd">
```

```
    <params>
        <param name="stream.filter.type">SAX</param>
    </params>

    <g:groovy executeOnElement="category">
        <g:script>
        <!--
        use(DOMCategory) {
            // Modify "supplies": we need an extra 2 pens...
            if (category.'@type' == 'supplies') {
                category.item.each { item ->
                    if (item.text() == 'Pens') {
                        item['@quantity'] = item.'@quantity'.toInteger() +
 2;
                    }
                }
            }
        }

        // When using the SAX filter, we need to explicitly write the
 fragment
        // to the result stream...
        writeFragment(category);
        -->
        </g:script>
    </g:groovy>

</smooks-resource-list>
```

Processing Non-XML Data

Smooks relies on a *stream reader* to generate a flow of SAX events. These emanate from the source message's data stream. A stream reader is a class that implements either the `XMLReader` interface (*http://java.sun.com/j2se/1.5.0/docs/api/org/xml/sax/XMLReader.html*) or the `SmooksXMLReader` interface (*http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/xml/SmooksXMLReader.html*.)

Smooks uses the default XMLReader (*http://java.sun.com/j2se/1.5.0/docs/api/org/xml/sax/helpers/XMLReaderFactory.html#createXMLReader%28%29*), but it can read non-XML data sources if a specialized XMLReader is configured:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">

    <reader class="com.acme.ZZZZReader" />

    <!--
    Other Smooks resources, e.g. <jb:bean> configs for
    binding data from the ZZZZ data stream into Java Objects....
    -->

</smooks-resource-list>
```

The reader can also be configured with a set of handlers, features and parameters. Here is a full example of a configuration:

```
<reader class="com.acme.ZZZZReader">
    <handlers>
        <handler class="com.X" />
        <handler class="com.Y" />
    </handlers>
    <features>
        <setOn feature="http://a" />
        <setOn feature="http://b" />
        <setOff feature="http://c" />
        <setOff feature="http://d" />
    </features>
    <params>
        <param name="param1">val1</param>
        <param name="param2">val2</param>
    </params>
</reader>
```

A number of non-XML Readers are available with Smooks "out-of-the-box:"

- *http://www.milyn.org/javadoc/v1.2/smooks-cartridges/csv/org/milyn/csv/CSVReader.html*
- *http://www.milyn.org/javadoc/v1.2/smooks-cartridges/edi/org/milyn/smooks/edi/EDIReader.html*
- *http://www.milyn.org/javadoc/v1.2/smooks-cartridges/edi/org/milyn/json/JSONReader.html*[1]

- *http://www.milyn.org/javadoc/v1.2/smooks-cartridges/edi/org/milyn/delivery/java/XStreamXMLReader.html*

Any of these XML readers can be configured as outlined above but bear in mind that some of them have specialized configuration namespaces that simplify configuration.

Each of the readers will also convert certain special characters in a data field to the equivalent *XML Entity References*. **.** will be converted to >, and will be converted to &. If the data field is being converted to an XML attribute, then two additional characters will be converted to equivalent XML Entity References: **'** will be converted to ' and **"** will be converted to ".

7.1. Processing CSV

CSV processing through the designated reader is configured through the *http://www.milyn.org/xsd/smooks/csv-1.2.xsd* configuration namespace.

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <!--
    Configure the CSV to parse the message into a stream of SAX events.
    -->
    <csv:reader fields="firstname,lastname,gender,age,country"
 separator="|" quote="'" skipLines="1" />

</smooks-resource-list>
```

Example 7.1. A Simple Configuration

The above configuration will generate an event stream that takes the following form:

```
<csv-set>
    <csv-record>
        <firstname>Tom</firstname>
        <lastname>Fennelly</lastname>
        <gender>Male</gender>
        <age>21</age>
        <country>Ireland</country>
    </csv-record>
    <csv-record>
        <firstname>Tom</firstname>
        <lastname>Fennelly</lastname>
        <gender>Male</gender>
        <age>21</age>
        <country>Ireland</country>
    </csv-record>
</csv-set>
```

The <csv-set> and <csv-record> element names can be modified by setting the rootElementName and recordElementName attributes.

7.1.1. Ignoring Fields

One or more fields of a CSV record can be ignored by specifying the `$ignore$` token as the field's configuration value. One can specify the number of fields to be ignored simply by following the `$ignore$` token with a number (for example, `$ignore$3` to ignore the next three fields.) `$ignore$+` ignores all of the fields up to the end of the CSV record.

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="firstname,$ignore$2,age,$ignore$+" />

</smooks-resource-list>
```

7.1.2. Binding CSV Records to Java

Smooks v1.2 has added support that makes the binding of CSV records to Java Objects a very trivial task. There is no longer a need to use the Javabean Cartridge directly (that is, the Smooks main Java binding functionality.)

A Person's CSV record set such as this:

```
Tom,Fennelly,Male,4,Ireland
Mike,Fennelly,Male,2,Ireland
```

can be bound to a person (no "getters" or "setters"):

```
public class Person {
    private String firstname;
    private String lastname;
    private String country;
    private Gender gender;
    private int age;
}

public enum Gender {
    Male,
    Female;
}
```

using a configuration of the form:

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

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="firstname,lastname,gender,age,country">
        <!-- Note how the field names match the property names on the
 Person class. -->
        <csv:listBinding beanId="people" class="org.milyn.csv.Person" />
    </csv:reader>

</smooks-resource-list>
```

In order to execute this configuration, use:

```
Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(csvStream), result);

List<Person> people = (List<Person>) result.getBean("people");
```

Smooks also supports the creation of Maps from the CSV record-set, as per:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="firstname,lastname,gender,age,country">
        <csv:mapBinding beanId="people" class="org.milyn.csv.Person"
 keyField="firstname" />
    </csv:reader>

</smooks-resource-list>
```

The configuration above produces a Map of "Person" instances, keyed by the firstname value for each Person. It can be executed in the following way:

```
Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(csvStream), result);

Map<String, Person> people = (Map<String, Person>)
 result.getBean("people");

Person tom = people.get("Tom");
Person mike = people.get("Mike");
```

Virtual Models (*http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Virtual_Object_Models_.28Maps_.26_Lists.29*) are also supported, so one can defined the class attribute as a **`java.util.Map`** and bind CSV field values into Map instances, which are in turn added to a List or a Map.

7.1.3. Programmatic Configuration

Configuring the CSV Reader on a Smooks instance programmatically is trivial as no XML is required. A number of options are available:

7.1.3.1. Configuring Directly on the Smooks Instance

The following code configures a Smooks instance with a CSV Reader for that will be used to analyse a "people" record-set (see above), binding the latter into a List of Person instances:

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new
 CSVReaderConfigurator("firstname,lastname,gender,age,country")
                  .setBinding(new CSVBinding("people", Person.class,
 CSVBindingType.LIST)));

JavaResult result = new JavaResult();
smooks.filterSource(new StreamSource(csvReader), result);

List<Person> people = (List<Person>) result.getBean("people");
```

Of course, configuring the Java Binding is completely optional. The Smooks instance could instead (or in conjunction with it) be programmatically configured with other Visitor implementations for the purpose of carrying out other forms of processing on the CSV record-set.

7.1.3.2. CSV List and Map Binders

If you're just interested in binding CSV Records directly onto a List or Map of a Java type that reflects the data in your CSV records, then you can use the **`CSVListBinder`** or **`CSVMapBinder`** classes.

```
// Note: The binder instance should be cached and reused...
CSVListBinder binder = new
 CSVListBinder("firstname,lastname,gender,age,country", Person.class);

List<Person> people = binder.bind(csvStream);
```

Example 7.2. CSVListBinder

```
// Note: The binder instance should be cached and reused...
CSVMapBinder binder = new
 CSVMapBinder("firstname,lastname,gender,age,country", Person.class,
 "firstname");

Map<String, Person> people = binder.bind(csvStream);
```

Example 7.3. CSVMapBinder

If more control over the binding process is needed, then revert back to the lower-level APIs:

- Configuring Directly on the Smooks Instance
- Java Binding

7.1.4. Processing EDI

EDI processing in Smooks supported through the *http://www.milyn.org/xsd/smooks/edi-1.2.xsd* configuration namespace.

The following is a basic configuration:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:edi="http://www.milyn.org/xsd/smooks/edi-1.2.xsd">
 <!--
    Configure the EDI Reader to parse the message stream into a stream of
 SAX events.
    -->
    <edi:reader mappingModel="edi-to-xml-order-mapping.xml"
 validate="false"/>
</smooks-resource-list>
```

- mappingModel: defines the *EDI Mapping Model* configuration which is used to process the EDI message stream, which goes to a stream of SAX events. These events can then that be processed by Smooks.
- validate: this attribute turns data type validation in the *EDI Parser* on and off. Validation is "on" by default. It makes sense to turn it off on the EDI Reader if the EDI data is being bound into a Java Object model (using Java Bindings a la <jb:bean>.) This is because the validation will be happening at the binding level anyway .

7.1.4.1. EDI Mapping Models

The EDI-to-SAX event mapping is performed on the basis of a "Mapping Model" supplied to the EDI Reader. This model must be based on the schema found at: *http://www.milyn.org/xsd/smooks/edi-1.2.xsd*. From this schema, it can be seen that *segment groups* (nested segments) are supported, including "groups within groups," "repeating segments" and "repeating-segment groups." Be sure to review the schema.

The following illustration attempts to create a visualisation of the mapping process. The **`input-message.edi`** file specifies the EDI input, **`edi-to-xml-order-mapping.xml`** describes how to map that EDI message to SAX events and **`expected.xml`** illustrates the XML event stream that would result from the application of the mapping.

