Transformer API Developers Guide

*** CONFIDENTIAL ***

Produced by:	Trace Financial Limited 224-232 St. John Street London EC1V 4QR
Version :	4.1
Date:	26/01/15

Document Change Control

Version	Date	Comments
1.0	14/09/2004	Created
1.1	15/09/2004	Updated following comments
1.2	16/09/2004	Updated after API changes, further comments and to have syntax highlighted code.
1.3	21/09/2004	API Changes
1.4	24/06/2005	Minor API Changes
1.5	02/06/2006	Classpath information added.
2.0	15/11/2005	Calling a built project, the coarse API and TBeans added
2.1	04/04/2007	Document rearranged for 3.0 Release
2.2	10/05/2007	Updated following comments
2.3	25/07/2007	Brought up to date prior to Transformer 3.0 release
2.4	22/08/2007	Coarse API Examples added
2.5	25/09/2007	Final changes before Transformer 3.0 release
2.6	31/01/2008	log4j properties example changed
2.7	21/02/2008	XML SAT Section improved
2.8	06/10/2008	Additional Runtime Context Information added.
2.9	04/04/2011	Brought up to date prior to 3.2 release
3.0	17/10/2011	Final update before 3.2 release
3.1	25/06/2012	Update prior to 3.3 release
3.2	07/09/2012	Service Performer and MomNodeBasket API section added
3.3	26/04/2013	NodeSpec Information added prior to 3.4 release
3.4	13/03/2014	Updates following internal review
4.0	25/07/2014	Information on Lookup Tables added
4.1	26/01/2015	Tidy up prior to 3.5 release

Contents

1 Introduction

This document is intended to illustrate how the Transformer API can be used in real world situations. It is intended for both Java programmers who have to deploy a Transformer project that has already been configured and tested and for the writers of new Mappers (and other TBeans) who need to extend the out-of-the-box functionality. A thorough knowledge of Java and Transformer is assumed throughout. A Glossary is provided at the end of the document to familiarise the user with Transformer terminology.

The document runs through a series of common scenarios and highlights the API calls that need to be made to perform the specified task. A zip file is also provided that includes the Transformer project used in the samples and the source code that is shown below.

To aid clarity the data being used is very simple however the calls that are made on real data, however complex, should be exactly as outlined here. The API programmer should note that in all cases, the retrieval of data, the way the data are outputted, and the way in which exception conditions are dealt with, will vary greatly in real world applications. The examples outlined are a balance between clarity and completeness.

For a full description of the Java classes and method calls the programmer should refer to the Transformer API Javadoc.

The Transformer API can be partitioned into different levels:

1) The detailed API that contains all of the low level calls that load definitions, parse messages, perform mappings, format outputs etc.
2) The API for calling Exposed Service Operations which encapsulate multiple low level calls
3) The Coarse API which generates Java code – one method per Exposed Service Operation.

Three other aspects of calling Transformer are also dealt with – namely how to invoke SQL Query Definitions at runtime, how to write simple TBeans and how to use the inbuilt Streaming facilities to process large amounts of data.

Before the samples are dealt with individually a few general points need to be made.

1.1 Java Classpath

Before the samples can be built the following jar files must be added to the Java CLASSPATH. Assuming the user wishes to use the log4J logging mechanism.

- transformer-runtime-with-log4j-<release number>-complete.jar
- currencylib-<release number>.jar

On Windows the runtime jar file can usually be found in the following directory:

C:\Program Files\Trace Financial Ltd\Transformer\<release number>\runtime\single

And the currencies jar can be found in:

C:\Program Files\Trace Financial Ltd\Transformer\<release number>\lib

If the user does not wish to use log4J then separate jars can be added to the classpath to achieve the same goal. All the necessary jars can be found in:

C:\Program Files\Trace Financial Ltd\Transformer\<release number>\runtime\jars

1.2 Transformer Concepts

Project
A Transformer Project is a self-contained set of user-definable configuration files named according to a logical tree structure. Different type of item can be configured, for example Message and Component Definitions, Mapping Definitions, Message Validation Rules, Enrichment Rules etc. These can be used to allow an application to convert input of one message type to output of another type.

Exposed Services and Operations
One particular kind of configuration file defines "Exposed Services". An Exposed Service is a means of defining an "entry point" for an embedding application to use Mappings, Enrichments and Validations via a simple naming mechanism. The person defining the Project can specify a number of Operations within an Exposed Service, give each of them a unique name, and define what Mappings/Enrichments etc. that Operation relates to. An Exposed Service Operation definition also determines the type of input and output parameters which should be supplied to and expected from the Operation.

It is possible to call Transformer at runtime without defining Exposed Services, using the "Detailed API", but in that case the API programmer has to take responsibility for many tasks which can otherwise be left to the encapsulated functionality of the Service Performer API.

Service Builder
Different types of Service Builder are available within Transformer. These expose the services in different ways. One of them generates Java code (POJO), hence the generated Coarse API conventions described later in this document.

1.3 Transformer Project Deployment

An embedded application needs certain resources from the Transformer software installation, i.e., the jars described in section .1.1 above. It also needs access to one or more Transformer Projects. These will be supplied to it in the form of additional jar-files built using the Transformer GUI, or the Transformer design-time command-line interface, or perhaps a build script of some kind that uses that command-line interface.

Definition of the Transformer projects, and production of the project jar-files, may be the responsibility of someone other than the embedded application developer.

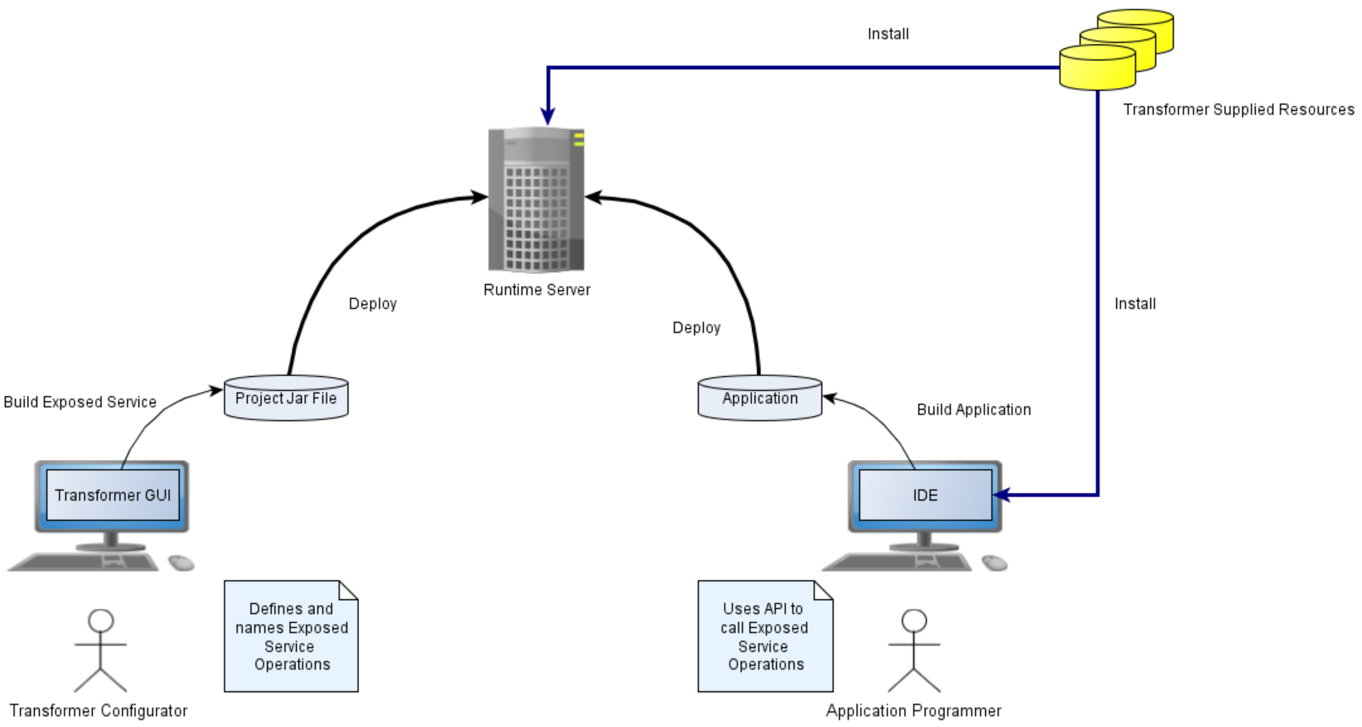
Defining and deploying a Transformer Project involves the following steps:

- Within the project…
    - Define a **project key** value
    - Define a named Exposed Service
    - Define one or more named Service Operations, each of which defines the number and type of inputs and outputs it requires.
- Build the Transformer project to create a jar file.
- The jar-file can then be supplied to the embedding application developer.

To utilise this jar-file and call Exposed Service Operations within the embedding application, the developer must be provided with the following information:

- The **project key**.
- The **name** of the **Exposed Service**.
- The **name(s)** of any **Operations** within the Exposed Service which require to be called.
- For *each* Service Operation, the developer must be told both the type and number of any required inputs and the type and number of any outputs expected.

This is illustrated in the following diagram:



2 Transformer Projects

2.1 What is a Project?

As already explained, a Transformer Project is a self-contained set of user definable configuration files named according to a logical directory type structure. The files can be provided in one or more physical directories or jar files, in a similar fashion to the provision of Java class files in a classpath with one or more entries. In the GUI design tool, one physical directory has a special significance in that it contains editable information, and additional (non-editable) configuration information can be provided in additional directories or jar files, known as libraries. In a runtime deployment, this distinction is unimportant, since the configuration information is used but not changed.

A Project provides two things:

- Separation between configuration information – names are only unique within a Project and nothing in one Project refers to or depends on anything in another Project.
- Caching of configuration information, including the optimized structures which are built from the "raw" configuration files.

Transformer uses a directory type structure to distinguish different configuration items. The structure can be seen in Navigator Pane of the Transformer GUI. From looking at the structure it should be clear that all Transformer objects reside in groups – typically Message Definition or Mapping Definition Groups (although there are other structures for Rules, Tests and Test Data). When an API programmer needs to obtain the Runtime definition of an Object (e.g. Mapping Definition) s/he needs to know the following:

- the name of the Object
- the name of the Group that it resides in
- in some cases the type of the group. The type of the group is important as some objects can reside in both Message Definition and Mapping Definition Groups.

Messages can only exist in Message Definition Groups:

```
MessageDefinition md =
        MessageDefinition.getMessageDefinitionRunTime(project,groupName,messageName);
```

But Mappings can appear in both Message Definition Groups and Mapping Definition Groups so an extra argument is required to the get runtime method:

```
MappingDefinition mapd = MappingDefinition.getMappingDefinitionRunTime(project,
          GroupName.MAPPING_DEFINITION_GROUPS.toString(), mappingGroup, mappingName);


MappingDefinition mapd = MappingDefinition.getMappingDefinitionRunTime(project,
          GroupName.MESSAGE_DEFINITION_GROUPS.toString(), msgGroup, mappingName);
```

The arguments to the method calls are basically the "unknown" directory names in the Navigator Pane directory structure.

Rules take this idea one stage further. A Rule must belong to a Rule Set and must have a target (either Component, Component Type or Message). To identify the target the programmer needs to know the group that the target is in, the name of the target and the type of the target therefore the call to create a rule will look something like this:

```
ValidationRule vr = ValidationRule.getValidationRuleRunTime(project,
                        ruleSetName, targetGroupName, targetType, targetName, ruleName);
```

2.2 Project Loading

The simplest way to create a Project instance via the API is to simply load a project's tpj file, as used in the GUI design tool. This will automatically deal with any libraries used by the project. This method is useful during the testing phase when the code being written and Transformer project exist on the same machine.

```
      Project project = new Project("C:/transformer/apiTest/ApiTest.tpj");
```

There are other Project constructors which use the built form of a project rather than going via a tpj file, but in a production environment, the creation of a Project instance is usually not handled directly by the API programmer, it is left to a TransformerProjectManager.

TransformerProjectManager is an abstract class which manages the loading, caching and sharing of Project instances in an efficient and thread-safe manner. There are different project manager implementations available, which vary according to the mechanism by which the files that form the project are to be loaded. Formally, the API programmer is interacting with an IProjectProvider interface, for which the various TransformerProjectManagers provide implementations. The programmer can request a Project instance from the provider by passing a String **project key**. Each project manager implementation has its own algorithm to interpret the key in a way that allows it to call one of the constructors available in the Project class.

One point to make is that there is an overhead in the loading of the Project. Projects can become very large structures so a caching approach for project loading is a good idea. The implementations of the TransformerProjectManager class provide a caching mechanism.

The main project manager implementations are:

- ClasspathProjectManager – this creates Projects from built jars that are placed on the classpath. The project key that must be passed by the programmer is the same as the Project Key specified when the projects were built.
- DeployDirectoryProjectManager – this creates Projects from built jars that are placed in a deploy directory. The project key that must be passed by the programmer is the same as the Project Key specified when the projects were built.
- OSGiProjectProvider and BundleProjectManager – this creates Projects from OSGi bundles in an OSGi container. The project key that must be passed by the programmer is the same as the Project Key specified when the projects were built.
- FileBasedProjectManager – this creates Projects by loading jar files from disk. The key is the location of the jar file. This implementation pre-dates the build and deploy enhancements that were introduced in Transformer 3.4.
- TpjFileProjectManager – this loads and caches Projects from disk where the key is the name of the tpj file.

2.3 Obtaining Information and Efficiency

All of the examples have project location and names of message definitions and mappings, among others, hard coded but in real world applications where would these variables come from? If the code is highly specific to a particular project or message or is being tested then hard coding these values (as in all the examples) is fine. However, in a more complicated scenario this approach is usually unacceptable. This information would normally be obtained from the "context" in which the code is being used – the most obvious case would be to use JNDI to "look up" the information but there are numerous other possibilities including:

- using the message header or envelope
- obtaining the information from Database using ODBC
- from the message properties in JMS
- pre-parsing the message body to obtain the information.

It is worth considering whether to load commonly used message definitions or mapping definitions at start-up, rather than dynamically loading them in response to events received. This has an advantage of reducing the response time for the first message that is processed and also means that any exceptions arising from loading the information can be dealt with before any messages are processed.

2.4 Project and ProjectManager Lifecycles

Using a project manager involves a pair of method calls.

```
Project project = manager.getProject("com.acme.PaxToSWIFT",this);
```

The first parameter to the getProject method is the project key. The second parameter is the requestor for the Project – once the project is no longer required it should be closed with the following call (note that the project key and the requestor must be the same as in the earlier call):

```
manager.releaseProject("com.acme.PaxToSWIFT",this);
```

The Project will not be closed until it has been released by all requestors. The ProjectManager can be told whether to close projects as soon as it has been released, normally this is not what is done, since the overhead of creating a project instance is quite high. The alternative approach is to explicitly call the "closeIdleProjects" method on the Project manager.

Correct usage of a project manager to ensure that projects are released therefore implies a try/finally approach from the API programmer. There are some utility classes which encapsulate this try/finally in a convenient way, described in section 8.

The DeployDirectoryProjectManager and FileBasedProjectManager can be used to "Hot Swap" the project at runtime. This is useful if you are calling Transformer from a long running application and you do not wish to stop the application to update a Transformer project. Before this task can be performed the user should ensure that releaseProject() is called after every operation is performed using the project.

To actually perform the hot swap the user should overwrite the project jar file and then call dropProjectWhenIdle(String key) on the Project Manager. As before the key is the location of the project on disk. The call to dropProjectWhenIdle should be performed from a different thread to the one do the work (i.e. the one processing messages and calling mappings etc.). The dropProjectWhenIdle will wait until any current mappings have finished and then replace the project. If a mapping (or anything lese that requires access to the project) is called whilst the dropProjectWhenIdle is running then it will block until dropProjectWhenIdle is complete so that the call will use the new version of the project.

Instances of the different project manager implementations can be obtained as follows:

The ClasspathProjectManager uses factory methods rather than a public constructor. There are three methods:

```
public static ClasspathProjectManager getInstance()

public static ClasspathProjectManager getInstance(DataSourcesFactory dsFactory)

public static ClasspathProjectManager getInstance(DataSourcesFactory dsFactory,
                                                  ClassLoader classLoader)
```

Jars produced by building libraries and services (please see the Transformer help system for a description of these processes) can be placed on the classpath available to the specified class loader (if no class loader is specified, the class loader that loaded the ClasspathProjectManager class itself is used).

DataSourcesFactories are explained in section 9 below.

The DeployDirectoryProjectManager has a public constructor that takes two arguments, a deploy directory and a working directory. Jars produced by building libraries and services can be placed in the deploy directory. The project manager will make working copies of these in the working directory. When placing a jar into the deploy directory, a subdirectory structure corresponding to the project key should be used. E.g., for a project key com.acme.messaging.PaxToSWIFT, the jar should be placed in a com/acme/messaging subdirectory of the deploy directory. It also has a second public constructor allowing a DataSourcesFactory to be specified.

The OSGiProjectProvider is an IProjectProvider implementation for use with an OSGi container. When using OSGi the jars produced for libraries and services are OSGi bundles which must be installed and started in the OSGi container. The OSGiProjectProvider is a singleton class which can be accessed via a static method:

```
public static OsgiProjectProvider getInstance()
```

The FileBasedProjectManager has two public constructors, one which takes no arguments and one which specifies a working directory. If the working directory is used, copies of the jars are made in that directory, otherwise a copy of each jar is made alongside it in its original location. There is also an additional constructor allowing a DataSourcesFactory to be specified. The FileBasedProjectManager requires the API programmer to pass the actual name of the jar file (as an absolute path, or relative to the working directory of the embedding application), as a project key. It does not support deployed libraries.

The TpjFileProjectManager has a public no-args constructor. The TpjFileProjectManager is for opening projects which are in their design-time state with a .tpj file, rather than in a built state as a collection of jar files. It requires the API programmer to pass the actual name of the .tpj file (as an absolute path, or relative to the working directory of the embedding application) as a project key.

3 Runtime Context

A Runtime Context is an object which the caller of a NamedActionSet (a mapping or rule etc.) can pass into an execution method to provide contextual information. It serves two purposes:

- It passes information to Transformer core code – specifically it allows database Connection sharing between the embedding application and Transformer.
- It acts as an opaque object which is passed by the caller and can have relevance to the code inside a TBean usually a Mapping action.

For example, the user could create an implementation of the RuntimeContext that has set and get methods for the current user information. Before the mapping is called the RuntimeContext has the user information added to it and then it can be extracted in a suitable mapping action.

A basic RuntimeContext implementation is provided as part of Transformer. The RuntimeContextImpl uses simple HashMaps to implement all the features of the RuntimeContext interface. An instance of the Runtime Context can be created like so:

```
RuntimeContext ctx = new RuntimeContextImpl();
```

Values can be stored against keys in the Context like this:

```
ctx.putUserObject("MY_KEY1","Some value");

ctx.putUserObject("MY_KEY2","Some other value");
```

Values can be retrieved from the Runtime Context in a mapping (or Enrichment etc.) using the GetStringFromGlobal mapping action. The PutStringToGlobal mapping action will add Strings into the Runtime Context in a similar way to the putUserObject method on the Context itself.

The Runtime Context object can then be passed in as a parameter to mapping (or enrichment etc.) calls in both the detailed and the Coarse APIs (examples of how to do this are in the subsequent sections of this document).

After the call to the mapping (or exposed service operation in the Coarse API) has returned the Runtime Context can be used to extract values set during the mapping like so:

```
String value3 = (String)ctx.getUserObject("MY_KEY3");
```

The PutStringToGlobal mapping action is limited to only adding String BOTs into the Runtime Context that is why in the example the value stored in “MY_KEY3” has been cast to a Java String Object.

4 Exceptions

All the examples deal with the different kinds of exception that are thrown during the Transformer API calls. In these simple cases the exception stack is just printed out, however, in real world situations what to do in certain error conditions could vary from stopping the entire application to ignoring the exception completely. To aid exception handling a brief description of each exception is given below as well as the list of calls that throw them.

4.1 ConfigFileStructureException

Occurs if a Config file (a Message Definition, a Mapping Definition etc.) is incorrectly constructed.

Thrown by:

- new Project
- All the Static getRuntime methods
- MessageDefinitionGroup.getMessageDefinitionRunTimeFromContent
- EnrichmentRuleSet.loadRuleSet
- ValidationRuleSet.loadRuleSet

Note that it should not be possible for this to be thrown if the project has been built, since this requires the project to be valid.

4.2 ConfigFileContentException

This is thrown when a Config file has invalid configuration details and would usually have to be fixed in the Transformer application.

Thrown by:

- MessageDefinitionGroup.getMessageDefinitionRunTimeFromContent
- All the Static getRuntime methods
- EnrichmentRuleSet.loadRuleSet
- ValidationRuleSet.loadRuleSet

As with the ConfigFileStructureException, most ConfigFilesContentExceptions will be visible in the GUI design tool and will prevent a project being built. However, it is possible for some to arise due to the environment of the runtime deployment, for example if data sources or other external resources required by definitions are not available, or are inconsistent with other definitions provided at design time.

4.3 UnrecognizedMessageException

This is thrown if the Message Type Identifier (MTI) fails to identify a message. This would usually be caused by an error in the message data or if the MTI is incorrectly configured.

Thrown by:

- MessageDefinitionGroup.getMessageDefinitionRunTimeFromContent

4.4 ParseException

This exception is thrown when the parser fails to break up the message. Again, it is usually a sign that there is some sort of error in the message data.

Thrown by:
- MomNode.makeMom

4.5 WriteMessageContentException

This is thrown when the MomNode cannot be formatted or written out to the specified stream.

Thrown by:
- MomNode.format
- MomNode.formatAsXml

4.6 ActionException

When a mapping, enrichment, condition definition, or rule is defined with the GUI design tool, the user can specify circumstances in which an exception should be thrown, typically relating to missing or unexpected data. ActionException is the general class for this kind of exception, and has several subclasses.

Thrown by any NamedActionSet execution method e.g:
- MomNode.makeNodeViaMap
- MappingDefinition.makeOutputBasket
- ConditionDefinition.evaluate
- EnrichmentRuleSet.enrich
- ValidationRuleSet.validate

4.7 MomNodeException

When the detailed API is used to manipulate MOM structures (Message Object Models), operations can cause MomNodeExceptions to be thrown if the attempted operation would represent an invalid message. There are different subclasses relating to structure exception and content exceptions. The API methods allow the API programmer to choose to suppress the throwing of these exceptions, in which case they are attached to the relevant MOM nodes for subsequent interrogation. However, the method declarations for MOM manipulations always include a throws clause for MomNodeException.

5 Characters and Bytes

The embedding application API programmer and the person defining the Transformer project must agree on the parameter types being used in Exposed Service Operations, and to make the right decisions need to appreciate the nature of the messages being exchanged and the ultimate origin/destination.

5.1 Bits, Bytes and Strings - Background and Terminology

This introductory section is probably very familiar to Java programmers reading this, but it is just set out explicitly to provide the rationale behind the configuration options in Transformer.

The basic point is that computers store and exchange data as streams of bits, and that the "correct" interpretation requires external definition of some kind – whether that be via the protocol, the message standard, knowledge in the embedding application that the counterparty is using an EBCDIC-based IBM mainframe, or whatever.

A byte is just a name for 8 bits, whereas a character is an entirely separate concept sometime defined as "the smallest component of written language that has semantic value". The Unicode consortium has attempted to catalogue all the characters in all the written languages in the world and initially allowed for a total of 65,535 different characters, although there is now an extension to allow for more. The term "String" is used in Java and other programming languages to mean a string of characters, and is also used in that sense in the Transformer configuration GUI.

There are many different conventions specifying how bytes can represent characters, variously referred to as "encodings", "character sets" and "code pages". Any particular computer will have a default encoding (referred to as a "default platform encoding") as part of its operating system configuration which it uses in the absence of more specific instruction when reading/writing disk files or communicating with other systems. Many of the encoding conventions have the same encoding for basic Latin characters but differ with accents, punctuation, currency symbols etc., and for non-Latin characters (Cyrillic, Arabic, Chinese etc.). Some encodings such as the EBCDIC family have different encodings for the basic Latin characters too.
For example, here's how a particular bit pattern may be interpreted with three different encoding systems:

Bit Pattern	*Meaning in Microsoft cp1252*	*Meaning in ISO-8859-2*	*Meaning in UTF-8*
01000001	A (Latin Capital A)	A (Latin Capital A)	A (Latin Capital A)
11000001	Ã (Latin Capital A with Tilde)	Ă (Latin Capital A with Breve)	***Not valid***
10100011	£ (Pound Sign)	Ł (Latin Capital L with Stroke)	***Not valid***
10010011	" (Left Double Quotn. Mark)	***Not valid***	***Not valid***

Note that when an invalid byte sequence is encountered systems often replace it with a ?. Alternatively they may throw an exception.

Conversely here are some examples of how a particular character might be represented as bits with three different encoding systems:

Character	*Bits in Microsoft cp1252*	*Bits in ISO-8859-2*	*Bits in UTF-8*
A	01000001	01000001	01000001
Ã	11000001	***Not representable***	11000011:10000011
Ă	***Not representable***	11000001	11000100:10000010
£	10100011	***Not representable***	11000010:10100011
Ł	***Not representable***	10100011	11000101:10000001
“	10010011	***Not representable***	11100010:10000000:10011100

Note that when a character which cannot be encoded is encountered systems often replace it with a null byte or a specified replacement byte - alternatively they may throw an exception.
It is also important to realise that not all data which is stored or exchanged relates to characters at all. For example, the bit pattern 11111100 could have any of the following meanings – or many others – it is entirely a question of interpretation:

As a character encoded in cp1252	ü (small u with umlaut)
As an unsigned binary number	a numeric value: 252
As a signed binary number	a numeric value: -4

5.2 Relevance to Messaging Applications

When there is an information exchange between systems, the definition of the exchange format may be expressed in terms of characters or in terms of bytes.