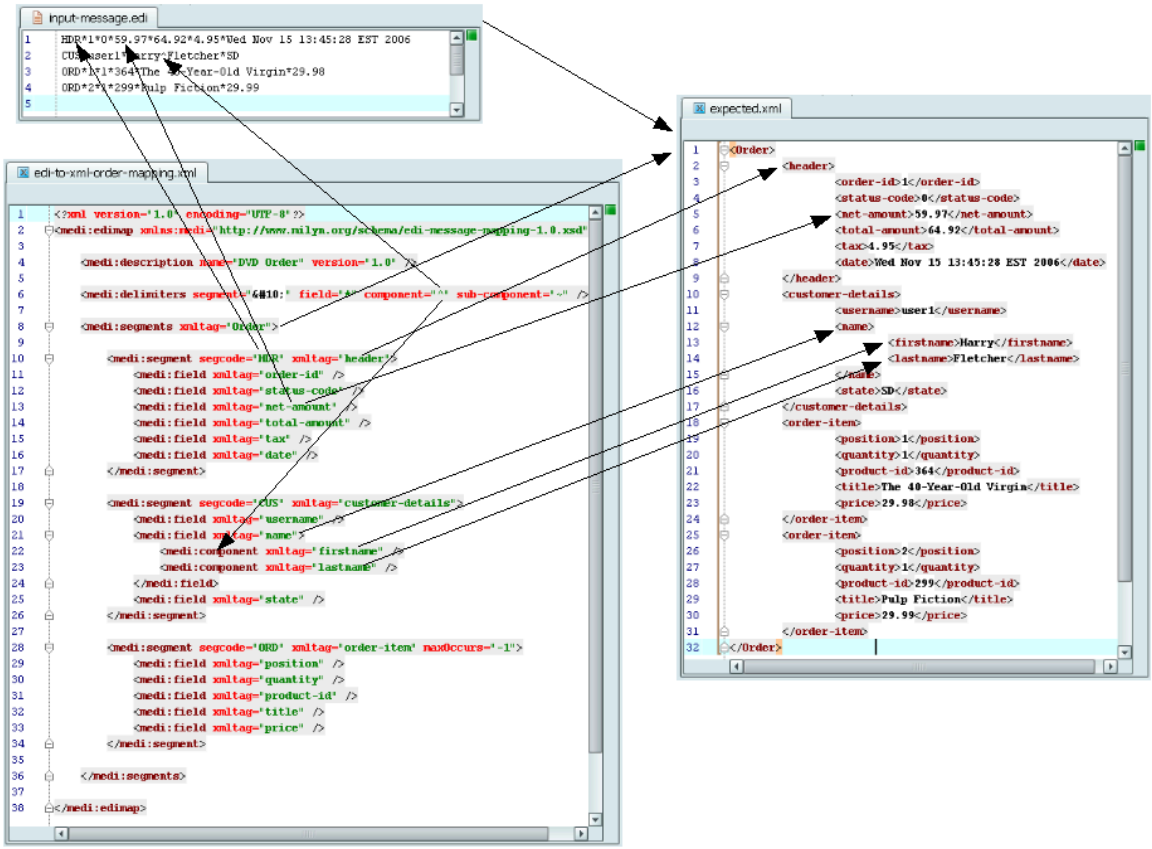

Figure 7.1. The Mapping Process

The illustration above highlights the following:

- How the message delimiters (segment, field, component and sub-component) are specified in the mapping. In particular, how special characters like that for the linefeed are specified by using *XML Character References*.
- How segment groups (that is, nested segments) are specified. In this case the first two segments are part of a group.
- How the actual field, component and sub-component values are specified and mapped to the target SAX events (in order to generate the XML.)

7.1.4.1.1. Segment Cardinality

Not shown above is how the element supports the two optional attributes, minOccurs and maxOccurs (there is a default value of "one" in each cases.) These attributes can be used to control both the optional and the required characteristics of a segment. A maxOccurs value of `-1` indicates that the segment can repeat any number of times in that location of the (unbounded) EDI message.

7.1.4.1.2. Segment Groups

Segment groups can be added using the <segmentGroup> element. A segment group is matched by the first segment in the group. A segment group can contain nested <segmentGroup> elements but its first element must be a . <segmentGroup> elements support minOccurs and maxOccurs cardinality. They also support an optional xmlTag attribute, when if present will result in the XML generated by a matched segment group that is to be inserted inside an element to have the name of the xmlTag attribute value.

7.1.4.1.3. Segment Matching

Segments are matched in one of two ways:

- by an exact match on the segment code (*segcode*.)
- by a *regex pattern match* (see *http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/package-summary.html* on the full segment, where the segcode attribute defines the regex pattern (for example, `segcode="1A\*a.*"`.)

7.1.4.1.4. Required Values and Truncation

- required: <field>, <component> and <sub-component> configurations support a "required" attribute, which flags that <field>, <component> and <sub-component> as requiring a value. By default, values are not required (fields, components and sub-components.)
- truncatable , <field> and <component> configurations support a "truncatable" attribute. For a segment, this means that parser errors will not be generated when it does not specify trailing fields that are not "required" (see the "required" attribute above.) This is likewise the case for fields or components and components or sub-components. By default, segments, fields, and components are not truncatable.

So, a <field>, a <component> and a <sub-component> can each be present in a message in one of the following states:

- XML-to-XML
- Present with a value `''(required="true")''`
- Present without a value `''(required="false")''`
- Not Present `''(required="false" and truncatable="true")''`

7.1.4.2. Imports

Many message groups use the same segment definitions. Being able to define these segments once and import them into a top-level configuration saves on a great deal of duplication. A simple configuration demonstrating the "import" feature is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<medi:edimap xmlns:medi="http://www.milyn.org/schema/edi-message-
mapping-1.2.xsd">

    <medi:import truncatableSegments="true" truncatableFields="true"
 truncatableComponents="true" resource="example/edi-segment-definition.xml"
 namespace="def"/>
```

```
    <medi:description name="DVD Order" version="1.0"/>

    <medi:delimiters segment="&#10;" field="*" component="^" sub-
component="~" escape="?"/>

    <medi:segments xmltag="Order">
        <medi:segment minOccurs="0" maxOccurs="1" segref="def:HDR"
 segcode="HDR" xmltag="header"/>
        <medi:segment minOccurs="0" maxOccurs="1" segref="def:CUS"
 segcode="CUS" xmltag="customer-details"/>
        <medi:segment minOccurs="0" maxOccurs="-1" segref="def:ORD"
 segcode="ORD" xmltag="order-item"/>
    </medi:segments>

</medi:edimap>
```

The configuration example above demonstrates the use of "import," (which was were introduced in **Smooks v1.1**), where single segments or segments containing child segments can be separated into another file for re-use in the future.

- segref: contains a `namespace:name` referencing the segment to import.
- truncatableSegments: this overrides the truncatableSegments that was specified in the imported resource mapping file.
- truncatableFields: this overrides the truncatableFields that was specified in the imported resource mapping file.
- truncatableComponents: this overrides the truncatableComponents that was specified in the imported resource mapping file.

7.1.4.3. Type Support

Since Version 1.2, the <field>, <component> and <sub-component> elements all support a type attribute that allows a data-type specification. It actually consists of two attributes:

- type: this attribute specifies the basic data-type.
- typeParameters: the typeParameters attribute specifies data decoding parameters for the **`DataDecoder`** (*http://www.milyn.org/javadoc/v1.2/commons/org/milyn/javabean/decoders/package-summary.html*) that is associated with the specified type.

The following example shows the type support:

```
<?xml version="1.0" encoding="UTF-8"?>
<medi:edimap xmlns:medi="http://www.milyn.org/schema/edi-message-
mapping-1.2.xsd">

    <medi:description name="Segment Definition DVD Order" version="1.0"/>
```

```
    <medi:delimiters segment="&#10;" field="*" component="^" sub-
component="~" escape="?"/>

    <medi:segments xmltag="Order">

        <medi:segment segcode="HDR" xmltag="header">
            <medi:field xmltag="order-id"/>
            <medi:field xmltag="status-code" type="Integer"/>
            <medi:field xmltag="net-amount" type="BigDecimal"/>
            <medi:field xmltag="total-amount" type="BigDecimal"/>
            <medi:field xmltag="tax" type="BigDecimal"/>
            <medi:field xmltag="date" type="Date"
 typeParameters="format=yyyyHHmm"/>
        </medi:segment>

    </medi:segments>

</medi:edimap>
```

This type system has a number of uses:

- Field Validation.
- *Edifact Java Compiler*, known by the abbreviation EJC. (See *)http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#EJC_-_Edifact_Java_Compiler*[2].)

7.1.4.4. Programmatic Configuration

Programmatically configuring the Smooks instance to use the EDI Reader is achieved via the **`EDIReaderConfigurator`** (see *http://www.milyn.org/javadoc/v1.2/smooks-cartridges/edi/org/milyn/smooks/edi/EDIReaderConfigurator.html*.)

```
Smooks smooks = new Smooks();

// Create and initialise the Smooks config for the parser...
smooks.setReaderConfig(new EDIReaderConfigurator("/edi/models/
invoice.xml"));

// Use the smooks as normal
smooks.filterSource(....);
```

7.1.4.5. EJC (Edifact Java Compiler)

EJC makes the process of going from EDI to Java much more simple. EJC is similar to XJC JAXBs (see *http://jaxb.dev.java.net/*), except it is used for EDI messages.

EJC generates:

- a Java Object model for a given EDI Mapping Model.

- a Smooks Java Binding configuration, used to populate the Java Object model from an instance of the EDI message that has been described by the EDI Mapping Model (see the first point above.)
- a Factory class that makes it very easy to use EJC to bind EDI data to Java Object models.

EJC facilitates the writing of simple Java code. Here is an example:

```
// Create an instance of the EJC generated Factory class.  This should
 normally be cached and reused...
OrderFactory orderFactory = OrderFactory.getInstance();

// Bind the EDI message stream data into the EJC generated Order model...
Order order = orderFactory.fromEDI(ediStream);

// Process the order data...
Header header = order.getHeader();
Name name = header.getCustomerDetails().getName();
List<OrderItem> orderItems = order.getOrderItems();
```

EJC can be executed through either **Maven** or **Ant**.

7.1.4.6. EJC Maven Plug-In

Executing the **Maven** Plug-In for EJC is very simple. Just install the plug-in on the POM file as follows:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.milyn</groupId>
            <artifactId>maven-ejc-plugin</artifactId>
            <version>1.2</version>
            <configuration>
                <ediMappingFile>edi-model.xml</ediMappingFile>
                <packageName>com.acme.order.model</packageName>
            </configuration>
            <executions>
                <execution><goals><goal>generate</goal></goals></execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

The plug-in has three configuration parameters:

- ediMappingFile: the path to the EDI Mapping Models file (*http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#EDI_Mapping_Models*), within the **Maven** project. (This is optional. The default is **`src/main/resources/edi-model.xml`**.)
- packageName: this is the Java package into which the generated Java Artifacts are to be kept (Java Object Model and **`Factory`** class.)

- destDir: the destination directory in which the generated artifacts are created and from which they are compiled. (This is optional. The default is **`target/ejc`**.)

7.1.4.7. EJC Ant Task

Executing EJC from an **Ant** script is trivially easy. Just configure the EJC **Ant** task and execute it:

```
<target name="ejc">

    <taskdef resource="org/milyn/ejc/ant/anttasks.properties">
        <classpath><fileset dir="/smooks-1.2/lib" includes="*.jar"/></
classpath>
    </taskdef>

    <ejc edimappingmodel="src/main/resources/edi-model.xml"
         destdir="src/main/java"
         packagename="com.acme.order.model"/>

    <!-- Ant as usual from here on... compile and jar the source... -->

</target>
```

7.1.4.7.1. Using EJC

The easiest way in which to commence using EJC is to learn from the example at *http://www.smooks.org/mediawiki/index.php?title=V1.2:ejc*.

7.1.5. Processing JSON

Processing *JavaScript Object Notation* (JSON) with Smooks requires that a JSON reader be configured:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:json="http://www.milyn.org/xsd/smooks/json-1.1.xsd">

    <json:reader/>

</smooks-resource-list>
```

The following example demonstrates key replacement:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:json="http://www.milyn.org/xsd/smooks/json-1.1.xsd">

    <json:reader>
        <json:keyMap>
            <json:key from="some key">someKey</json:key>
```

```
            <json:key from="some&amp;key" to="someAndKey" />
        </json:keyMap>
    </json:reader>

</smooks-resource-list>
```

The following is a full configuration example for the JSON reader:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
 xmlns:json="http://www.milyn.org/xsd/smooks/json-1.1.xsd">

    <json:reader keyWhitspaceReplacement="_" keyPrefixOnNumeric="n"
 illegalElementNameCharReplacement="." nullValueReplacement="##NULL##" />

</smooks-resource-list>
```

- keyWhitspaceReplacement: this is the replacement character for white spaces in a JSON map key. By default, this is not defined in order that the reader does not search for white space.
- keyPrefixOnNumeric: this is the prefix character to add if the JSON node name starts with a number. By default, this is not defined. This is so that the reader does not search for element names that commence with a number.
- illegalElementNameCharReplacement: if illegal characters are encountered in a JSON element name, they are replaced with this value.
- nullValueReplacement: this is the replacement string for JSON `NULL` values. The default is an empty string.
- encoding: this is the default encoding of any JSON message InputStream processed by this Reader. The default is `UTF-8`. NOTE: One should not need to use this configuration parameter. It will be removed in a future release. Rather, one should manage the JSON stream source character encoding by supplying a java.io.Reader to the **`Smooks.filterSource()`** method.

7.1.5.1. Programmatic Configuration

Smooks is programmatically configured to read a JSON configuration by using the **`JSONReaderConfigurator`** class. (See *http://www.milyn.org/javadoc/v1.2/smooks-cartridges/json/org/milyn/json/JSONReaderConfigurator.html*.)

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new JSONReaderConfigurator()
        .setRootName("root")
        .setArrayElementName("e"));

// Use Smooks as normal...
```

7.1.6. Configuring the Default Reader

In order to set features on the default reader, simply omit the class name from the configuration:

```
<reader>
    <features>
        <setOn feature="http://a" />
        <setOn feature="http://b" />
        <setOff feature="http://c" />
        <setOff feature="http://d" />
    </features>
</reader>
```

Java-to-Java Transformations

Smooks can transform one Java object graph into another. To achieve this transformation, Smooks uses the SAX processing model. This means that no intermediate object model is constructed for populating the target Java object graph. Instead, it goes straight from the source Java object graph to a stream of SAX events. The latter are used to populate the target Java object graph.

8.1. Source and Target Object Models

The required mappings from the source to the target Object models are as follows:

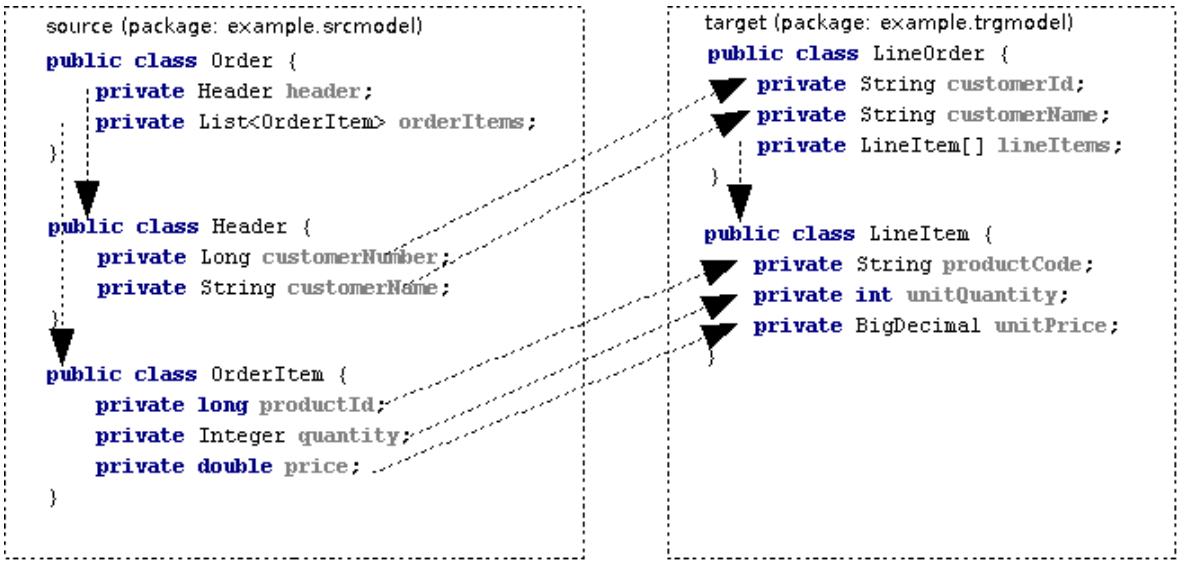

Figure 8.1. Java-to-Java mapping

8.2. Source Model Event Stream

By using the HTML Smooks Report Generator tool, we can see that the event stream produced by the source Object Model is as follows:

```
<example.srcmodel.Order>
    <header>
        <customerNumber>
        </customerNumber>
        <customerName>
        </customerName>
    </header>
    <orderItems>
        <example.srcmodel.OrderItem>
            <productId>
            </productId>
            <quantity>
            </quantity>
            <price>
            </price>
        </example.srcmodel.OrderItem>
```

```
    </orderItems>
</example.srcmodel.Order>
```

So we need to target the Smooks Javabean resources at this event stream. This is shown in the Smooks configuration.

8.3. Smooks Configuration

The Smooks configuration (**`smooks-config.xml`**) for performing this transformation is as follows. See *Section 8.2, "Source Model Event Stream"*.

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.2.xsd">

    <jb:bean beanId="lineOrder" class="example.trgmodel.LineOrder"
      createOnElement="example.srcmodel.Order">
        <jb:wiring property="lineItems" beanIdRef="lineItems" />
        <jb:value property="customerId" data="header/customerNumber" />
        <jb:value property="customerName" data="header/customerName" />
    </jb:bean>

    <jb:bean beanId="lineItems" class="example.trgmodel.LineItem[]"
      createOnElement="orderItems">
        <jb:wiring beanIdRef="lineItem" />
    </jb:bean>

    <jb:bean beanId="lineItem" class="example.trgmodel.LineItem"
      createOnElement="example.srcmodel.OrderItem">
        <jb:value property="productCode"
            data="example.srcmodel.OrderItem/productId" />
        <jb:value property="unitQuantity"
            data="example.srcmodel.OrderItem/quantity" />
        <jb:value property="unitPrice"
            data="example.srcmodel.OrderItem/price" />
    </jb:bean>

</smooks-resource-list>
```

8.4. Smooks Execution

The source object model is provided to Smooks via a **`org.milyn.delivery.JavaSource`** Object. This object is created by passing the root object of the source model to the constructor. The resulting JavaSource object is used in the `Smooks#filter` method. The resulting code could look like the following:

```
protected LineOrder runSmooksTransform(Order srcOrder)
    throws IOException, SAXException
{
    Smooks smooks = new Smooks("smooks-config.xml");
```

```
    ExecutionContext executionContext = smooks.createExecutionContext();

    // Transform the source Order to the target LineOrder via a
    // JavaSource and JavaResult instance...
    JavaSource source = new JavaSource(srcOrder);
    JavaResult result = new JavaResult();

    // Configure the execution context to generate a report...
    executionContext.setEventListener(
        new HtmlReportGenerator("target/report/report.html"));

    smooks.filterSource(executionContext, source, result);

    return (LineOrder) result.getBean("lineOrder");
}
```

Rules

In Smooks, the term "rules" refers to a general concept that is not specific to any particular cartridge. A RuleProvider can be configured and referenced from other components. As of Smooks v1.2, the only cartridge using rules functionality is the Validation Cartridge. Refer to *Chapter 10, Validation* for more details.

Let us start by looking at what "rules" are, and how they are used.

9.1. Rule Configuration

Rules are centrally defined in "ruleBase" definitions. A single Smooks configuration can reference many "ruleBase" definitions. A rulesBase configuration has a name, a rule src and a rule provider. The format of the rule source (src) is entirely dependent upon the provider implementation. The only requirement is that the individual rules be uniquely named within the context of a single source.

An example of a ruleBase configuration is as follows:

```
<smooks-resource-list  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:rules="http://www.milyn.org/xsd/smooks/rules-1.0.xsd">

<rules:ruleBases>
    <rules:ruleBase name="regexAddressing"
        src="/org/milyn/validation/address.properties"
        provider="org.milyn.rules.regex.RegexProvider" />
    <rules:ruleBase name="order"
        src="/org/milyn/validation/order/rules/order-rules.csv"
        provider="org.milyn.rules.mvel.MVELProvider"/>
</rules:ruleBases>

</smooks-resource-list>
```

9.1.1. Rulebase Configuration Options

The following are the configuration options for the <rules:ruleBase> configuration element:

name
: Is used to reference this rule from other components, such as from a validation configuration. This is required.

src
: A file or any other source that is meaningful to the RuleProvider. For example, it could be a file containing rules. This is required.

provider
: Is the actual provider implementation that you want to use. This is where the different technologies "come into play." In the above configuration, we have one RuleProvider that uses regular expressions. You can specify multiple ruleBase elements and have as many RuleProviders as you need. This is required.

9.2. RuleProvider Implementations

Rule Providers implement the `org.milyn.rules.RuleProvider` interface.

Smooks v1.2 supports two RuleProvider implementations "out-of-the-box:" **`RegexProvider`** and **`MVELProvider`**.

You can easily create custom RuleProvider implementations. Future versions of Smooks will include support for more RuleProviders. There will, for example, be a Drools RuleProvider.