When the information is partly or entirely conveying characters, a misunderstanding about how characters are encoded will result in information will be misinterpreted. Sometimes the encoding to use is part of a published standard (e.g., with FIX), sometimes it is not, in which case it has to be agreed on a bilateral or implementation basis.

Equally importantly, when the information is partly or entirely conveying non-character information it is essential to avoid treating it as if it were characters, because this would lead to loss or corruption of the underlying information. For example there are five bytes which are not valid in cp1252 - 10000001, 10001101, 10001111, 10010000 and 1001101. If someone tried to interpret non-character bytes as if they were cp1252, any occurrences of those bytes would end up being treated as 00111111 which is the bit pattern for a question mark, and information has been lost.

In summary, these are the essential questions to which answers must be established for successful information interchange:

- Is some or all of the information non-character, i.e., binary in nature?
- What character set/encoding is to be used to interpret information which does represent characters?

When messages are being exchanged over a particular protocol, e.g., http, the answers may be dictated by the protocol. However not all protocols will provide the answers.

When messages are being exchanged by a protocol or mechanism that does not provide answers, but the messages are XML, the answers to these are part of the XML standard itself:

(a) All the information can be considered as characters (although it is then possible via an additional layer of definition to say that some of those characters represent bytes via completely different convention, Base64 encoding, which is a separate issue and outside the scope of this document).
(b) The encoding to use can vary from message to message, but can be determined from the bytes of each message that is transmitted. The encoding may be specified in the prolog, or if that is absent, a ByteOrderMark (BOM) can appear, or if that is absent too, the encoding will be UTF-8.

When messages are ***not*** XML, and the protocol being used doesn't provide the answers, the answers have to come from a standards body, product vendor, or a bilateral agreement with a counterparty.

5.3 Considerations when Using Transformer

5.3.1 Message Definitions and Exposed Service Operations

Transformer is an embedded transformation engine. It does not directly read disk files or accept network requests. The interaction points between Transformer and the application within which it sits are the Exposed Service Operation definitions.

When you define these in the Transformer GUI you specify whether you want the interaction to be in terms of bytes or Strings.

In Transformer's standard libraries and templates, there are several kinds of message definition which represent the situation where the information being exchanged is not entirely character-based:

- Excel workbooks must always be treated as binary
- Some COBOL messages must be considered binary, e.g., those that contain COMPUTATIONAL fields.
- FIX messages must be considered binary if they are going to contain certain fields, e.g., tag 355.

There are also converters and parsers in the toolkit group that can be used to define messages which don't follow one of the standard templates. Some of these also relate to non-character situations, e.g., the BytesToInteger converter for binary numbers.

For messages which fall into one of the categories which are not entirely character based, the Exposed Service Operation parameter type must be set to "bytes".

For other messages, where the data is entirely character-based, the choice of parameter type is an implementation issue about which software component is best placed to encapsulate the encoding/decoding between bytes and Strings. If the responsibility is best handled in Transformer, the Exposed Service Operation should be set up as bytes – if it is best handled in the embedding application, set up the Exposed Service Operation to use Strings.

5.3.2 Considerations for XML

If the embedding application calls Transformer passing the XML as bytes, Transformer will correctly inspect the prolog / BOM and decode accordingly.

If the embedding application calls Transformer passing the XML as a string, any encoding in the XML prolog is ignored by Transformer, and the correct decoding of the incoming bytes is entirely the responsibility of the embedding application.

When Transformer is producing XML output, it can be configured to specify a particular encoding in the prolog (via the Message Definition Group properties).

If Transformer returns XML as bytes it will have produced those bytes using that specified encoding.

If Transformer returns XML as a String to the embedding application, any encoding that Transformer has placed in the prolog is nothing more than a “hint” to the embedding application. How the String is eventually converted to bytes, and whether that conversion does actually use the encoding specified in Transformer is outside Transformer’s control.

5.3.3 Considerations for non-XML

For non-XML message definition group a Message Content Factory can be configured on the Message Definition Group properties. The responsibilities of the Message Content Factory include encoding/decoding Strings and bytes.

Some templates, such as Excel and Fix, have specific Message Content Factories tied to the template. Other more general purpose templates, or the toolkit, allow more configuration. There are two basic configurable properties:

(a) Are there any binary regions in the messages being exchanged, or does everything represent characters?
(b) What character set / encoding / code page is relevant?

Configuration of a Message Content Factory for a group is optional. If none is configured, the Exposed Service Operation can be set up to use either Strings or bytes. If the communication is in terms of bytes, these are just converted by applying default platform encoding to the entire message.

If a Message Content Factory is configured, and specifies that there are binary regions, the Exposed Service Operation must be set up to use bytes. If there are some fields in the message which are string-based (as there always will be), these will be converted individually using the specified character set.

If a Message Content Factory is configured, and specifies that there are no binary regions, the Exposed Service Operation can be set up either way, bytes or String. If the communication is in terms of bytes, these are converted by applying the specified encoding to the entire message.
A Message Content Factory also allows precise control over what to do if bytes are being used to communicate with the embedding application, but incoming messages cannot be decoded or outgoing messages cannot be encoded (for examples of why this might arise, see the tables in the introduction).

6 Detailed API

There are a number of basic tasks that the API can perform:

- Get Runtime Definitions
- Make a Mom Object from its Definition and some data i.e. parse the data.
- Create a new Mom Object by performing a Mapping
- Format a Mom for output

and a number of more advanced tasks some of which are:

- Identify a Message from it's content
- Perform an Enrichment
- Invoke a Condition
- Apply a Rule Set (i.e. apply Validation or Enrichment to a Message)
- Checking data structure and error reporting

6.1 Getting Runtime Definitions

As mentioned in section 2.3 before any Transformer definitions can be used they have to be loaded. There are a number of ways to do this. A general approach is to use the Transformer path to the definition to directly load the ConfigObject. For example, for a Message called "CSV1" in the Message Definition Group "MyCSV", the following code could be used to build up the directory structure using a String array to load the definition:

```
ConfigObject co = project.getRunTimeObject(new String[] {
                                       "MessageDefinitionGroups",
                                       "MyCsv",
                                       "Messages",
                                       "CSV1"});
```

As an aid to the programmer some of the common directory names such as "Messages" have been created as statics in the GroupName class. If the user knows the type of object being created the ConfigObject can be cast to the desired type:

```
MessageDefinition md = (MessageDefinition) project.getRunTimeObject(
                             new String[] { GroupName.MESSAGE_DEFINITION_GROUPS.toString(),
                                            "MyCsv",
                                            GroupName.MAPPING_DEFINITION_GROUPS.toString(),
                                            "CSV1"});
```

Alternatively, the directory structure can be used to obtain the ConfigFile and the ConfigObject can be loaded from that:

```
ConfigFile cf = project.getConfigFile("MessageDefinitionGroups/MyCsv/Messages/CSV1");
ConfigObject co = project.getRunTimeObject(cf);
```

For most definitions static convenience methods have been provided to make the loading of definitions easier (as mentioned in section **Error! Reference source not found.**). The user provides the name and the group of the thing being loaded. The static methods appear in each of the classes that require loading. E.g.

```
MessageDefinition md = MessageDefinition.getMessageDefinitionRunTime(
                                            project,
                                            groupName,
                                            MessageName);
```

Or

```
MappingDefinition mapd = MappingDefinition.getMappingDefinitionRunTime(
                                       project,
                                       Project.MAP_GROUPS,mappingGroup,
                                       mappingName);
```

Finally for Message Definitions, if a Message Type Identifier (MTI) has been configured for the group then the definition can be loaded by supplying the message data. In this example the `messageData` variable can be a String or byte array.

```
MessageDefinitionGroup mdg = MessageDefinitionGroup.getMessageDefinitionGroupRunTime(
                                       project, groupName);
MessageDefinition md1 = mdg.getMessageDefinitionRunTimeFromContent(messageData);
```

6.2 Parsing Data - Creating MomNodes

A MomNode is a structure containing a Dad definition and some data. To create one, both a Dad object (such as a Message Definition, a Component or a Type) and optionally some data are required:

```
MessageDefinition md = ...;
String messageData = ...;
MomNode mom = MomNode.makeMom(md,messageData);
```

To create an empty Mom structure just the Dad definition is required:

```
MomNode mom = MomNode.makeMom(md);
```

Once the Mom Node has been produced it can be manipulated with some other simple API calls. MomNode is the base class for two sub-classes – ComplexNode and SimpleNode. A ComplexNode is a container for other MomNodes so child nodes can be added or removed from it. SimpleNodes actually store the individual data items and can be viewed as leaves in a MomNode tree. SimpleNodes have values which can be represented as typed values, relating to different Business Object Types, or as Strings.

To set the content of a simple node the following code could be used:

```
SimpleNode sn = …
sn.setStringValue("AAA");
```

ComplexNodes contain children that are combinations of other MomNodes (both Simple and Complex). It is possible to find/add/delete children of complex nodes. The code below shows a method that could be used to set all the children in a MomNode tree to a particular String value. Note that if any of the SimpleNodes in the structure were using Business Object Types of anything other than String then the following code could cause an error.

```
public void setAllContent(MomNode mn,String newContent) throws MomNodeException {
  if (mn instanceof ComplexNode) {
   ComplexNode cn = (ComplexNode) mn;
```

```
          Enumeration en = cn.childEnumeration();
          while (en.hasMoreElements()) {
            MomNode child = (MomNode)en.nextElement();
            setAllContent(child,newContent);
          }
        } else if (mn instanceof SimpleNode) {
          SimpleNode sn = (SimpleNode)mn;
          sn.setStringValue(newContent);
        }
      }
```

Note that if the `newContent` String supplied did not match the business object type or constraints defined for the node, a MomNodeException could be thrown.

An ExceptionMask is a token passed to various methods to indicate categories of MomNodeException that should be suppressed or ignored.

When parsing a message, an ExceptionMask can be passed to indicate that structure exceptions such as (exceeding the maximum permitted repetition count of a node) should not terminate the parse operation. When assigning a String value to a node, an ExceptionMask can be passed to indicate that a string that does not represent valid content should not cause an exception to be thrown. Instead, the exception can be retrieved later using the node's getExceptions method.

E.g.

```
sn.setStringValue(newContent, ExceptionMask.MaskEverything);
```

Or

```
sn.setStringValue(newContent, ExceptionMask.MaskNothing);
```

6.3 Checking Message Validity

Message validity, or rather message invalidity, is represented in Transformer by the attachment of one or MomNodeException objects to invalid nodes in a MOM structure. Each MomNodeException has a “rule set name” which allows the API to distinguish between different kinds of validation failure. There are some inbuilt ruleset names, relating to special subclasses of MomNodeException, and also the ability for additional rulesets to be defined via the GUI design tool, and associated with user-defined validation rules.

MomNodeExceptions can get attached at any of the following times:

- when a message is parsed
- when an API operation manipulates adds or removes nodes in a MOM structure or sets values of simple nodes
- when a message is created or enriched via mapping or rule
- when a validation ruleset is invoked

The API allows the validity of a message to be checked, so that (for example) appropriate routing decisions can be taken.

```
MomNode mn = …
if (mn.hasExceptions()) {
  String errorString = mn.getExceptionReport();
  ...
}
```

The hasExceptions and getExceptionReport methods can be applied to named rules sets as a means of filtering the reported errors.

See section 6.8 on how to apply Validation Rules.

6.4 Calling Mappings

A mapping can be called in one of two ways. All mappings can be called using a MomNodeBasket which is used to wrap up all the inputs and the outputs of a mapping. Simple one-to-one mappings can use an alternative, simpler mechanism which will perform the mapping in a single call.

6.4.1 Calling Simple Mappings

A simple one-to-one mapping (that does not use a simple BOT as the input or the output) can be called once the mapping definition has been loaded and the input MomNode constructed. The makeNodeViaMap method is used and the return value from the method is the MomNode representing the output of the mapping.

```
MomNode mom = ...
MappingDefintion mapd = ...
MomNode output = mom.makeNodeViaMap(mapd);
```

6.4.2 Calling Complex Mappings - MomNodeBaskets

For complex mappings (i.e. those involving multiple inputs or outputs, repeating inputs or outputs or simple BOTS as inputs or outputs) a different approach is required. The user must create a MomNodeBasket to hold all of the inputs and the result of the mapping is another MomNodeBasket that holds all of the outputs.

An input Basket can be created by making the appropriate call on the Mapping definition:

```
MomNodeBasket basketIn = mapd.makeInputBasket();
```

Once the basket has been created the input mom nodes can be added to it – notice that the position of the inputs in the basket must be specified.

```
basketIn.putNode(mom1,0);
basketIn.putNode(mom2,1);
```

If the basket requires repeating inputs then two indices are required – the first to specify which input it is and the second to specify the repeat index:

```
basketIn.addRepeatingNode(mom1,0,0);
basketIn.addRepeatingNode(mom2,0,1);
```

To call the mapping the following code should be used. The output MomNodes can be retrieved from the basket:

```
MomNodeBasket basketOut = mapd.makeOutputBasket(basketIn);
MomNode firstOuput = basketOut.getNode(0);
MomNode secondOutput = basketOut.getNode(1);
```

Alternatively, a RuntimeContext may be passed into the Mapping call:

```
RuntimeContext context = ...
MomNodeBasket basketOut = mapd.makeOutputBasket(basketIn,context);
```

If the number or type of MomNodes in the basket is not known some code is required to pick the data from the basket - the following code snippet is an example of what can be done:

```
for (int i = 0; i < outputBasket.getDefinitionCount(); i++) {
   if (outputBasket.isNodeRepeating(i)) {
      for (int j = 0; j < outputBasket.getNumberOfRepeats(i); j++) {
         MomNode mn1 = outputBasket.getRepeatingNode(i, j);
         if (mn1 != null) {
         //Do Something with the output Mom
            …
         }
      }
   } else {
      MomNode mn1 = outputBasket.getNode(i);
      if (mn1 != null) {
     //Do Something with the output Mom
         …
      }
   }
}
```

6.5 Formatting Outputs

A MomNode can be serialised into a string via the format command:

```
String output = mom.format();
```

To create the XML string form the following call is made. The first example displays the XML in a "pretty format" whilst the second serialises the output onto a single line.

```
output = mom.formatAsXML(MomNode.SERIALIZE_FORMATTED);

output = mom.formatAsXML(MomNode.SERIALIZE_UNFORMATTED);
```

There are format methods that take Writers or OutputStreams if the output is required in other forms:

```
void format(OutputStream stream, short format)
void format(Writer writer, short format)
```

6.6 Calling Enrichments

Enrichment Definitions are called in a similar way to mappings. After the Enrichment has been loaded and an Input Basket created and filled the following can be used:

```
NamedEnrichmentDefinition enrichment = …
enrichment.enrich(inputBasket);
```

Alternatively, a RuntimeContext may be passed into the Enrichment call:

```
enrichment.enrich(inputBasket,context);
```

6.7 Calling Condition Definitions

Condition Definitions are called in a similar way to Enrichments – the Condition must be loaded, the Input Basket must be created and filled and then the following can be called:

```
ConditionDefinition cd = …
boolean result = cd.evaluate(basket);
```

Alternatively, a RuntimeContext may be passed into the Condition:

```
boolean result = cd.evaluate(basket,context);
```

6.8 Applying Rules or Rule Sets

Whole Rule Sets can be applied to MomNodes by first loading the definitions and then making the appropriate call which for Enrichment Rule Sets is:

```
ers.enrich(mom);
```

And for Validation Rule Sets is:

```
boolean result = vrs.validate(mom);
```

If the validation fails any error messages will be attached to the input MomNode and can be retrieved using:

```
String errorString = mom.getExceptionReport();
```

Again, in either case a RuntimeContext can be added to the call:

```
boolean result = vrs.validate(mom, context);
```

6.9 Example Code

Some examples of code for different scenarios is provided.

6.9.1 Format a Non-XML Message as XML

This class reads in a simple CSV (i.e. non-XML) message and reformats it to and then prints its XML equivalent message.

```java
package apitest;

import com.tracegroup.transformer.configfiles.*;
import com.tracegroup.transformer.dad.*;
import com.tracegroup.transformer.mom.*;

/**
 * This class formats a simple Comma Seperated File to its XML equivalent.
 * @author Keith Donovan
 */
public class FormatCsvAsXML {

  public FormatCsvAsXML() {
    try {

      //The message data. This could be picked up from a file or passed into a MDB
      //or got off a data bus.
      //Note that in this example the first field gives the type of the message
      String messageData = "CSV1,Hello,100";

      //The project location and the group name
      //These could be hard coded or got out of JNDI
      //Creating a new project is a big overhead - this is not something
      //that should be done for each message. The project once constructed
      //should be cached in some way.
      Project project = new Project("C:/transformer/apiTest/ApiTest.tpj");
      String groupName = "MyCsv";

      //Given the group name get the MDG object.
        MessageDefinitionGroup mdg =
            MessageDefinitionGroup.getMessageDefinitionGroupRunTime(project, groupName);

      //You know the group, a message Type Identifier (MTI) has been set up
      //so you should be able to get a message definition object if the data is OK
      //Note that if you know the name of the message definition you can do this more quickly
      //with a call like this:
      // MessageDefinition md =
      //     MessageDefinition.getMessageDefinitionRunTime(project,groupName,MessageName);
      MessageDefinition md =
          mdg.getMessageDefinitionRunTimeFromContent(messageData);

      //Construct a Mom Structure from the data
      MomNode mom = MomNode.makeMom(md,messageData);

      //Format the Mom as XML
      System.out.println(mom.formatAsXML(MomNode.SERIALIZE_FORMATTED));

    }
    catch (ConfigFileStructureException cfse) {
      //Occurs if a config file is incorrectly structured
      //thrown by new Project,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfse.printStackTrace();
    }
    catch (ConfigFileContentException cfce) {
      //Occurs if one of the config files has invalid configuration details
      //thrown by MessageDefinitionGroup.getMessageDefinitionGroupRunTime
      //and mdg.getMessageDefinitionRunTimeFromContent
      cfce.printStackTrace();
    }
    catch(UnrecognizedMessageException ume) {
      //Caused by the MessageTypeIdentifier failing to ID the message
      //thrown by the mdg.getMessageDefinitionRunTimeFromContent call
      ume.printStackTrace();
    }
    catch(ParseException pe) {
      //Caused by the parser failing to break up the data during the MomNode.makeMom call
      pe.printStackTrace();
    }
    catch(WriteMessageContentException wmce) {
      //Caused by mom.formatAsXML failing
      wmce.printStackTrace();
    }
  }

  public static void main(String[] args) {
    FormatCsvAsXML csv2Xml = new FormatCsvAsXML();

  }

}
```

6.9.2 Validate a Message with a Validation Rule Set

This class reads in two CSV messages, one is valid and one is invalid. A simplified Validation Rule Set is applied to them and the results from the validations are printed out.

```
package apitest;

import com.tracegroup.transformer.configfiles.*;
import com.tracegroup.transformer.dad.*;
import com.tracegroup.transformer.mom.*;

/**
 * This class validates a pair of CSV messages by applying a validation rule set to them.
 * @author Keith Donovan
 */
public class ValidateCsv {

  //Set up some dummy message data
  String _messageDataGood = "CSV1,aaa,123";

  //This data is wrong. If the 56A field is present then there should be a
  //57a field present as well.
  String _messageDataBad = "CSV1,aaa";



  public ValidateCsv() {
    try {
      //The project location and the group name
      //These could be hard coded or got out of JNDI
      //Creating a new project is a big overhead - this is not something
      //that should be done for each message. The project once constructed
      //should be cached in some way.
      Project project = new Project("C:/transformer/apiTest/ApiTest.tpj");
      String groupName = "MyCsv";
      //Get the name of the validation rule set
      String ruleSetName = "ValCsv";

      //Given the group name get the MDG object.
      MessageDefinitionGroup mdg =
          MessageDefinitionGroup.getMessageDefinitionGroupRunTime(project,groupName);

      //You know the group, a message Type Identifier (MTI) has been set up
      //so you should be able to get a message definition object if the data is OK
      //Note that if you know the name of the message definition you can do this more quickly
      //with a call like this:
      // MessageDefinition md =
      //    MessageDefinition.getMessageDefinitionRunTime(project,groupName,MessageName);
      MessageDefinition md = mdg.getMessageDefinitionRunTimeFromContent(_messageDataGood);

      //Get a Rule Set Object
      ValidationRuleSet vrs =
          ValidationRuleSet.getValidationRuleSetRunTime(project,ruleSetName);

     //Load the rules for this message definition
     vrs.loadRuleSet(md);

     //Construct a Mom Structure from the good data
     MomNode mom1 = MomNode.makeMom(md,_messageDataGood);
     //Then validate it - should be OK
     validate(vrs,mom1,"1");

     //Construct a Mom Structure from the bad data
     MomNode mom2 = MomNode.makeMom(md,_messageDataBad);
     //Then validate it - should cause problems
     validate(vrs,mom2,"2");

    }
    catch (ConfigFileStructureException cfse) {
      //Occurs if a config file is incorrectly structured
      //thrown by new Project, ValidationRuleSet.getValidationRuleSetRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfse.printStackTrace();
    }
    catch (ConfigFileContentException cfce) {
      //Occurs if one of the config files has invalid configuration details
      //thrown by ValidationRuleSet.getValidationRuleSetRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfce.printStackTrace();
    }
    catch(UnrecognizedMessageException ume) {
      //Caused by the MessageTypeIdentifier failing to ID the message
      //thrown by the mdg.getMessageDefinitionRunTimeFromContent call
      ume.printStackTrace();
    }
    catch(ParseException pe) {
      //Caused by the parser failing to break up the data during the MomNode.makeMom call
      pe.printStackTrace();
    }
    catch(ActionException ae) {
      //Caused by an error during the validation
      ae.printStackTrace();
    }

  }


  private void validate(ValidationRuleSet vrs, MomNode mom, String messageID)
      throws ActionException {
    boolean result = vrs.validate(mom);
    if (result) {
      System.out.println("Message " + messageID + " is  Valid");
    } else {
      //Get the exception report passing in the validation rule set name
      //This ensures only the errors for that validation rule set are reported
      System.out.println("Message " + messageID + " is  Invalid.");
      System.out.println(mom.getExceptionReport(vrs.getName().toString()));
    }
  }

  public static void main(String[] args) {
    ValidateCsv vs = new ValidateCsv();
  }

}
```

6.9.3 Enrich a Message with an Enrichment Rule Set

An XML message is read in and enriched by applying a simple Enrichment Rule Set. The updated message is then printed out.

```java
package apitest;

import com.tracegroup.transformer.configfiles.*;
import com.tracegroup.transformer.dad.*;
import com.tracegroup.transformer.mom.*;

/**
 * This class enriches a simple XML message. The message is enriched by changing
 * the value of the first (AAA) field.
 * @author Keith Donovan
 */
public class EnrichXml {
  public EnrichXml() {
    try {

      //Simple XML data
      String messageData = "<AAA>" +
          "<One>Eins</One>" +
          "<Two>Zwei</Two>" +
          "<Three>Drei</Three>" +
          "</AAA>";

      //The project location and the group name
      //These could be hard coded or got out of JNDI
      //Creating a new project is a big overhead - this is not something
      //that should be done for each message. The project once constructed
      //should be cached in some way.
      Project project = new Project("C:/transformer/apiTest/ApiTest.tpj");
      String groupName = "MyXml";
      //The name of the rule set we are going to apply
      String ruleSetName = "MyXmlEnrichment";

      //Given the group name get the MDG object.
      MessageDefinitionGroup mdg =
          MessageDefinitionGroup.getMessageDefinitionGroupRunTime(project,groupName);

      //You know the group, a message Type Identifier (MTI) has been set up
      //so you should be able to get a message definition object if the data is OK
      //Note that if you know the name of the message definition you can do this more quickly
      //with a call like this:
      // MessageDefinition md =
      //    MessageDefinition.getMessageDefinitionRunTime(project,groupName,MessageName);
      MessageDefinition md = mdg.getMessageDefinitionRunTimeFromContent(messageData);

      //Get a Rule Set Object
      EnrichmentRuleSet ers =
          EnrichmentRuleSet.getEnrichmentRuleSetRunTime(project,ruleSetName);

      //Load all the rules for this message
      ers.loadRuleSet(md);

      //Construct a Mom Structure from the data
      MomNode mom = MomNode.makeMom(md,messageData);

      //Call the Enrichment
      ers.enrich(mom);

      //Print Out the results - they should be different from what we started with
      System.out.println(mom.format(MomNode.SERIALIZE_FORMATTED));
    }
    catch (ConfigFileStructureException cfse) {
      //Occurs if a config file is incorrectly structured
      //thrown by new Project, EnrichmentRuleSet.getEnrichmentRuleSetRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfse.printStackTrace();
    }
    catch (ConfigFileContentException cfce) {
      //Occurs if one of the config files has invalid configuration details
      //thrown by EnrichmentRuleSet.getEnrichmentRuleSetRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfce.printStackTrace();
    }
    catch(UnrecognizedMessageException ume) {
      //Caused by the MessageTypeIdentifier failing to ID the message
      //thrown by the mdg.getMessageDefinitionRunTimeFromContent call
      ume.printStackTrace();
    }
    catch(ParseException pe) {
      //Caused by the parser failing to break up the data during the MomNode.makeMom call
      pe.printStackTrace();
    }
    catch(ActionException ae) {
      //Caused by an error during the enrichment
      ae.printStackTrace();
    }
    catch(WriteMessageContentException wmce) {
      //Caused by mom.format failing
      wmce.printStackTrace();
    }

  }


  public static void main(String[] args) {
    EnrichXml vs = new EnrichXml();
  }

}
```