9.2.1. RegexProvider

As its name suggests, **`RegexProvider`** allows you to use regular expressions. You can utilize it to define low-level rules that are specific to the data field formats contained within the message that is being filtered. In other words, it can be used to determine that a particular field is a valid email address.

The configuration of a Regex ruleBase would look like this:

```
<smooks-resource-list  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:rules="http://www.milyn.org/xsd/smooks/rules-1.0.xsd">

    <rules:ruleBases>
        <rules:ruleBase name="customer"
            src="/org/milyn/validation/order/rules/customer.properties"
            provider="org.milyn.rules.regex.RegexProvider"/>
    </rules:ruleBases>

</smooks-resource-list>
```

Regex expressions are defined in the standard **`.properties`** file format. An example of a **`customer.properties`** Regex rule definition file (from the above example) might be:

```
# Customer data rules...
customerId=[A-Z][0-9]{5}
customerName=[A-Z][a-z]*, [A-Z][a-z]
```

9.2.1.1. Useful Regular Expressions

A list of useful regular expressions is maintained on the Smooks Wiki at *http://www.smooks.org/mediawiki/index.php?title=Useful_Regular_Expressions*.

9.2.2. MVELProvider

The **`MVELProvider`** allows rules to be defined as MVEL expressions. These expressions are executed on the contents of the Smooks Javabean bean context. That means they require the data to be bound, from the message being filtered, into Java objects that are within the Smooks bean context. (Data binding is covered in *Chapter 4, Java Binding* .) This allows you to define more complex, higher-level rules on message fragments. An example of such a high-level rule might be, "Is the product contained in the targeted order item fragment to be found within the age eligibility constraints of the customer who was specified in the order header's details?"

Additional information about MVEL can be found at *http://mvel.codehaus.org/*.

Configuration of an MVEL ruleBase would look like this:

```
<smooks-resource-list  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:rules="http://www.milyn.org/xsd/smooks/rules-1.0.xsd">

    <rules:ruleBases>
        <rules:ruleBase name="order"
            src="/org/milyn/validation/order/rules/order-rules.csv"
            provider="org.milyn.rules.mvel.MVELProvider"/>
    </rules:ruleBases>

</smooks-resource-list>
```

MVEL rules must be defined as comma-separated value (CSV) files. The easiest way to edit these files is by using a spreadsheet application such as **Open Office.org** or **Microsoft Excel**. Each rule record contains two fields:

1. A Rule Name
2. An MVEL Expression

Comment/header rows can be added by prefixing the first field with a hash ('#') character.

An example of an MVEL rule CSV file as seen in OpenOffice is as follows:

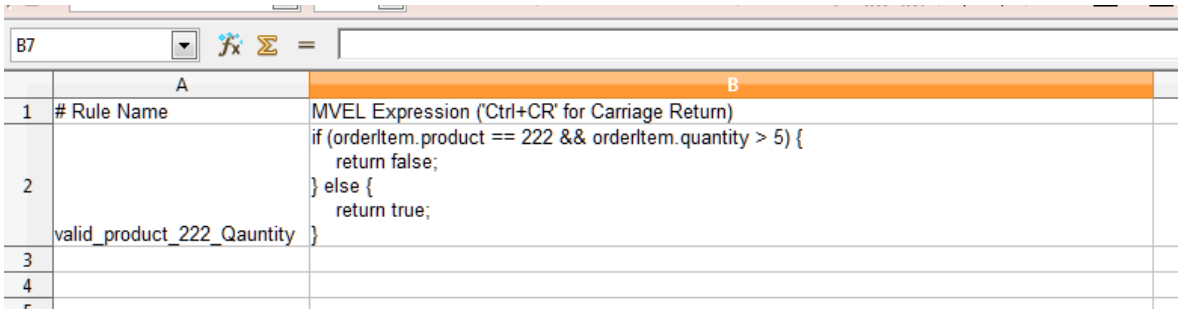

Figure 9.1. MVEL rule CSV file viewed in OpenOffice

Validation

The Smooks Validation Cartridge builds on the functionality provided by the Rules Cartridge. It provides rules-based fragment validation. See *Chapter 9, Rules*.

The validation provided by the components of the Smooks Validation Cartridge allows you to perform more detailed validation (over the likes of XSD/Relax) on message fragments. As with everything in Smooks, the validation functionality is supported across all supported data formats. This means you can perform very accurate validation on not just XML data, but also on EDI, JSON, CSV etc.

Validation configurations are defined by the *http://www.milyn.org/xsd/smooks/validation-1.0.xsd* configuration name-space.

10.1. Validation Configuration

Smooks supports a number of different types of rule provider. These can be used by the Validation Cartridge. They provide different levels of validation. These different forms of validation are all configured in exactly the same way. The Smooks Validation Cartridge sees a rule provider as an abstract resource that it can target at message fragments. This is in order to perform validation on the data contained in that message fragment.

A configuration for a validation rule is very simple. You simply need to specify:

executeOn
: The fragment on which the rule is to be executed.

excecuteOnNS
: The fragment name-space (NS) that that executeOn belongs to.

name
: The name of the rule to be applied. This is a composite rule that references a ruleBase and ruleName combination in a "dot-delimited" format (for instance, `ruleBaseName.ruleName`.)

  Refer to *Section 10.1.3, “Composite Rule Name”* for more details.

onFail
: The severity of a failure to match with the validation rule. Refer to *Section 10.1.2, “onFail”* for more details.

Here is an example of a validation rule's configuration:

```
<validation:rule executeOn="order/header/email"
    name="regexAddressing.email" onFail="ERROR" />
```

10.1.1. Configuring Maximum Failures

You can set a maximum number for the amount of validation failures per Smooks filter operation. An exception will be thrown if this value is exceeded. Note that validations configured with OnFail.FATAL will always throw an exception and stop processing.

To configure the maximum validation failures, add this following to your Smooks configuration:

```
<params>
```

```
    <param name="validation.maxFails">5</param>
</params>
```

10.1.2. onFail

The onFail attribute in the validation configuration specifies the action to be taken when a rule matches. This is for the purpose of reporting validation failures.

The following options are available:

OK
: Save the validation as an "okay" validation. Calling `ValidationResults.getOks` will return all validation warnings. This can be useful for content-based routing.

WARN
: Save the validation as a warning. Calling `ValidationResults.getWarnings` will return all validation warnings.

ERROR
: Save the validation as an error. Calling `ValidationResults.getErrors` will return all validation errors.

FATAL
: Will throw a validation exception, `ValidationException`, as soon as a validation failure occurs. Calling `ValidationResults.getFatal` will return the fatal validation failure.

10.1.3. Composite Rule Name

When a RuleBase is referenced in Smooks you use a composite rule name in the following format: <ruleProviderName>.<ruleName>.

ruleProviderName
: Identifies the rule provider and maps to the name attribute in the ruleBase element.

ruleName
: Identifies a specific rule the rule provider knows about. This could be a rule defined in the src resource.

10.2. Validation Results

Validation results are captured by `Smooks.filterSource` by specifying a **`ValidationResult`** instance in the `filterSource` method call. When the `filterSource` method returns, the **`ValidationResult`** instance will contain all validation data.

Here is an example of executing Smooks to perform message fragment validation:

```
ValidationResult validationResult = new ValidationResult();

smooks.filterSource(new StreamSource(messageInStream),
 new StreamResult(messageOutStream), validationResult);

List<OnFailResult> errors = validationResult.getErrors();
```

```
List<OnFailResult> warnings = validationResult.getWarnings();
```

From this example you can see, individual warning, error and validation results are made available from the **`ValidationResult`** object in the form of **`OnFailResult`** instances. The **`OnFailResult`** object provides details about an individual failure.

10.3. Localized Validation Messages

The Validation Cartridge provides support for specifying localized messages for validation failures. These messages can be defined in standard Java **`ResourceBundle`** files, the **`.properties`** format. A convention is used here, based on the rule source name (src). The validation message bundle base name is derived from the rule source (src) by dropping the rule source file extension and adding an extra folder named **`i18n`** e.g. for an MVEL ruleBase source of **`/org/milyn/validation/order/rules/order-rules.csv`**, the corresponding validation message bundle base name would be **`/org/milyn/validation/order/rules/i18n/order-rules`**.

The validation cartridge supports application of FreeMarker templates on the localized messages. This allows the messages to contain contextual data from the bean context, as well as data about the actual rule failure. FreeMarker-based messages must be prefixed with `ftl:` and the contextual data is referenced using the normal FreeMarker notation. The beans from the bean context can be referenced directly, while the **`RuleEvalResult`** and rule failure path can be referenced through the "ruleResult" and "path" beans.

Here is an example message that uses **`RegexProvider`** rules:

```
customerId=ftl:Invalid customer number '${ruleResult.text}' at '${path}'.
Customer number must match pattern '${ruleResult.pattern}'.
```

10.4. Example

A detailed validation example can be see at *http://www.smooks.org/mediawiki/index.php?title=V1.2:validation-basic*

Processing Huge Messages (GBs)

One of the main new features introduced in Smooks v1.0 is the ability to process huge messages (Gbs in size). Smooks supports the following forms of processing huge messages:

One-to-One Transformation
: This is the process of transforming a huge message from its source format (such as XML), to a target format such as EDI or CSV.

Splitting and Routing
: Splitting of a huge message into smaller, more consumable ones and routing of those smaller messages to a number of different destination types (such as a file, JMS or a Database).

Persistence
: Persisting the components of the huge message to a database, from which they can be more easily queried and processed. Within Smooks, we consider this to be a form of "Splitting and Routing" (as it is, in effect, routing to a Database).

All of the above actions can be achieved without the need to write any code; in other words, they can be undertaken in a declarative manner. Typically, any of the types of processing discussed above would have required the developer to write a substantial amount of code and that code would be "ugly" and difficult to maintain. It might also have been implemented as a multi-stage process whereby the huge message is split into smaller messages (stage #1) and then each smaller message is processed in turn to have it persist, route or so on. (stage #2). This would be undertaken in an effort to make that code a little more maintainable and reusable. With Smooks, most of these use-cases can be handled without the need to write any code. As well as that, they can also be handled in a single pass over the source message, splitting and routing in parallel (plus routing to multiple destinations of different types and in different formats).

Note

Be sure to read *Chapter 4, Java Binding* .

For performance reasons, you should use the SAX filter when processing huge messages with Smooks.

11.1. One-to-One Transformation

If the requirement is to process a huge message by transforming it into a single message of another format, the easiest mechanism with Smooks is to apply multiple FreeMarker templates to the Source message Event Stream, outputting to a **`Smooks.filterSource`** Result stream.