6.9.4 Perform a Simple Mapping

A simple CSV (i.e. non-XML) message is mapped to a different XML message. The new XML message is then printed out.

```
package apitest;

import com.tracegroup.transformer.configfiles.*;
import com.tracegroup.transformer.dad.*;
import com.tracegroup.transformer.mom.*;

/**
 * This clas performs a simple one-to-one mapping.
 * @author Keith Donovan
 */
public class MapCsv {
  public MapCsv() {

    try {

      //The message data. This could be picked up from a file or passed into a MDB
      //or got off a data bus.
      //Note that in this example the first field gives the type of the message
      String messageData = "CSV1,Hello,100";

      //The project location and the group name
      //These could be hard coded or got out of JNDI
      //Creating a new project is a big overhead - this is not something
      //that should be done for each message. The project once constructed
      //should be cached in some way.
      Project project = new Project("C:/transformer/apiTest/ApiTest.tpj");
      String groupName = "MyCsv";
      //The mapping information
      String mappingGroup = "MyCsv2MyXml";
      String mappingName = "Csv1ToAAA";

      //Given the group name get the MDG object.
      MessageDefinitionGroup mdg =
          MessageDefinitionGroup.getMessageDefinitionGroupRunTime(project,groupName);

      //You know the group, a message Type Identifier (MTI) has been set up
      //so you should be able to get a message definition object if the data is OK
      //Note that if you know the name of the message definition you can do this more quickly
      //with a call like this:
      // MessageDefinition md =
      //     MessageDefinition.getMessageDefinitionRunTime(project,groupName,MessageName);
      MessageDefinition md = mdg.getMessageDefinitionRunTimeFromContent(messageData);

      //Construct a Mom Structure from the data
      MomNode mom = MomNode.makeMom(md, messageData);

      //Get the mapping definition
      MappingDefinition mapd = MappingDefinition.getMappingDefinitionRunTime(project,
          GroupName.MAPPING_DEFINITION_GROUPS.toString(),mappingGroup,mappingName);

      //Perform the mapping and produce an output Mom Node
      MomNode output = mom.makeNodeViaMap(mapd);

      //Format the Output Mom
      System.out.println(output.format(MomNode.SERIALIZE_FORMATTED));

    }
    catch (ConfigFileStructureException cfse) {
      //Occurs if a config file is incorrectly structured
      //thrown by new Project, MappingDefinition.getMappingDefinitionRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfse.printStackTrace();
    }
    catch (ConfigFileContentException cfce) {
      //Occurs if one of the config files has invalid configuration details
      //thrown by MappingDefinition.getMappingDefinitionRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfce.printStackTrace();
    }
    catch (UnrecognizedMessageException ume) {
      //Caused by the MessageTypeIdentifier failing to ID the message
      //thrown by the mdg.getMessageDefinitionRunTimeFromContent call
      ume.printStackTrace();
    }
    catch (ParseException pe) {
      //Caused by the parser failing to break up the data during the MomNode.makeMom call
      pe.printStackTrace();
    }
    catch (WriteMessageContentException wmce) {
      //Caused by mom.format failing
      wmce.printStackTrace();
    }
    catch(ActionException ae) {
      //Caused by an error during the mapping
      ae.printStackTrace();
    }

  }

  public static void main(String[] args) {
    MapCsv mc = new MapCsv();
  }

}
```

6.9.5 Use a Condition Definition to Decide on Which Mapping to Use

This class reads in a CSV message and then applies a Condition Definition to the message, depending on the result of the condition a choice is made as to which mapping should be applied to the message. After the mapping has been applied the new XML message is printed out.

```
package apitest;

import com.tracegroup.transformer.configfiles.*;
import com.tracegroup.transformer.dad.*;
import com.tracegroup.transformer.mom.*;

/**
 * This class uses a Condition Definition to determine which Mapping to perform.
 * <P>
 * The condition checks the numeric value of the third field. If the value is less
 * than 100 then the Csv1ToAAA Mapping is performed otherwise the Csv1ToBBB
 * Mapping is done.
 * @author Keith Donovan
 */
public class ConditionalMapping {
  public ConditionalMapping() {
    try {

      //The message data. This could be picked up from a file or passed into a MDB
      //or got off a data bus.
      //Note that in this example the first field gives the type of the message
      String messageData = "CSV1,Hello,100";

      //The project location and the group name
      //These could be hard coded or got out of JNDI
      //Creating a new project is a big overhead - this is not something
      //that should be done for each message. The project once constructed
      //should be cached in some way.
      Project project = new Project("C:/transformer/apiTest/ApiTest.tpj");
      String groupName = "MyCsv";
      //The mapping information
      String mappingGroup = "MyCsv2MyXml";
      String mappingName1 = "Csv1ToAAA";
      String mappingName2 = "Csv1ToBBB";
      String conditionDef = "IsCLessThan100";

      //Given the group name get the MDG object.
      MessageDefinitionGroup mdg =
          MessageDefinitionGroup.getMessageDefinitionGroupRunTime(project,groupName);

      //You know the group, a message Type Identifier (MTI) has been set up
      //so you should be able to get a message definition object if the data is OK
      //Note that if you know the name of the message definition you can do this more quickly
      //with a call like this:
      // MessageDefinition md =
      //    MessageDefinition.getMessageDefinitionRunTime(project,groupName,MessageName);
      MessageDefinition md = mdg.getMessageDefinitionRunTimeFromContent(messageData);

      //Construct a Mom Structure from the data
      MomNode mom = MomNode.makeMom(md, messageData);

      //We are going to call a Condition Definition which checks the third field
      //If the value in the third field is less than 100 (it returns true) and
      //then it calls mapping 1 or if its 100 or greater it calls mapping 2

      ConditionDefinition cd = NamedConditionDefinition.
          getNamedConditionDefinitionRunTime(
          project, GroupName.MESSAGE_DEFINITION_GROUPS.toString(), groupName, conditionDef);

      //Conditions can take multiple inputs
      //Therefore the input message structures must be placed into a basket
      //Make the basket structure by getting it from the condition
      MomNodeBasket basket = cd.makeInputBasket();
      //Fill the basket with the mom
      basket.putNode(mom,0);

      //Evaluate the condition
      boolean result = cd.evaluate(basket);


      MappingDefinition mapd;
      //Get the appropriate mapping definition
      if (result) {
        mapd = MappingDefinition.getMappingDefinitionRunTime(project,
            GroupName.MAPPING_DEFINITION_GROUPS.toString(),mappingGroup,mappingName1);
      } else {
        mapd = MappingDefinition.getMappingDefinitionRunTime(project,
            GroupName.MAPPING_DEFINITION_GROUPS.toString(),mappingGroup,mappingName2);
      }

      //Perform the mapping and produce an output Mom Node
      MomNode output = mom.makeNodeViaMap(mapd);

      //Format the Output Mom
      System.out.println(output.format(MomNode.SERIALIZE_FORMATTED));

    }
    catch (ConfigFileStructureException cfse) {
      //Occurs if a config file is incorrectly structured
      //thrown by new Project, MappingDefinition.getMappingDefinitionRunTime,
      //NamedConditionDefinition.getNamedConditionDefinitionTypeRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfse.printStackTrace();
    }
    catch (ConfigFileContentException cfce) {
      //Occurs if one of the config files has invalid configuration details
      //thrown by MappingDefinition.getMappingDefinitionRunTime,
      //NamedConditionDefinition.getNamedConditionDefinitionTypeRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfce.printStackTrace();
    }
    catch (UnrecognizedMessageException ume) {
      //Caused by the MessageTypeIdentifier failing to ID the message
      //thrown by the mdg.getMessageDefinitionRunTimeFromContent call
      ume.printStackTrace();
    }
    catch (ParseException pe) {
      //Caused by the parser failing to break up the data during the MomNode.makeMom call
      pe.printStackTrace();
    }
    catch (WriteMessageContentException wmce) {
      //Caused by mom.format failing
      wmce.printStackTrace();
    }
    catch(ActionException ae) {
      //Caused by an error during the mapping or the condition
      ae.printStackTrace();
    }

  }

  public static void main(String[] args) {
    ConditionalMapping cm = new ConditionalMapping();
  }

}
```

6.9.6 Perform a Many to One Mapping

Two different CSV messages are read in and are combined, by the use of a single mapping, to one output XML message. The new XML message is then printed.

```
package apitest;

import com.tracegroup.transformer.configfiles.*;
import com.tracegroup.transformer.dad.*;
import com.tracegroup.transformer.mom.*;

/**
 * This class performs a many-to-one mapping. It copies data from CSV1 and CSV2
 * type messages into an XML message.
 * @author Keith Donovan
 */
public class ManyToOne {
  public ManyToOne() {

    try {

      //The message data. This could be picked up from a file or passed into a MDB
      //or got off a data bus.
      //Note that in this example the first fields give the type of the message
      String messageDataOne = "CSV1,Hello,100";
      String messageDataTwo = "CSV2,Bye,99";

      //The project location and the group name
      //These could be hard coded or got out of JNDI
      //Creating a new project is a big overhead - this is not something
      //that should be done for each message. The project once constructed
      //should be cached in some way.
      Project project = new Project("C:/transformer/apiTest/ApiTest.tpj");
      String groupName = "MyCsv";
      //The mapping information
      String mappingGroup = "MyCsv2MyXml";
      String mappingName = "Csv1Csv2ToCCC";

      //Given the group name get the MDG object.
      MessageDefinitionGroup mdg =
          MessageDefinitionGroup.getMessageDefinitionGroupRunTime(project,groupName);

      //You know the group, a message Type Identifier (MTI) has been set up
      //so you should be able to get a message definition object if the data is OK
      //Note that if you know the name of the message definitions you can do this more quickly
      //with a call like this:
      // MessageDefinition md1 =
      //    MessageDefinition.getMessageDefinitionRunTime(project,groupName,MessageName1);
      MessageDefinition md1 = mdg.getMessageDefinitionRunTimeFromContent(messageDataOne);
      MessageDefinition md2 = mdg.getMessageDefinitionRunTimeFromContent(messageDataTwo);

      //Construct the Mom Structures from the data
      MomNode mom1 = MomNode.makeMom(md1, messageDataOne);
      MomNode mom2 = MomNode.makeMom(md2, messageDataTwo);
      //Get the mapping definition
      MappingDefinition mapd = MappingDefinition.getMappingDefinitionRunTime(project,
          GroupName.MAPPING_DEFINITION_GROUPS.toString(),mappingGroup,mappingName);

      //We have multiple inputs so we need to bundle them together by
      //putting them in a basket.
      //Create the input basket by getting it from the mapping.
      MomNodeBasket basketIn = mapd.makeInputBasket();
      //Add the two mom nodes to the basket
      basketIn.putNode(mom1,0);
      basketIn.putNode(mom2,1);

      //Perform the mapping by making the call on the mapping definition
      //Note that this is different to the simple mapping case
      //A basket is returned from the mapping which we have to then pull apart
      MomNodeBasket basketOut = mapd.makeOutputBasket(basketIn);

      //We know there is only one output from the mapping so just get the first MomNode
      MomNode output = basketOut.getNode(0);

      //Format the Output Mom
      System.out.println(output.format(MomNode.SERIALIZE_FORMATTED));

    }
    catch (ConfigFileStructureException cfse) {
      //Occurs if a config file is incorrectly structured
      //thrown by new Project, MappingDefinition.getMappingDefinitionRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfse.printStackTrace();
    }
    catch (ConfigFileContentException cfce) {
      //Occurs if one of the config files has invalid configuration details
      //thrown by MappingDefinition.getMappingDefinitionRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfce.printStackTrace();
    }
    catch (UnrecognizedMessageException ume) {
      //Caused by the MessageTypeIdentifier failing to ID the message
      //thrown by the mdg.getMessageDefinitionRunTimeFromContent call
      ume.printStackTrace();
    }
    catch (ParseException pe) {
      //Caused by the parser failing to break up the data during the MomNode.makeMom call
      pe.printStackTrace();
    }
    catch (WriteMessageContentException wmce) {
      //Caused by mom.format failing
      wmce.printStackTrace();
    }
    catch(ActionException ae) {
      //Caused by an error during the mapping
      ae.printStackTrace();
    }

  }

  public static void main(String[] args) {
    ManyToOne mc = new ManyToOne();
  }

}
```