This can be done in one of two ways with FreeMarker templates, depending on the type of model that is appropriate:

1. Using FreeMarker and NodeModels for the model.
2. Using FreeMarker and a Java Object model for the model. The model can be constructed from data in the message, using the Javabean Cartridge.

Option #1 above is obviously the one of choice, if the trade-offs are permissible in your use case. Please see the FreeMarker Templating documentation for more details.

The following images shows an <order> message, as well as the <salesorder> message to which we need to transform the <order> message:

Figure 11.1. Huge Messages

Imagine a situation where the <order> message contains millions of <order-item> elements. Processing a huge message in this way with Smooks and FreeMarker (using NodeModels) is quite straightforward. Because the message is huge, we need to identify multiple NodeModels in the message, so that the runtime memory footprint is as low as possible. We cannot process the message using a single model, as the full message is just too big to hold in memory. In the case of the <order> message, there are 2 models, one for the main <order> data (blue highlight) and one for the <order-item> data (beige highlight):

```
<order id='332'>
    <header>
        <customer number="123">Joe</customer>
    </header>
    <order-items>
        <order-item id='1'>
            <product>1</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
        <order-item id='2'>
            <product>2</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
        <order-item id='3'>
            <product>3</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
        <order-item id='4'>
            <product>4</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>

        <!-- And more <order-item> elements... -->

    </order-items>
</order>
```

Figure 11.2. Huge Message Models

So in this case, the most information that will be in memory at any one time is the main order data, plus one of the order-items. Because the NodeModels are nested, Smooks makes sure that the order data NodeModel never contains any of the data from the order-item NodeModels. Also, because Smooks filters the message, the order-item NodeModel will be overwritten for every order-item (i.e. they are not collected). See Mixing DOM and SAX Models with Smooks.

Configuring Smooks to capture multiple NodeModels for use by the FreeMarker templates is just a matter of configuring the DomModelCreator, targeting it at the root node of each of the models. Note again that Smooks also makes this available to SAX filtering (which is the key to processing huge messages.)

You should refer to *http://www.smooks.org/mediawiki/index.php?title=Visitor* for more details on the Visitor pattern in use here.

The Smooks configuration for creating the NodeModels for this message are:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!--
    Filter the message using the SAX Filter (i.e. not DOM, so no
    intermediate DOM for the "complete" message - there are "mini" DOMs
    for the NodeModels below)....
    -->
    <params>
```

```
        <param name="stream.filter.type">SAX</param>
        <param name="default.serialization.on">false</param>
    </params>

    <!--
    Create 2 NodeModels. One high level model for the "order"
    (header etc) and then one for the "order-item" elements...
    -->
    <resource-config selector="order,order-item">
        <resource>org.milyn.delivery.DomModelCreator</resource>
    </resource-config>

    <!-- FreeMarker templating configs to be added below... -->
```

Now the FreeMarker templates need to be added. We need to apply 3 templates in total:

1. A template to output the order "header" details, up to but not including the order items.
2. A template for each of the order items, to generate the <item> elements in the <salesorder>.
3. A template to close out the message.

With Smooks, we implement this by defining two FreeMarker templates. One to cover #1 and #3 (combined) above, and a second to cover the <item> elements.

The first FreeMarker template is targeted at the <order-items> element and looks as follows:

```
<ftl:freemarker applyOnElement="order-items">
        <ftl:template><!--<salesorder>
        <details>
        <orderid>${order.@id}</orderid>
        <customer>
        <id>${order.header.customer.@number}</id>
        <name>${order.header.customer}</name>
        </customer>
        </details>
        <itemList>
        <?TEMPLATE-SPLIT-PI?>
        </itemList>
        </salesorder>-->
        </ftl:template>
        </ftl:freemarker>
```

You will notice the <?TEMPLATE-SPLIT-PI?> Processing Instruction. This tells Smooks where to split the template, outputting the first part of the template at the start of the <order-items> element, and the other part at the end of the <order-items> element. The <item> element template (the second template) will be output in between.

The second FreeMarker template is very straightforward. It simply outputs the <item> elements at the end of every <order-item> element in the source message:

```
<ftl:freemarker applyOnElement="order-item">
```

```
    <ftl:template>
    <!--
        <item>
            <id>${.vars["order-item"].@id}</id>
            <productId>${.vars["order-item"].product}</productId>
            <quantity>${.vars["order-item"].quantity}</quantity>
            <price>${.vars["order-item"].price}</price>
        </item>
    -->
    </ftl:template>
</ftl:freemarker>
```

Because the second template fires on the end of the <order-item> elements, it effectively generates output into the location of the <?TEMPLATE-SPLIT-PI?> Processing Instruction in the first template. Note that the second template could have also referenced data in the "order" NodeModel.

This is available, in the Tutorials section, as an executable example.

This approach to performing a one-to-one transformation upon a huge message works simply because the only objects in memory at any one time are the details of the order header and the current <order-item>. These are in the Virtual Object Model. Obviously, it cannot work if the transformation is so obscure as to always require full access to all the data in the source message, if, for example, the messages needs to have all the order items reversed or sorted. In such a case however, you do have the option of routing the order details and items to a database and then using the latter's storage, query and paging features to perform the transformation.

11.2. Splitting and Routing

Another common approach to processing huge messages is to split them out into smaller messages that can be processed independently. Of course Splitting and Routing is not just a solution for processing huge messages. It is often needed with smaller messages, too (message size may be irrelevant) where, for example, order items in a message need to be split out and routed (based on content) to different departments or partners for processing.

Under these conditions, the message formats required at the different destinations may also vary:

- "destination1" required XML via the file system,
- "destination2" requires Java objects via a JMS Queue,
- "destination3" picks the messages up from a table in a Database etc.
- "destination4" requires EDI messages via a JMS Queue,

With Smooks, all of the above is possible. You can perform multiple splitting and routing operations to multiple destinations (of different types) in a single pass over a message.

The key to processing huge messages is to make sure that you always maintain a small memory footprint. You can do this using the Javabean Cartridge by making sure you are only binding the most relevant message data into the bean context at any one time. In the following sections, the examples are all based on splitting and routing of order-items out of an order message. The solutions that are shown all happen to work for huge messages because the Smooks Javabean Cartridge binding configurations are implemented such that the only data held in memory at any given time are the details for the main order and the "current" order item.

Complex splitting operations are supported through use of the Javabean Cartridge. It extracts the data from the split message. In this way, you can extract and recombine data from across different sub-hierarchies of the source message, to produce the split messages. It also means you can easily generate the split messages in a range of different formats, through the use of templates. There is more about this later in the chapter.

11.2.1. Routing to File

File-based routing is performed via the **`<file:outputStream>`** configuration from the *http://www.milyn.org/xsd/smooks/file-routing-1.1.xsd* configuration namespace.

This section illustrates how you can combine the following Smooks functionality to split a message out into smaller messages on the file system.

1. The Javabean Cartridge for extracting data from the message and holding it in variables in the bean context. In this case, we could also use DOM NodeModels for capturing the order and order-item data to be used as the templating data models.

2. The <file:outputStream> configuration from the "'Routing Cartridge"' for managing file system streams (naming, opening, closing, throttling creation etc).

3. The Templating Cartridge (FreeMarker Templates) for generating the individual split messages from data bound in the bean context by the Javabean Cartridge (see #1 above). The templating result is written to the file output stream (#2 above).

In the example, we want to process a huge order message and route the individual order item details to file. The following illustrates what we want to achieve. As you can see, the split messages don't just contain data from the order item fragments. They also contain data from the order header and root elements.

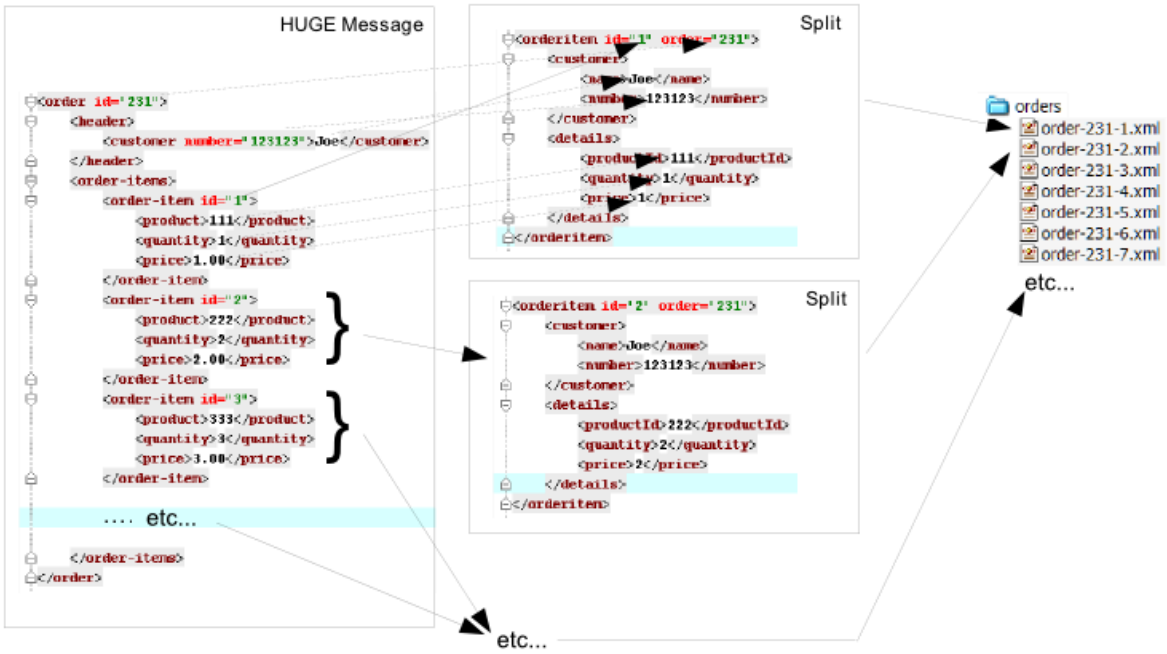

Figure 11.3. File Split Required

To achieve this with Smooks, we assemble the following Smooks configuration:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.2.xsd"
    xmlns:file="http://www.milyn.org/xsd/smooks/file-routing-1.1.xsd"
    xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!--
    Filter the message using the SAX Filter (i.e. not DOM, so no
    intermediate DOM, so we can process huge messages...
    -->

    <params><param name="stream.filter.type">SAX</param></params>

    <!--
    Extract and decode data from the message. Used in the freemarker
    template (below). Note that we could also use a NodeModel here...
    -->
    <!-- (1) -->
    <jb:bean beanId="order" class="java.util.Hashtable"
        createOnElement="order">
        <jb:value property="orderId" decoder="Integer" data="order/@id"/>
        <jb:value property="customerNumber" decoder="Long"
            data="header/customer/@number"/>
        <jb:value property="customerName" data="header/customer"/>
        <jb:wiring property="orderItem" beanIdRef="orderItem"/>
    </jb:bean>
    <!-- (2) -->
    <jb:bean beanId="orderItem" class="java.util.Hashtable"
        createOnElement="order-item">
        <jb:value property="itemId"
            decoder="Integer" data="order-item/@id"/>
        <jb:value property="productId" decoder="Long"
            data="order-item/product"/>
        <jb:value property="quantity" decoder="Integer"
            data="order-item/quantity"/>
        <jb:value property="price" decoder="Double"
            data="order-item/price"/>
    </jb:bean>

    <!--
    Create/open a file output stream. This is writen to by the
    freemarker template (below)..
    -->
    <!-- (3) -->
    <file:outputStream openOnElement="order-item"
        resourceName="orderItemSplitStream">
        <file:fileNamePattern>
            order-${order.orderId}-${order.orderItem.itemId}.xml
        </file:fileNamePattern>
        <file:destinationDirectoryPattern>
            target/orders
        </file:destinationDirectoryPattern>
```

```
        <file:listFileNamePattern>
            order-${order.orderId}.lst
        </file:listFileNamePattern>
        <file:highWaterMark mark="10"/>
    </file:outputStream>

    <!--
    Every time we hit the end of an <order-item> element, apply this
    freemarker template, outputting the result to the
    "orderItemSplitStream" OutputStream, which is the file output
    stream configured above.
    -->
    <!-- (4)  -->
    <ftl:freemarker applyOnElement="order-item">
        <ftl:template>target/classes/orderitem-split.ftl</ftl:template>
        <ftl:use>
            <!--
            Output the templating result to the "orderItemSplitStream"
            file output stream...
            -->
            <ftl:outputTo outputStreamResource="orderItemSplitStream"/>
        </ftl:use>
    </ftl:freemarker>

</smooks-resource-list>
```