6.9.7 Perform a Many to One Mapping with Repeats

Two similar CSV messages are read in and are combined, by the use of a single mapping, to one output XML message. The new XML message is then printed.

```
package apitest;

import com.tracegroup.transformer.configfiles.*;
import com.tracegroup.transformer.dad.*;
import com.tracegroup.transformer.mom.*;

/**
 * This class calls a mapping which is expecting a repeating input.
 * The repeating inputs are mapped to a single output on the right hand side.
 * @author Keith Donovan
 */
public class ManyToOneRepeating {
  public ManyToOneRepeating() {

    try {

      //The message data. This could be picked up from a file or passed into a MDB
      //or got off a data bus.
      //Note that in this example the first fields give the type of the message
      String messageDataOne = "CSV1,Hello,100";
      String messageDataTwo = "CSV1,Bye,99";

      //The project location and the group name
      //These could be hard coded or got out of JNDI
      //Creating a new project is a big overhead - this is not something
      //that should be done for each message. The project once constructed
      //should be cached in some way.
      Project project = new Project("C:/transformer/apiTest/ApiTest.tpj");
      String groupName = "MyCsv";
      //The mapping information
      String mappingGroup = "MyCsv2MyXml";
      String mappingName = "Csv1RepeatingToCCC";

      //Given the group name get the MDG object.
      MessageDefinitionGroup mdg =
          MessageDefinitionGroup.getMessageDefinitionGroupRunTime(project,groupName);

      //You know the group, a message Type Identifier (MTI) has been set up
      //so you should be able to get a message definition object if the data is OK
      //Note that if you know the name of the message definition you can do this more quickly
      //with a call like this:
      // MessageDefinition md =
      //    MessageDefinition.getMessageDefinitionRunTime(project,groupName,MessageName);
      // The message types for the two inputs are the same so we only need to do this once.
      MessageDefinition md = mdg.getMessageDefinitionRunTimeFromContent(messageDataOne);

      //Construct the Mom Structures from the data
      //We know in this case there are only two iterations
      //but this could use a container and a loop if there are an unknown
      //number of iterations
      MomNode mom1 = MomNode.makeMom(md, messageDataOne);
      MomNode mom2 = MomNode.makeMom(md, messageDataTwo);

      //Get the mapping definition
      MappingDefinition mapd = MappingDefinition.getMappingDefinitionRunTime(project,
          GroupName.MAPPING_DEFINITION_GROUPS.toString(),mappingGroup,mappingName);


      //We have multiple inputs so we need to bundle them together by
      //putting them in a basket.
      //Create the input basket by getting it from the mapping.
      MomNodeBasket basketIn = mapd.makeInputBasket();
      //Add the two mom nodes to the basket
      //Again this could be done inside a loop
      basketIn.addRepeatingNode(mom1,0,0);
      basketIn.addRepeatingNode(mom2,0,1);

      //Perform the mapping by making the call on the mapping definition
      //Note that this is different to the simple mapping case
      //A basket is returned from the mapping which we have to then pull apart
      MomNodeBasket basketOut = mapd.makeOutputBasket(basketIn);

      //We know there is only one output from the mapping so just get the first MomNode
      MomNode output = basketOut.getNode(0);

      //Format the Output Mom
      System.out.println(output.format(MomNode.SERIALIZE_FORMATTED));

    }
    catch (ConfigFileStructureException cfse) {
      //Occurs if a config file is incorrectly structured
      //thrown by new Project, MappingDefinition.getMappingDefinitionRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfse.printStackTrace();
    }
    catch (ConfigFileContentException cfce) {
      //Occurs if one of the config files has invalid configuration details
      //thrown by MappingDefinition.getMappingDefinitionRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfce.printStackTrace();
    }
    catch (UnrecognizedMessageException ume) {
      //Caused by the MessageTypeIdentifier failing to ID the message
      //thrown by the mdg.getMessageDefinitionRunTimeFromContent call
      ume.printStackTrace();
    }
    catch (ParseException pe) {
      //Caused by the parser failing to break up the data during the MomNode.makeMom call
      pe.printStackTrace();
    }
    catch (WriteMessageContentException wmce) {
      //Caused by mom.format failing
      wmce.printStackTrace();
    }
    catch(ActionException ae) {
      //Caused by an error during the mapping
      ae.printStackTrace();
    }

  }

  public static void main(String[] args) {
    ManyToOneRepeating mc = new ManyToOneRepeating();
  }

}
```

6.9.8 Perform a One to Many Mapping

A single CSV message is read in to a single mapping, and mapped into two different XML messages. The new messages are then printed out.

```
package apitest;

import com.tracegroup.transformer.configfiles.*;
import com.tracegroup.transformer.dad.*;
import com.tracegroup.transformer.mom.*;
```

```java
/**
 * This class maps a simple CVS structure to two separate XML messages.
 * @author Keith Donovan
 */
public class OneToMany {
  public OneToMany() {

    try {

      //The message data. This could be picked up from a file or passed into a MDB
      //or got off a data bus.
      //Note that in this example the first field gives the type of the message
      String messageDataOne = "CSV3,Felix,100,Flossy,Daisy,Pinky";


      //The project location and the group name
      //These could be hard coded or got out of JNDI
      //Creating a new project is a big overhead - this is not something
      //that should be done for each message. The project once constructed
      //should be cached in some way.
      Project project = new Project("C:/transformer/apiTest/ApiTest.tpj");
      String groupName = "MyCsv";
      //The mapping information
      String mappingGroup = "MyCsv2MyXml";
      String mappingName = "Csv3ToAAABBB";

      //Given the group name get the MDG object.
      MessageDefinitionGroup mdg =
          MessageDefinitionGroup.getMessageDefinitionGroupRunTime(project,groupName);

      //You know the group, a message Type Identifier (MTI) has been set up
      //so you should be able to get a message definition object if the data is OK
      //Note that if you know the name of the message definition you can do this more quickly
      //with a call like this:
      // MessageDefinition md =
      //    MessageDefinition.getMessageDefinitionRunTime(project,groupName,MessageName);
      MessageDefinition md1 = mdg.getMessageDefinitionRunTimeFromContent(messageDataOne);


      //Construct the Mom Structures from the data
      MomNode mom1 = MomNode.makeMom(md1, messageDataOne);

      //Get the mapping definition
      MappingDefinition mapd =
          MappingDefinition.getMappingDefinitionRunTime(project,
          GroupName.MAPPING_DEFINITION_GROUPS.toString(), mappingGroup, mappingName);


      //We have complex mapping so we need to bundle the inputs together by
      //putting them in a basket.
      //Create the input basket by getting it from the mapping.
      MomNodeBasket basketIn = mapd.makeInputBasket();
      //Add the mom node to the basket
      basketIn.putNode(mom1,0);

      //Perform the mapping by making the call on the mapping definition
      //Note that this is different to the simple mapping case
      //A basket is returned from the mapping which we have to then pull apart
      MomNodeBasket basketOut = mapd.makeOutputBasket(basketIn);

      //We know there are two outputs from the mapping so we'll just get them.
      //In a more complicated case we could use a loop.
      MomNode output1 = basketOut.getNode(0);
      MomNode output2 = basketOut.getNode(1);

      //Format the Output Moms
      System.out.println(output1.format(MomNode.SERIALIZE_FORMATTED));
      System.out.println(output2.format(MomNode.SERIALIZE_FORMATTED));
    }
    catch (ConfigFileStructureException cfse) {
      //Occurs if a config file is incorrectly structured
      //thrown by new Project, MappingDefinition.getMappingDefinitionRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfse.printStackTrace();
    }
    catch (ConfigFileContentException cfce) {
      //Occurs if one of the config files has invalid configuration details
      //thrown by MappingDefinition.getMappingDefinitionRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfce.printStackTrace();
    }
    catch (UnrecognizedMessageException ume) {
      //Caused by the MessageTypeIdentifier failing to ID the message
      //thrown by the mdg.getMessageDefinitionRunTimeFromContent call
      ume.printStackTrace();
    }
    catch (ParseException pe) {
      //Caused by the parser failing to break up the data during the MomNode.makeMom call
      pe.printStackTrace();
    }
    catch (WriteMessageContentException wmce) {
      //Caused by mom.format failing
      wmce.printStackTrace();
    }
    catch(ActionException ae) {
      //Caused by an error during the mapping
      ae.printStackTrace();
    }
  }

  public static void main(String[] args) {
    OneToMany mc = new OneToMany();
  }

}
```

6.9.9 Perform a One to Many Mapping with Repeats

A single CSV message is read in to a single mapping, and mapped into two similar XML messages. The new messages are then printed out.

```
package apitest;

import com.tracegroup.transformer.configfiles.*;
import com.tracegroup.transformer.dad.*;
import com.tracegroup.transformer.mom.*;

/**
 * This class maps a simple CVS structure a repeating XML message.
 * @author Keith Donovan
 */
public class OneToManyRepeating {
  public OneToManyRepeating() {

    try {

      //The message data. This could be picked up from a file or passed into a MDB
      //or got off a data bus.
      //Note that in this example the first field gives the type of the message
      String messageDataOne = "CSV3,Felix,Silver,Flossy,Daisy,Pinky";
      //The project location and the group name
      //These could be hard coded or got out of JNDI
      //Creating a new project is a big overhead - this is not something
      //that should be done for each message. The project once constructed
      //should be cached in some way.
      Project project = new Project("C:/transformer/apiTest/ApiTest.tpj");
      String groupName = "MyCsv";
      //The mapping information
      String mappingGroup = "MyCsv2MyXml";
      String mappingName = "Csv3ToBBBRepeating";

      //Given the group name get the MDG object.
      MessageDefinitionGroup mdg =
          MessageDefinitionGroup.getMessageDefinitionGroupRunTime(project,groupName);

      //You know the group, a message Type Identifier (MTI) has been set up
      //so you should be able to get a message definition object if the data is OK
      //Note that if you know the name of the message definition you can do this more quickly
      //with a call like this:
      // MessageDefinition md =
      //     MessageDefinition.getMessageDefinitionRunTime(project,groupName,MessageName);
      MessageDefinition md1 = mdg.getMessageDefinitionRunTimeFromContent(messageDataOne);

      //Construct the Mom Structure from the data
      MomNode mom1 = MomNode.makeMom(md1, messageDataOne);

      //Get the mapping definition
      MappingDefinition mapd = MappingDefinition.getMappingDefinitionRunTime(project,
          GroupName.MAPPING_DEFINITION_GROUPS.toString(),mappingGroup,mappingName);



      //We have complex mapping so we need to bundle the inputs together by
      //putting them in a basket.
      //Create the input basket by getting it from the mapping.
      MomNodeBasket basketIn = mapd.makeInputBasket();
      //Add the two mom nodes to the basket
      basketIn.putNode(mom1,0);


      //Perform the mapping by making the call on the mapping definition
      //Note that this is different to the simple mapping case
      //A basket is returned from the mapping which we have to then pull apart
      MomNodeBasket basketOut = mapd.makeOutputBasket(basketIn);

      //We know there are only two repeats so just fetch them. In more complicated
      //cases we could use a loop here
      MomNode repeat1 = basketOut.getRepeatingNode(0,0);
      MomNode repeat2 = basketOut.getRepeatingNode(0,1);

      //Format the Output Moms
      System.out.println(repeat1.format(MomNode.SERIALIZE_FORMATTED));
      System.out.println(repeat2.format(MomNode.SERIALIZE_FORMATTED));

    }
    catch (ConfigFileStructureException cfse) {
      //Occurs if a config file is incorrectly structured
      //thrown by new Project, MappingDefinition.getMappingDefinitionRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfse.printStackTrace();
    }
    catch (ConfigFileContentException cfce) {
      //Occurs if one of the config files has invalid configuration details
      //thrown by MappingDefinition.getMappingDefinitionRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfce.printStackTrace();
    }
    catch (UnrecognizedMessageException ume) {
      //Caused by the MessageTypeIdentifier failing to ID the message
      //thrown by the mdg.getMessageDefinitionRunTimeFromContent call
      ume.printStackTrace();
    }
    catch (ParseException pe) {
      //Caused by the parser failing to break up the data during the MomNode.makeMom call
      pe.printStackTrace();
    }
    catch (WriteMessageContentException wmce) {
      //Caused by mom.format failing
      wmce.printStackTrace();
    }
    catch(ActionException ae) {
      //Caused by an error during the mapping
      ae.printStackTrace();
    }


  }

  public static void main(String[] args) {
    OneToManyRepeating mc = new OneToManyRepeating();
  }

}
```

6.9.10 Perform a Mapping to a String BOT

A single field in a CSV message is mapped to a String Business Object Type. This is considered to be a complex mapping and follows the same pattern.

```
package apitest;

import com.tracegroup.transformer.configfiles.*;
import com.tracegroup.transformer.dad.*;
import com.tracegroup.transformer.mom.*;

/**
 * This class maps a simple CVS structure to a single String Business Object Type.
 * @author Keith Donovan
 */
public class OneToBot {
  public OneToBot() {

    try {

      //The message data. This could be picked up from a file or passed into a MDB
      //or got off a data bus.
      //Note that in this example the first field gives the type of the message
      String messageDataOne = "CSV1,Alpha,100";


      //The project location and the group name
      //These could be hard coded or got out of JNDI
      //Creating a new project is a big overhead - this is not something
      //that should be done for each message. The project once constructed
      //should be cached in some way.
      Project project = new Project("C:/transformer/apiTest/ApiTest.tpj");
      String groupName = "MyCsv";
      //The mapping information
      String mappingGroup = "MyCsv2MyXml";
      String mappingName = "Csv1ToStringBot";

      //Given the group name get the MDG object.
      MessageDefinitionGroup mdg =
          MessageDefinitionGroup.getMessageDefinitionGroupRunTime(project,groupName);

      //You know the group, a message Type Identifier (MTI) has been set up
      //so you should be able to get a message definition object if the data is OK
      //Note that if you know the name of the message definition you can do this more quickly
      //with a call like this:
      // MessageDefinition md =
      //    MessageDefinition.getMessageDefinitionRunTime(project,groupName,MessageName);
      MessageDefinition md1 = mdg.getMessageDefinitionRunTimeFromContent(messageDataOne);


      //Construct the Mom Structures from the data
      MomNode mom1 = MomNode.makeMom(md1, messageDataOne);

      //Get the mapping definition
      MappingDefinition mapd =
          MappingDefinition.getMappingDefinitionRunTime(project,
          GroupName.MAPPING_DEFINITION_GROUPS.toString(), mappingGroup, mappingName);

      //If the output is a single BOT then that is classed as a complex mapping.
      //We have complex mapping so we need to bundle the inputs together by
      //putting them in a basket.
      //Create the input basket by getting it from the mapping.
      MomNodeBasket basketIn = mapd.makeInputBasket();
      //Add the mom node to the basket
      basketIn.putNode(mom1,0);

      //Perform the mapping by making the call on the mapping definition
      //Note that this is different to the simple mapping case
      //A basket is returned from the mapping which we have to then pull apart
      MomNodeBasket basketOut = mapd.makeOutputBasket(basketIn);

      //We know there is only a single output and it will be simple
      //so we can cast it straight to a SimpleNode
      SimpleNode output = (SimpleNode)basketOut.getNode(0);

      //Format the Output Mom
      System.out.println(output.getStringValue());


    }
    catch (ConfigFileStructureException cfse) {
      //Occurs if a config file is incorrectly structured
      //thrown by new Project, MappingDefinition.getMappingDefinitionRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfse.printStackTrace();
    }
    catch (ConfigFileContentException cfce) {
      //Occurs if one of the config files has invalid configuration details
      //thrown by MappingDefinition.getMappingDefinitionRunTime,
      //MessageDefinitionGroup.getMessageDefinitionGroupRunTime and
      //mdg.getMessageDefinitionRunTimeFromContent
      cfce.printStackTrace();
    }
    catch (UnrecognizedMessageException ume) {
      //Caused by the MessageTypeIdentifier failing to ID the message
      //thrown by the mdg.getMessageDefinitionRunTimeFromContent call
      ume.printStackTrace();
    }
    catch (ParseException pe) {
      //Caused by the parser failing to break up the data during the MomNode.makeMom call
      pe.printStackTrace();
    }
    catch(ActionException ae) {
      //Caused by an error during the mapping
      ae.printStackTrace();
    }
```

```
  }

  public static void main(String[] args) {
    OneToBot mc = new OneToBot();
  }

}
```

6.10 Summary

There are a number of basic tasks that the API can perform:

- Get Runtime Definitions
- Make a Mom Object from its Definition and some data i.e. parse the data.
- Create a new Mom Object by performing a Mapping
- Format a Mom for output

and a number of more advanced tasks some of which are:

- Identify a Message from its content
- Perform an Enrichment
- Invoke a Condition
- Apply a Rule Set (i.e. apply Validation or Enrichment to a Message)
- Checking data structure and error reporting

Detailed examples of these tasks have been covered but the API calls that match these tasks are as follows:

- Get Runtime Definitions - The static getRuntime methods on each Definition e.g.

  ```
  MappingDefinition  md  =  MappingDefinition.getMappingDefinitionRunTime(…)
  ```

- Make a Mom Object from its Definition and some data i.e. parse the data – The static makeMom method on MomNode e.g.

  ```
  MomNode mom = MomNode.makeMom(messageDef, messageData);
  ```

- Create a new Mom Object by performing a Mapping – the makeOutputBasket method on MappingDefinitoin or the makeNodeViaMap method on MomNode e.g.

  ```
  MomNodeBasket basketOut = mapDef.makeOutputBasket(basketIn);
  ```

  Or

  ```
  MomNode output = mom.makeNodeViaMap(mapDef);
  ```

- Format a Mom for output – the format or formatAsXml methods on MomNode e.g.

  ```
  outputMom.format(MomNode.SERIALIZE_FORMATTED);
  ```

- Identify a Message from it’s content – invoke the Message Type Identifier (MTI) by calling getMessageDefinitionRunTimeFromContent on Message Definition Group, this relies on the fact that an MTI has been set up for that MDG e.g.

  ```
  MessageDefinition md1 = mdg.getMessageDefinitionRunTimeFromContent(messageDataOne);
  ```

- Perform an Enrichment – by calling enrich on a NamedEnrichmentDefinition

  ```
  enrichDef.enrich(basketIn);
  ```

- Invoke a Condition – by calling evaluate on a Condition Definition

  ```
  boolean result = cd.evaluate(basket);
  ```

- Apply a Rule Set (i.e. apply Validation or Enrichment to a Message) – by calling enrich on an EnrichmentRuleSet or validate on a ValidationRuleSet e.g.

  ```
  ers.enrich(mom);
  ```

  Or

  ```
  boolean result = vrs.validate(mom);
  ```

- Checking data structure and error reporting – by calling validateStructure and getExceptionReport on a MomNode e.g.

  ```
  System.out.println(mom.getExceptionReport(vrs.getName().toString()));
  ```

7 Coarse API Generated Code

The Coarse API takes a different approach to exposing calls to the API programmer.

The GUI design tool includes facilities to generate Java code which encapsulates use of the classes and methods discussed in the previous sections and provides the API programmer with a means to indirectly call Mappings, Enrichments, and Validation Rules etc. just by making a single Java call.

The Coarse API can be accessed by creating a Coarse API exposed service and then building it. This will generate some code that greatly simplifies the calling of a mapping or validation rule etc.

For example, the user of the GUI design tool may choose to expose a mapping from a back office system to a settlement system as a service where the input and output data were being provided in a String format. This will generate Java code exposing a method which might look something like:

```
public String backOfficeToSettlement(String messageIn);
```

The API programmer just needs to supply the input data, and the generated code will call the mapping and return the serialized result. To give a further example, if your Exposed Service is called “TestService” and your Operation is called “CallMapping” and the Operation has been configured to expect a String Input and A String Output. Then to call your mapping the following code is required:

```
String inputData = ...

TestService srv = new TestService();

String output = srv.callMapping(inputData);
```

The code generation is configured by creating a Coarse API Exposed Service in the Transformer GUI and then adding an Exposed Service Operation for each mapping (or enrichment etc) that needs to be called. Building the service will create a Java Class of the same name, which includes a method representing each operation.

The package in which the classes are generated is a parameter to the service, if the parameter is not specified the default package is com.tracegroup.transformer.exposedservices.serviceName. It is important that if the user specifies a package name that it should be unique for each service.

The method signature for each generated method will be determined by the Operation’s service performer and its parameters.

- The number of input parameters to the method is determined by the number of inputs to the mapping (or enrichment etc) being called.
- The input parameters types are determined by the “Parameter Type” property of the Operation. The four options are:
    1. Bytes - will require the data to be provided as byte arrays.
    2. String - will require the data to be provided as a String.
    3. MomNode – the appropriate MomNode must have already be constructed from the data stream
    4. External Object – a Java Object that has previously been imported into Transformer and has a definition representing it.

- If any of the inputs to the mapping are repeating, then the corresponding input to the method will be turned into an array. So if the “Parameter Type” is a String then the required input will be an array of Strings whilst if the “Parameter Type” is Bytes then a two dimensional byte array will be required.
- If the mapping has two or more outputs then an additional class will be created to store the outputs. The class will be called <*NameOfOperation*>OutputWrapper and will have get and set accessors for each of the outputs.
- If the action being called is performing any sort of validation on the output message, then the method will return an Object to wrap up the returned data and any error messages. The returned Object will depend on the “Parameter Type” of the Operation and the Service Performer. For mappings using Strings a ValidatedStringMessage will be returned, for byte mappings a ValidatedBytesMessage will be used and for Operations using a MessageValidation service performer then a ValidationReport will be returned.

When a Coarse API Service is built a number of files are created in the specified output directory. The created files are:

`ServiceName.jar` - this is the Jar file that contains the built project and the compiled, generated class files representing the service. The name of the jar is determined by a combination of the project key, project version and status. In previous releases these were two separate jar files.

`generated_files/src` - these are the java source files created by the service.

`generated_files/classes` - these are the compiled java class files.

7.1 Coarse API Exceptions

The generated methods are declared to throw exceptions as detailed below:

- StructureException – this will get thrown if any of your config files are invalid. If you’ve checked and built your project then it is likely that the main reason for this is that that database is unavailable and so the database queries can’t be validated.
- RuleException – these correspond to the individual actions in a mapping erroring. This may be the result of a mistake in the mapping or a can be a deliberate strategy from the author of the mapping in the face of invalid data or uncatered for data patterns – there needs to be an agreement between the mapping authors and the programmers of the embedding application as to what is the significance of these exceptions
- UnrecognizedMessageException – this is an MTI failure – i.e. not being able to identify a message and will only occur if you expose an entire group. I.e. you receive a Swift message type that you are not expecting to deal with
- MomException – this occurs if you have a fundamental problem with a mom node i.e. an input message cannot be parsed or a output XML message has invalid characters in it. This exception will also be thrown if Input Validation is turned on and fails.

7.2 Coarse API and Database Connections

A Database Connection may be shared with an Exposed Service via the use of the RuntimeContext. There is an option in the GUI design tool to add the RuntimeContext as a parameter to each generated method.

E.g. for the following generated method:

```
     String backOfficeToSettlement(String messageIn, RuntimeContext context)
```

The following calling code could be used:

```
    Connection conn = ...
    ExposedService srv = ...

    RuntimeContextImpl ctx = new RuntimeContextImpl();
    ctx.setConnection("myDSN", conn);
    srv.backOfficeToSettlement(msgContent,ctx);
```

8 Service Performer API

With the generated code approach, if there is a new project, a new exposed service or a new exposed service operation then new calling code will have to be written and compiled. In some situations the calling application would be able to work out the names of the projects, services and operations from some kind of configuration, routing table, or metadata, etc., and it is not desirable to have write or compile calling code to take on a new project or operation.

The service performer API allows more generalised calling code to interact with exposed services defined in Transformer projects, without the need for reprogramming or compilation.

The application programmer just has to supply the Project Key, the name of the service, the name of the operation, and the input data.

Actually, the Coarse API generated code just sits on top of the Service Performer API

A Service Performer is class that is associated with an Exposed Service Operation and is used to invoke it. One way to look at a performer is as a single access point for all of the available API options in the detailed API.

There are two types of Service Performer: Single Message Service and Many to Many Service. The Single Message Service can be further broken down into the all the other types of Service which are not Many to Many (i.e. Message Validation, XML Interopability etc.).

The service performers can perform operations taking String or byte array input, creating MomNode objects, processing those objects, and producing output which can be serialized back to String or byte array, and can also convey validation information about the input (or output). The MomNode objects themselves carry the most information (e.g., a MomNode object can give various kinds of serialized form, and various kinds of error reports), but MomNodes are only really relevant if the API programmer is going to use the Detailed API manipulation methods. Also because a MomNode has a reference back to the Project that created it there is a “leakage” consideration if MomNode or MomNodeWrapper is returned back into the calling application at large, the Project instance is non-collectable as long as some part of the calling application has a reference to that MomNode.

So it is normally more appropriate to get a serialized form back (either String or byte array), or a validation report, or if the caller wants a serialized form ***and*** a validation report a composite data structure has to be returned that contains both. Composite data objects are also needed for input and output when the operation is Many To Many. The composite data objects, or transport objects, are described in section 8.1.

So the most general approach is to call a method provided by all ServicePerformers which expects input in a form compatible with the definition of the Service Operation, and returns output in a form determined by the definition of the Service Operation. The ServicePerformerExecutor class has static methods which encapsulate the necessary calls to get and release the Project instance and to locate the Service Operation definition and get the relevant ServicePerformer. The API programmer will normally call this method:

```
/**
 * Performs a ServiceOperation and re-throws any exceptions as the simple serializable
 * exception in the exposed services package.
 * @param projectProvider the Project provider
 * @param projectKey the Project key
 * @param serviceName the service name
 * @param operationName the operation name
 * @param input an input Object consistent with the input parameter type settings on the
 * service operation definition
 * @param context the RuntimeContext
 * @return the return value of the service operation in a form consistent with the
 *         operation definition
 */
public static Object performServiceWithSimpleExceptions(
        final IProjectProvider projectProvider,
        final String projectKey,
        final String serviceName,
        final String operationName,
        final Object input,
        final RuntimeContext context)
        throws com.tracegroup.transformer.exposedservices.MomException,
               com.tracegroup.transformer.exposedservices.UnrecognizedMessageException,
               com.tracegroup.transformer.exposedservices.RuleException,
               com.tracegroup.transformer.exposedservices.StructureException
```

The input form should be one of the following:

- For a single message service with “String” input type, a String
- For a single message service with “Bytes” input type, a byte array
- For a single message service with “External Object” input type, the external object
- For a Many to Many service with “String” input type, an array of Strings (if there are repeating inputs, use nested arrays to represent the repeats)
- For a Many to Many service with “Bytes” input type, an array of byte arrays (if there are repeating inputs, use nested arrays to represent the repeats)
- For a Many to Many service with “External Object” input type, an array of objects (if there are repeating inputs, use nested arrays to represent the repeats)
- For any kind of service with “String” input type, a StringBasket
- For any kind of service with “Bytes” input type, a BytesBasket
- For any kind of service with “External Object” input type, an ObjectBasket

The output form will be as follows. If the input was a basket, the output will also be a basket, otherwise:

- For a single message service where validation is specified in the operation definition, a ValidatedStringMessage, ValidatedBytesMessage or ValidatedObjectMessage (depending on the output type)
- For a single message service where no validation is specified, a String, byte array or Object (depending on the output type)
- For a Many to Many service, a Map in which the key is the output definition name, and the value follows the same rules as above (i.e., value can be ValidatedStringMessage, ValidatedBytesMessage etc.). If there is a repeating output, the corresponding value in the map will be an array of ValidatedStringMessage, array of ValidatedBytesMessage, etc.