Smooks Resource configuration #1 and #2 define the Java Bindings for extracting the order header information (config #1) and the order-item information (config #2). This is the key to processing a huge message; making sure that we only have the current order item in memory at any one time. The Smooks Javabean Cartridge manages all this for you, creating and recreating the orderItem beans as the <order-item> fragments are being processed.

The '"<file:outputStream>"' configuration in configuration #3 manages the generation of the files on the file system. As you can see from the configuration, the file names can be dynamically constructed from data in the bean context. You can also see that it can throttle the creation of the files via the "highWaterMark" configuration parameter. This helps you manage file creation so as not to overwhelm the target file system.

Smooks Resource configuration #4 defines the FreeMarker templating resource used to write the split messages to the OutputStream created by the <file:outputStream> (config #3). See how config #4 references the <file:outputStream> resource. The Freemarker template is as follows:

```
<orderitem id="${.vars["order-item"].@id}" order="${order.@id}">
    <customer>
        <name>${order.header.customer}</name>
        <number>${order.header.customer.@number}</number>
    </customer>
    <details>
        <productId>${.vars["order-item"].product}</productId>
        <quantity>${.vars["order-item"].quantity}</quantity>
        <price>${.vars["order-item"].price}</price>
    </details>
```

```
</orderitem>
```

11.2.2. Routing to JMS

JMS routing is performed via the the `<jms:router>` configuration from the *http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd* configuration namespace.

The following is an example `<jms:router>` configuration that routes an **`orderItem_xml`** bean to a JMS Queue named "smooks.exampleQueue". You should also refer to *Section 11.2.1, "Routing to File"*.

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:jms="http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd"
    xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!--
    Filter the message using the SAX Filter (i.e. not DOM, so no
    intermediate DOM, so we can process huge messages...
    -->
    <params><param name="stream.filter.type">SAX</param></params>

    <!-- (1) -->
    <resource-config selector="order,order-item">
        <resource>org.milyn.delivery.DomModelCreator</resource>
    </resource-config>

    <!-- (2) -->
    <jms:router routeOnElement="order-item" beanId="orderItem_xml"
        destination="smooks.exampleQueue">
        <jms:message>
            <!-- Need to use special FreeMarker variable ".vars" -->
            <jms:correlationIdPattern>
                ${order.@id}-${.vars["order-item"].@id}
            </jms:correlationIdPattern>
        </jms:message>
        <jms:highWaterMark mark="3"/>
    </jms:router>

    <!-- (3) -->
    <ftl:freemarker applyOnElement="order-item">
        <!--
        Note in the template that we need to use the special FreeMarker
        variable ".vars" because of the hyphenated variable names
        ("order-item"). See http://freemarker.org/docs/ref_specvar.html.
        -->
        <ftl:template>/orderitem-split.ftl</ftl:template>
        <ftl:use>
            <!--
            Bind the templating result into the bean context, from where
            it can be accessed by the JMSRouter (configured above).
            -->
```

```
            <ftl:bindTo id="orderItem_xml"/>
        </ftl:use>
    </ftl:freemarker>

</smooks-resource-list>
```

In this case, we route the result of a FreeMarker templating operation to the JMS Queue (i.e. as a String). We could also have routed a full Object Model, in which case it would be routed as a Serialized ObjectMessage.

11.2.3. Routing to a Database using SQL

Routing to a Database is also quite easy. Please read *Section 11.2.1, "Routing to File"* before reading this section.

So we take the same scenario as with the File Routing example above, but this time we want to route the order and order item data to a Database. This is what we want to achieve:

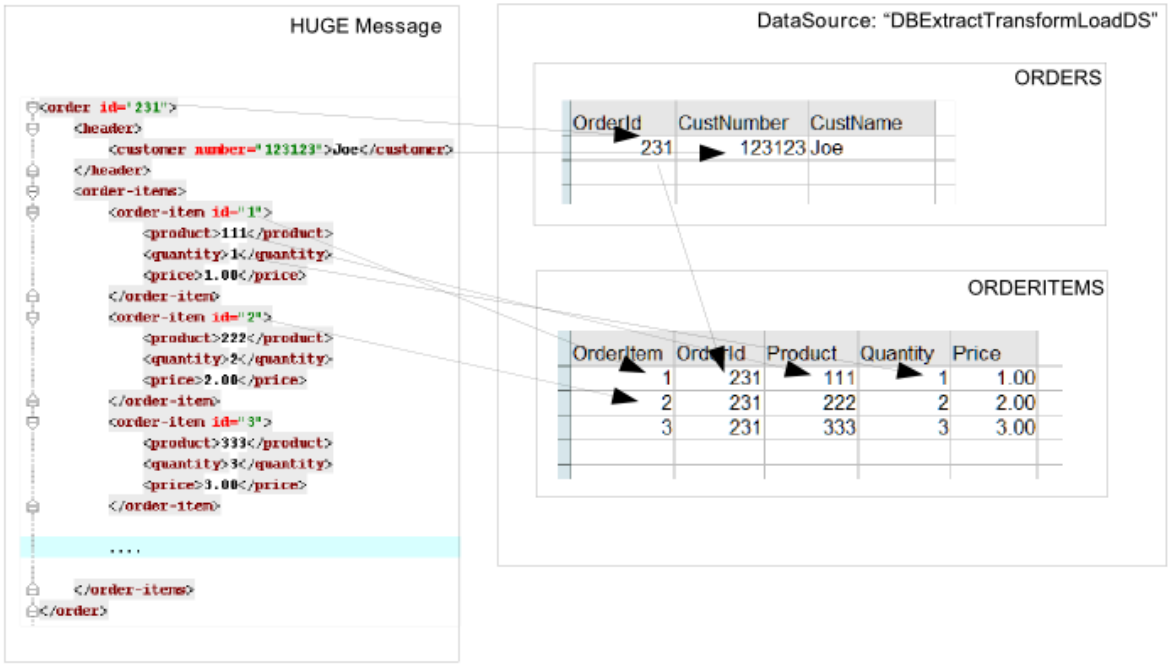

Figure 11.4. Database Split Required

First we need to define a set of Java bindings that extract the order and order-item data from the data stream:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.2.xsd">

    <!-- Extract the order data... -->
    <jb:bean beanId="order" class="java.util.Hashtable"
        createOnElement="order">
        <jb:value property="orderId" decoder="Integer" data="order/@id"/>
        <jb:value property="customerNumber" decoder="Long"
            data="header/customer/@number"/>
```

```
        <jb:value property="customerName" data="header/customer"/>
    </jb:bean>

    <!-- Extract the order-item data... -->
    <jb:bean beanId="orderItem" class="java.util.Hashtable"
        createOnElement="order-item">
        <jb:value property="itemId" decoder="Integer"
            data="order-item/@id"/>
        <jb:value property="productId" decoder="Long"
            data="order-item/product"/>
        <jb:value property="quantity" decoder="Integer"
            data="order-item/quantity"/>
        <jb:value property="price" decoder="Double"
            data="order-item/price"/>
    </jb:bean>

</smooks-resource-list>
```

Next we need to define datasource configuration and a number of <db:executor> configurations that will use that datasource to insert the data that was bound into the Java Object model into the database.

The Datasource configuration (namespace *http://www.milyn.org/xsd/smooks/datasource-1.1.xsd*) :

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:ds="http://www.milyn.org/xsd/smooks/datasource-1.1.xsd">

    <ds:direct bindOnElement="#document"
        datasource="DBExtractTransformLoadDS"
        driver="org.hsqldb.jdbcDriver"
        url="jdbc:hsqldb:hsql://localhost:9201/milyn-hsql-9201"
        username="sa"
        password=""
        autoCommit="false" />

</smooks-resource-list>
```

The <db:executor> configurations (namespace *http://www.milyn.org/xsd/smooks/db-routing-1.1.xsd*):

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:db="http://www.milyn.org/xsd/smooks/db-routing-1.1.xsd">

    <!--
    Assert whether it's an insert or update. Need to do this just
    before we do the insert/update...
    -->
    <db:executor executeOnElement="order-items"
        datasource="DBExtractTransformLoadDS" executeBefore="true">
        <db:statement>
            select OrderId from ORDERS where OrderId = ${order.orderId}
        </db:statement>
```

```
        <db:resultSet name="orderExistsRS"/>
    </db:executor>

    <!--
    If it's an insert (orderExistsRS.isEmpty()), insert the order before
    we process the order items...
    -->
    <db:executor executeOnElement="order-items"
        datasource="DBExtractTransformLoadDS" executeBefore="true">
        <condition>orderExistsRS.isEmpty()</condition>
        <db:statement>
            INSERT INTO ORDERS VALUES(${order.orderId},
            ${order.customerNumber}, ${order.customerName})
        </db:statement>
    </db:executor>

    <!-- And insert each orderItem... -->
    <db:executor executeOnElement="order-item"
        datasource="DBExtractTransformLoadDS" executeBefore="false">
        <condition>orderExistsRS.isEmpty()</condition>
        <db:statement>
            INSERT INTO ORDERITEMS VALUES (${orderItem.itemId},
            ${order.orderId}, ${orderItem.productId},
            ${orderItem.quantity}, ${orderItem.price})
        </db:statement>
    </db:executor>

    <!-- Ignoring updates for now!! -->

</smooks-resource-list>
```

Check out the db-extract-transform-load example.

Message Splitting and Routing

Please refer to *Section 11.2, “Splitting and Routing”*.

Persistence (Database Reading and Writing)

There are three methods for reading and writing to a database from within Smooks. As with many other features of Smooks, this capability relies heavily on the Smooks Java Binding capabilities provided in the Javabean Cartridge, or extends it:

1. Use a JDBC Datasource to access a database and use SQL statements to read from and write to the Database. This capability is provided through the Smooks Routing Cartridge. Refer to *Section 11.2.3, "Routing to a Database using SQL"*.

2. Use an entity persistence framework (like Ibatis, Hibernate or any JPA compatible framework) to access a database and use its query language or CRUD methods for reading and writing. Refer to *Section 13.1, "Entity Persistence Frameworks"* for more details.

3. Use custom Data Access Objects (DAO) to access a database and use its CRUD methods for reading and writing. Refer to *Section 13.2, "DAO Support"*.

Note

Be sure to read *Chapter 4, Java Binding* .

13.1. Entity Persistence Frameworks

With the new Smooks Persistence cartridge in Smooks 1.2, you can directly use several entity persistence frameworks from within Smooks (Hibernate, JPA etc).

Let's take a look at a Hibernate example. The same principals follow for any JPA compliant framework.

The data we are going to process is an XML order message. It should be noted however, that the input data could also be CSV, JSON, EDI, Java or any other structured/hierarchical data format. The same principals apply, no matter what the data format is!

```
<order>
    <ordernumber>1</ordernumber>
    <customer>123456</customer>
    <order-items>
        <order-item>
            <product>11</product>
            <quantity>2</quantity>
        </order-item>
        <order-item>
            <product>22</product>
            <quantity>7</quantity>
        </order-item>
    </order-items>
</order>
```

The Hibernate entities are:

```
@Entity
@Table(name="orders")
public class Order {
    @Id
    private Integer ordernumber;

    @Basic
    private String customerId;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
    private List orderItems = new ArrayList();

    public void addOrderLine(OrderLine orderLine) {
        orderItems.add(orderLine);
    }

    // Getters and Setters....
}

@Entity
@Table(name="orderlines")
public class OrderLine {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    @ManyToOne
    @JoinColumn(name="orderid")
    private Order order;

    @Basic
    private Integer quantity;

    @ManyToOne
    @JoinColumn(name="productid")
    private Product product;

    // Getters and Setters....
}

@Entity
@Table(name = "products")
@NamedQuery(name="product.byId", query="from Product p where p.id = :id")
public class Product {

    @Id
    private Integer id;

    @Basic
    private String name;
```

```
    // Getters and Setters....
}
```

What we want to do here is to process and persist the <order>. First thing we need to do is to bind the order data into the Order entities (Order, OrderLine and Product). To do this we need to:

1. Create and populate the **`Order`** and **`OrderLine`** entities using the Java Binding framework.
2. Wire each **`OrderLine`** instance into the **`Order`** instance.
3. Into each **`OrderLine`** instance, we need to lookup and wire in the associated order line Product entity.
4. And finally, we need to insert (persist) the Order instance.

To do this, we need the following Smooks configuration:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.2.xsd"
    xmlns:dao="http://www.milyn.org/xsd/smooks/persistence-1.2.xsd">

    <jb:bean beanId="order" class="example.entity.Order"
        createOnElement="order">
        <jb:value property="ordernumber" data="ordernumber" />
        <jb:value property="customerId" data="customer" />
        <jb:wiring setterMethod="addOrderLine" beanIdRef="orderLine" />
    </jb:bean>

    <jb:bean beanId="orderLine" class="example.entity.OrderLine"
        createOnElement="order-item">
        <jb:value property="quantity" data="quantity" />
        <jb:wiring property="order" beanIdRef="order" />
        <jb:wiring property="product" beanIdRef="product" />
    </jb:bean>

    <dao:locator beanId="product" lookupOnElement="order-item"
        onNoResult="EXCEPTION" uniqueResult="true">
        <dao:query>from Product p where p.id = :id</dao:query>
        <dao:params>
            <dao:value name="id" data="product" decoder="Integer" />
        </dao:params>
    </dao:locator>

    <dao:inserter beanId="order" insertOnElement="order" />

</smooks-resource-list>
```

If we want to use the named query "productById" instead of the query string then the DAO locator configuration will look like this:

```
<dao:locator beanId="product" lookupOnElement="order-item"
```

```
    lookup="product.byId" onNoResult="EXCEPTION" uniqueResult="true">
    <dao:params>
        <dao:value name="id" data="product" decoder="Integer"/>
    </dao:params>
</dao:locator>
```

The following code executes Smooks. Note that we use a SessionRegister object so that we can access the Hibernate Session from within Smooks.

```
Smooks smooks = new Smooks("smooks-config.xml");

ExecutionContext executionContext = smooks.createExecutionContext();

// The SessionRegister provides the bridge between Hibernate and the
// Persistence Cartridge. We provide it with the Hibernate session.
// The Hibernate Session is set as default Session.
DaoRegister register = new SessionRegister(session);

// This sets the DAO Register in the executionContext for Smooks
// to access it.
PersistenceUtil.setDAORegister(executionContext, register);

Transaction transaction = session.beginTransaction();

smooks.filterSource(executionContext, source);

transaction.commit();
```

13.2. DAO Support

Now let's take a look at a DAO based example. The example will read an XML file containing order information. This example reads an data from an XML file but it works just as easily for EDI, CSV or other formats. Using the javabean cartridge, it will bind the XML data into a set of entity beans. Using the id of the products within the order items (the <product> element) it will locate the product entities and bind them to the order entity bean. Finally, the order bean will be persisted.

The order XML message looks like this:

```
<order>
    <ordernumber>1</ordernumber>
    <customer>123456</customer>
    <order-items>
        <order-item>
            <product>11</product>
            <quantity>2</quantity>
        </order-item>
        <order-item>
            <product>22</product>
            <quantity>7</quantity>
        </order-item>
    </order-items>
```

```
</order>
```

The following custom DAO will be used to persist the Order entity:

```
@Dao
public class OrderDao {

    private final EntityManager em;

    public OrderDao(EntityManager em) {
        this.em = em;
    }

    @Insert
    public void insertOrder(Order order) {
        em.persist(order);
    }
}
```

When looking at this class you should notice the `@Dao` and `@Insert` annotations. The `@Dao` annotation declares that the **`OrderDao`** is a DAO object. The `@Insert` annotation declares that the `insertOrder` method should be used to insert **`Order`** entities.

The following custom DAO will be used to lookup the Product entities:

```
@Dao
public class ProductDao {

    private final EntityManager em;

    public ProductDao(EntityManager em) {
        this.em = em;
    }

    @Lookup(name = "id")
    public Product findProductById(@Param("id")int id) {
        return em.find(Product.class, id);
    }
}
```

When looking at this class, you should notice the `@Lookup` and `@Param` annotation. The `@Lookup` annotation declares that the `ProductDao#findByProductId` method is used to lookup **`Product`** entities. The name parameter in the `@Lookup` annotation sets the lookup name reference for that method. When the name isn't declared, the method name will be used. The optional `@Param` annotation let's you name the parameters. This creates a better abstraction between Smooks and the DAO. If you don't declare the `@Param` annotation the parameters are resolved by their position.

The Smooks configuration look likes this:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
```

```
    xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.2.xsd"
    xmlns:dao="http://www.milyn.org/xsd/smooks/persistence-1.2.xsd">

    <jb:bean beanId="order" class="example.entity.Order"
        createOnElement="order">
        <jb:value property="ordernumber" data="ordernumber"/>
        <jb:value property="customerId" data="customer"/>
        <jb:wiring setterMethod="addOrderLine" beanIdRef="orderLine"/>
    </jb:bean>

    <jb:bean beanId="orderLine" class="example.entity.OrderLine"
        createOnElement="order-item">
        <jb:value property="quantity" data="quantity"/>
        <jb:wiring property="order" beanIdRef="order"/>
        <jb:wiring property="product" beanIdRef="product"/>
    </jb:bean>

    <dao:locator beanId="product" dao="product" lookup="id"
        lookupOnElement="order-item" onNoResult="EXCEPTION">
        <dao:params>
            <dao:value name="id" data="product" decoder="Integer"/>
        </dao:params>
    </dao:locator>

    <dao:inserter beanId="order" dao="order" insertOnElement="order"/>

</smooks-resource-list>
```

The following code executes Smooks:

```
Smooks smooks=new Smooks("./smooks-configs/smooks-dao-config.xml");
ExecutionContext executionContext=smooks.createExecutionContext();

// The register is used to map the DAO's to a DAO name. The DAO name isbe
 used in
// the configuration.
// The MapRegister is a simple Map like implementation of the DaoRegister.
DaoRegister<object>register = MapRegister.builder()
.put("product",new ProductDao(em))
.put("order",new OrderDao(em))
.build();

PersistenceUtil.setDAORegister(executionContext,mapRegister);

// Transaction management from within Smooks isn't supported yet,
// so we need to do it outside the filter execution
EntityTransaction tx=em.getTransaction();
tx.begin();

smooks.filter(new StreamSource(messageIn),null,executionContext);
```

```
tx.commit();
```

Message Enrichment

When using the Persistence features of Smooks, the queried data is bound to the bean context (ExecutionContext). You can use the bound query data to enrich your messages e.g. where you are splitting and routing.

Refer to *Chapter 13, Persistence (Database Reading and Writing)* for additional details.

Global Configurations

Global configuration settings are configuration options that can be set once and be applied to all resources in a configuration.

Smooks supports two types of globals: default properties and global parameters.

Default Properties
: Specify default values for `<resource-config>` attributes. These defaults are automatically applied to **`SmooksResourceConfiguration`** when the corresponding `<resource-config>` does not specify the attribute.

Global Configuration Parameters
: Every `<resource-config>` in a Smooks configuration can specify `<param>` elements for configuration parameters. These parameter values are available at runtime through the **`SmooksResourceConfiguration`**, or are reflectively injected through the `@ConfigParam` annotation.

  Global Configuration Parameters are defined in one place and are accessible to all runtime components using the **`ExecutionContext`**.

Refer to *http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/cdr/SmooksResourceConfiguration.html* and *http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/container/ExecutionContext.html* for more details.

15.1. Default Properties

Default properties are properties that can be set on the root element of a Smooks configuration and have them applied to all resource configurations in the **`smooks-conf.xml`** file.

For example, if all the resource configurations have the same selector value, you could specify a `default-selector=order` instead of specifying the selector on on every resource configuration:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:xsl="http://www.milyn.org/xsd/smooks/xsl-1.1.xsd"
    default-selector="order">

    <resource-config>
        <resource>com.acme.VisitorA</resource>
        ...
    </resource-config>

    <resource-config>
        <resource>com.acme.VisitorB</resource>
        ...
    </resource-config>

<smooks-resource-list>
```

The following default configuration options are available:

default-selector
: Selector that will be applied to all resource-config elements in the smooks configuration file, where a selector is not defined.

default-selector-namespace
: The default selector namespace, where a namespace is not defined.

default-target-profile
: Default target profile that will be applied to all resources in the smooks configuration file, where a target-profile is not defined.

default-condition-ref
: Refers to a global condition by the conditions id. This condition is applied to resources that define an empty "condition" element (i.e. <condition/>) that does not reference a globally defined condition.

15.2. Global Configuration Parameters

Global properties differ from the default properties in that they are not specified on the root element and are not automatically applied to resources.

Global parameters are specified in a `<params>` element:

```
<params>
    <param name="xyz.param1">param1-val</param>
</params>
```

Global Configuration Parameters are accessible via the **`ExecutionContext`**.

```
public void visitAfter(
    final Element element, final ExecutionContext executionContext)
    throws SmooksException
    {
        String param1 = executionContext.getConfigParameter(
            "xyz.param1", "defaultValueABC");

        ....
}
```

15.2.1. Global Filter Setting Parameters

The following global configuration options are available for configuring Smooks filtering.

See *http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/transform/stream/StreamResult.html* and *http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/Smooks.html#filterSource(javax.xml.transform.Source,%20javax.xml.transform.Result...)*.

stream.filter.type
: Determines the type of processing model that will be used. Either **`SAX`** or **`DOM`**. Default is **`DOM`**. Refer to *Section 2.4, "Filtering Process Selection (Should DOM or SAX be Used?)"* for more information about the processing models.

`default.serialization.on`
: Determines if default serialization should be switched on, the default is **`"true"`**. Default serialization tells Smooks to locate a **`StreamResult`** (or **`DOMResult`**) in the Result objects provided to the `Smooks.filterSource` method and to, by default, serialize all events to that Result.

  This behavior can be turned off using this global configuration parameter and can be overridden on a per fragment basis by targeting a Visitor implementation at that fragment. The Visitor can take ownership of the Result writer in the case of SAX filtering, or modify the DOM in the case of DOM filtering. As an example of this, see the *http://www.milyn.org/javadoc/v1.2/smooks-cartridges/templating/org/milyn/templating/freemarker/FreeMarkerTemplateProcessor.html*

`terminate.on.visitor.exception`
: Determines whether an exception should terminate processing. The default is **`true`**.

`maintain.element.stack`
: Controls whether or not Smooks should maintain a contextual stack of element names from the source event stream.. Default value is **`"true"`**. This stack effectively allows Visitor logic to "see" where they are in a message hierarchy. There is a performance overhead associated with this however and it is not always required.

`close.source`
: Determines whether to close source instance streams passed to the `Smooks.filterSource` method. The default is **`"true"`**. The exception here is **`System.in`**, which will never be closed.

`close.result`
: Determines whether to close Result streams passed to the `Smooks.filterSource` method. Default is **`"true"`**. The exceptions here are **`System.out`** and **`System.err`**, which will never be closed.

Multiple Outputs/Results

This section looks at the different ways in which Smooks can produce output from the Filtering process.

Smooks can present output to the outside world in the following ways:

In Result Instances
: Returned in the **`Result`** instances passed to the `Smooks.filterSource` method.

During the Filtering Process
: Output generated and sent to external endpoints during the Filtering process. Possible external endpoints could include ESB Services, Files, JMS Destinations, and databases. This is where message fragment events are used to trigger routing of message fragments to external endpoints, for instance when splitting and routing fragments of a message. Refer to *Section 11.2, "Splitting and Routing"*.

A very important point to remember is that Smooks can generate output or results in either or both of the above ways, all in a single filtering pass of a message stream. It doesn't need to filter a message stream multiple times in order to generate multiple outputs/results. This is important for performance.

16.1. In Result Instances

This is the most common method of capturing output from the Smooks filtering process.

A look at the Smooks API reveals that Smooks can be supplied with multiple **`Result`** instances:

```
public void filterSource(Source source, Result... results) throws
 SmooksException
```

In terms of the types of **`Result`** that Smooks can work with, we're talking about the standard JDK **`StreamResult`** and **`DOMResult`** types, as well as some Smooks specializations:

`JavaResult`
: Result type for capturing the contents of the Smooks Java Bean context. Refer to *http://www.milyn.org/javadoc/smooks/org/milyn/payload/JavaResult.html*.

`ValidationResult`
: Result type for capturing Validation results. Refer to *Chapter 10, Validation* and *http://www.milyn.org/javadoc/smooks-cartridges/validation/org/milyn/validation/ValidationResult.html*.

`StringResult`
: Simple Result type used mainly when writing tests. It is a simple **`StreamResult`** extension wrapping a **`StringWriter`**. Refer to *http://www.milyn.org/javadoc/smooks/org/milyn/payload/StringResult.html*.

Note

As yet, Smooks does not support capturing of result data to multiple **`Result`** instances of the same type. For example, you can specify multiple **`StreamResult`** instances in the **`Smooks.filterSource`** method call, but Smooks will only output to first one declared.

16.1.1. StreamResults / DOMResults

These **`Result`** types receive special attention from Smooks. As Smooks processes the message **`Source`**, it produces a stream of events. If a **`StreamResult`** or **`DOMResult`** is supplied in the `Smooks.filterSource` call, Smooks will, by default , serialize the event stream to the supplied **`StreamResult`** or **`DOMResult`** as XML. Visitor logic can be applied to the event stream before serialization.

Important

This behaviour depends on the parameter **`default.serialization.on`** begin set to **`true`**. See *Section 15.2.1, "Global Filter Setting Parameters"*

This is the mechanism used to perform a standard **`1-input/1-xml-output`** character-based transformation.

16.2. During the Filtering Process

Smooks is also able to generate different types of output during the `Smooks.filterSource` process i.e. as it is filtering the message event stream, before it reaches the end of the message. A classic example of this is when to splitting and routing message fragments to different types of endpoints for processing by other processes.

Smooks doesn't "batch up" the message data and produce all the results/outputs after filtering the complete message. This is for performance, because you can utilize the message event stream to trigger the fragment transform and routing operations.

For instance, you might have an Order message that has hundreds of thousands (or even millions) of Order Items that need to be split out and routed to different departments in different formats, based on different criteria. The only effective way of handing messages of this magnitude is by streaming the process.

Performance Tuning

As with any software the performance of Smooks will suffer if it is configured or used incorrectly.

17.1. General

Cache and Reuse the Smooks Object.

Initialization of Smooks takes some time and therefore it is important that it is reused.

Only use the HTMLReportGenerator in development

When enabled, the HTMLReportGenerator incurs a significant performance overhead and with large message, can even result in OutOfMemory exceptions.

If possible, use SAX filtering

SAX processing is a lot faster than DOM processing and has a consistently small memory footprint. It is mandatory for processing large messages. However, you need to check that all of your Smooks Cartridges are SAX compatible. Refer to *Section 2.4, “Filtering Process Selection (Should DOM or SAX be Used?)”*.

Contextual selectors

Contextual selectors can obviously have a negative effect on performance e.g. evaluating a match for a selector like **`"a/b/c/d/e"`** will obviously require more processing than that of a selector like **`"d/e"`**. Obviously there will be situations where your data model will require deep selectors, but where it does not, you should try to optimize your selectors for performance.

17.2. Smooks Cartridges

You should refer to the documentation for each specific cartridge for performance optimization advice.

17.3. Javabean Cartridge

Where possible avoid using the Virtual Bean Model, and create Beans instead of maps. Creating and adding data to Maps is a lot slower then creating POJO's and calling the setter methods.

ESB Integration

Smooks plugins are available for a number of ESBs:

JBoss ESB
http://wiki.jboss.org/wiki/Wiki.jsp?page=MessageTransformation

Mule
http://www.mulesource.org/display/SMOOKS/Home

Apache Synapse/WS02
http://esbsite.org/resources.jsp?path=/mediators/upul/Smooks%20Transform%20Mediator

Testing

19.1. Unit Testing

Unit testing with Smooks is simple:

```
public class MyMessageTransformTest
{
    @Test
    public void test_transform() throws IOException, SAXException
    {
        Smooks smooks = new Smooks(
            getClass().getResourceAsStream("smooks-config.xml") );

        try {
            Source source = new StreamSource(
                getClass().getResourceAsStream("input-message.xml" ) );
            StringResult result = new StringResult();

            smooks.filterSource(source, result);

            // compare the expected xml with the transformation result.
            XMLUnit.setIgnoreWhitespace( true );
            XMLAssert.assertXMLEqual(
                new InputStreamReader(
                getClass().getResourceAsStream("expected.xml")),
                new StringReader(result.getResult()));
        } finally {
            smooks.close();
        }
    }
}
```

The test case above uses **XMLUnit**, *http://xmlunit.sourceforge.net*.

The following **Maven** dependency was needed to use **XMLUnit** in the above test:

```
<dependency>
    <groupId>xmlunit</groupId>
    <artifactId>xmlunit</artifactId>
    <version>1.1</version>
</dependency>
```

Appendix A. Revision History

Revision 1.0