8.1 Transport Classes

There are a number of classes that are used to wrap up the inputs to and outputs from Exposed Service calls. The transport classes fall into two categories those that are used solely as outputs:

- ValidatedStringMessage
- ValidatedBytesMessage
- ValidatedObjectMessage
- ValidationReport

And those that can be used either as inputs or outputs:

- StringBasket
- BytesBasket
- ObjectBasket

8.1.1 Output Only Classes

The first category is used for services which return a single message, and also perform validation on the output. So generally speaking the classes contain the message content and an error report.

ValidatedStringMessage is a class that has an accessor to retrieve the message contents as a String and further accessors to return the validation status

- String getText() – the message content
- boolean isValid() – is the message content valid
- String getErrorReport() – the error message describing the problem if the message content is invalid.

ValidatedBytesMessage is a class that has an accessor to retrieve the message contents as a byte array and further accessors to return the validation status

- byte[] getData() – the message content
- boolean isValid() – is the message content valid
- String getErrorReport() – the error message describing the problem if the message content is invalid.

ValidatedObjectMessage is a class that has an accessor to retrieve the message contents as an Object and further accessors to return the validation status

- Object getObject() – the message content
- boolean isValid() – is the message content valid
- String getErrorReport() – the error message describing the problem if the message content is invalid.

ValidationReport is a class that is used if the user is only interested in the validity of a message i.e. it does not return the contents of the original message

- boolean isValid() – is the message content valid
- String getErrorReport() – the error message describing the problem if the message is invalid.

8.1.2 Input/Output Classes

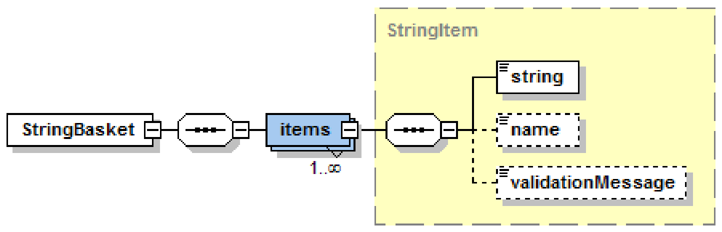
The classes that can be used as inputs or outputs are the Basket classes. These classes are containers for messages of a particular type e.g. a StringBasket is made up of one or more String based messages. As the Baskets can be contain multiple messages they can be used for both one-to-one and many-to-many services.

There are three concrete implementations of ItemBasket namely StringBasket (for the transport of String based messages), BytesBasket (for the transport of messages as byte arrays) and ObjectBasket (for Objects).

Each Basket class consist of one or more Items. An item residing in a basket can be addressed by position or optionally by name. The position (and name) of an item in its basket corresponds to inputs (or outputs) of the mapping that the basket is representing. The naming of the items is optional unless the corresponding mapping has both multiple and repeating inputs.

BytesBaskets always consist of BytesItems, StringBaskets are made up of StringItems and ObjectBaskets consist of ObjectItems. The Items consist of a field of the corresponding type (i.e. StringItem will have a String and ByteItem will have a byte[]), a validation message and a name. An item can be named so that it can be retrieved from its parent Basket by its name.

The relationship between a Basket and its items could be view diagrammatically as (this is just the String case):



The combination of the Basket and its Items corresponds to the structure of an input or output basket and in turn the input or output to a mapping. Each StringItem in the StringBasket corresponds to an input to (or output from) the mapping. So for example if you had a mapping that had the following input structure:

- Input1 – mandatory non-repeating
- Input2 – optional non-repeating
- Input3 – mandatory repeating

Then if you wanted to load the corresponding StringBasket with appropriate content to create Input1, an empty Input2 and two repeats of Input3 you would have to create the following.

The list in StringBasket contains three StringItems (one for the first input and two for the last) and they must be named (as there are repeats and multiple inputs). To access the Basket the user could use indices or names.

Loading a Basket is relatively straightforward, for example to load a StringBasket with two inputs one called in1 and another called in2 the user just needs to call the following:

```
String input1 = …
String input2 = …

StringBasket sbIn = new StringBasket();
sbIn.add("in1", input1);
sbIn.add("in2", input2);
```

9 Database and DataSource Considerations

There are two main considerations for the API programmer when deploying Transformer definitions which include use of databases.

9.1 DataSource Names and DataSources

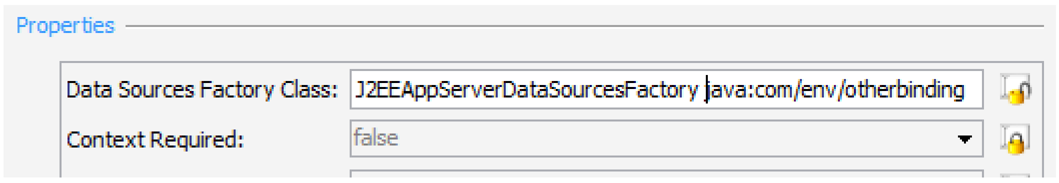
Transformer SQL Query Definitions and related definitions refer to databases by using data source names rather than storing any details. The correspondence between a data source name and a database is part of the environment in which Transformer is running and involves four stages:

1. The Transformer code gets an actual database connection from an object that implements the interface `javax.sql.DataSource`.
2. This `DataSource` object is obtained from an object that implements the interface `com.tracegroup.resources.DataSources`. This interface defines 5 simple methods, and the code is attached.
3. When a Transformer `Project` object is created a `DataSources` object is obtained from an object that implements the interface `com.tracegroup.transformer.plugin.DataSourcesFactory`. This interface defines one method which returns a `DataSources` object for a given Transformer `Project`. (It is also possible to specifically set the `DataSources` object for a `Project` through the API, but this is not usually necessary).
4. The `DataSourcesFactory` to use when projects are being created is configured by the `TransformerEnv` static method `setDataSourcesFactory`.

Transformer is shipped with three implementations of `DataSourcesFactory`.

The first of these is `J2EEDataSourcesFactory` which incorporates its own inner class implementation of the `DataSources` interface. The code works on the principle that the data source names will be bound to `DataSource` objects in the initial context provided by the app server. The Coarse API uses this DataSources Factory as its default.

The `J2EEDataSourcesFactory` uses `"java:comp/env/jdbc"` as its initial binding. This can be overridden in the Transformer GUI by adding an optional binding property after the class name in the Exposed Service Proeprties Dialog. See Screen shot below



Data Sources Factory Class: J2EEAppServerDataSourcesFactory java:com/env/otherbinding

Context Required: false

The second implementation `TDataSourcesFactory` uses an .ini file to load the Datasources - the GUI design tool uses this method to store database definitions.

Finally, the `ClasspathDataSourceFactory` loads the Datasources.ini file from the Classpath.

9.2 Database Transactionality Considerations

If Transformer is used to perform database updates, transactionality considerations are the responsibility of the API programmer when embedding the Transformer definitions into a calling application.

The two basic approaches are:

(a) Allow Transformer to allocate its own database connections, and rely on autocommit for each update. Obviously this is a simplisitic approach which may be inappropriate for many applications.

(b) Manage the connection(s) in the calling application, and pass the appropriate connections to Transformer when calling mappings or othe NamedActionSets which involve database updates, via a RuntimeContext object (see example below).

If a particular mapping or operation was using a DataSource named “myDSN” the following code could be used to manage the transactionality:

```
Connection conn = ...;
try {
  RuntimeContextImpl ctx = new RuntimeContextImpl();
  ctx.setConnection("myDSN", conn);
  //Call a mapping or perform an operation etc passing in the context
  ...
  conn.commit();
}
catch (Exception e) {
  conn.rollback();
}
```

10 API Use of SqlQueryDefinitions

The API programmer may leverage SqlQueryDefinitions which hava been created in the Transformer GUI design tool, by using them to select from, update, insert to or delete from a database. To perform these operations a set of SqlExecuter classes have been created (one for each type of operation). These Executers can be obtained from the SqlQueryDefinition that they use. Each Executor requires a java.sql.Connection to have already been made to the required database. Each type of Executer is detailed in turn.

10.1 Select

An SQL ResultSet can be obtained from a database by the use of a SqlQueryDefinition. To perform the database select a JDBC Connection to the database should already have been obtained and the query definition loaded. If the query requires a where condition to be built up then this can be done by supplying an array of values and calling the setWhereParameters method or specifying the named where parameters individually using the setWhereParameter method.

```
SqlQueryDefinition sql = …
Connection conn = …
Object[] params = …
SqlSelecter sel = sql.rgetSelector();
PreparedStatement ps = sel.getPreparedStatement(conn);
sel.setWhereParameters(ps, params);
ResultSet rs = sel.execute(ps);
```

10.2 Insert

A complex node representing a SqlQueryDefinition result set can be inserted into a database. To perform the database insert a JDBC Connection to the database should already have been obtained and the query definition, representing the table being altered, already loaded.

```
SqlQueryDefinition sql = …
ComplexNode newCn = …
Connection conn = …
SqlInserter ins = sql.rgetInserter(false);
PreparedStatement ps = ins.getPreparedStatement(conn);
ins.execute(ps, newCn);
```

An insert with generated keys can also be performed. The result of the insert will be the primary key of the inserted row.

```
SqlInserter ins = sql.rgetInserter(true);
PreparedStatement ps = ins.getPreparedStatement(conn);
ins.execute(ps, newCn);
BigDecimal key = (BigDecimal)ins.getResults(ps, 0);
```

10.3 Update

A complex node representing a SqlQueryDefinition result set can be used for a database update. To perform the database update a JDBC Connection to the database should already have been obtained and the query definition, representing the table being altered, already loaded. If the update requires a where condition to be built up then this can be done by supplying an array of values and calling the setWhereParameters method or specifying the named where parameters individually using the setWhereParameter method. The result of the update will be the number of rows affected.

```
SqlQueryDefinition sql = …
ComplexNode newCn = …
Connection conn = …
SqlUpdater up = sql.rgetUpdater();
PreparedStatement ps = up.getPreparedStatement(conn);
up.setWhereParameter(ps, "key", new BigInteger("10001"));
int rows = up.execute(ps, newCn);
```

10.4 Delete

A SqlQueryDefinition set can be used to delete rows from a database. The only relevant parts of the query are the table name and the where condition. To perform the database delete a JDBC Connection to the database should already have been obtained and the query definition, representing the table being altered, already loaded. If the delete requires a where condition to be built up then this can be done by supplying an array of values and calling the setWhereParameters method or specifying the named where parameters individually using the setWhereParameter method. The result of the delete will be the number of rows affected.

```
SqlQueryDefinition sql = …
Connection conn = …
SqlDeleter del = sql.rgetDeleter();
PreparedStatement ps = del.getPreparedStatement(conn);
del.setWhereParameter(ps, "key", new BigInteger("10001"));
int rows = del.execute(ps);
```

11 Log4J and the Application Log

Log4j is part of the Apache logging services project, and combines efficiency and sophistication. It allows configuration via programmatic means and also via configuration files. More details, and downloads, are available at *http://logging.apache.org/log4j*.

Transformer's interaction with log4j is as follows:

- A hierarchy of log4j Loggers (aka Categories) represents the fact that Trace applications wish to pass messages to users.
  - ➢ The head of the hierarchy for application messages is "`com.tracegroup.APP.LOG`", and there are more specific categories for specific Trace applications, specifically "`com.tracegroup.APP.LOG.transformer`".
  - ➢ A different branch of this hierarchy "`com.tracegroup.APP.EXCEPTION`" is used as a means of reporting any exceptions that occur.
- Output to the application log Logger can arise from the following places:
  - ➢ Transformer itself outputs various information, warning and error messages.
  - ➢ The Log mapping action which can be included in MappingDefinitions and Rules produces output to the application log at INFO level.
  - ➢ Mappers or other TBeans written by Transformer users may use methods of the `TransformerEnv` class such as `print`, `printWarning`, `displayException` etc., which produce output to the application log. Note, however, that these should only relate to situations where the message is relevant to the system user, not as a means of tracing or debugging for the developer's benefit (use log4j directly for that purpose, with a Logger category based on the class name as is normal practice).
- Transformer provides Log4j Appender implementations to output to System.out (for simplisitic runtime behaviour), and to the GUI MessagesPane (GUI behaviour).
- Mappers or other TBeans written by Transformer users are free to use log4j directly, and if desired an appropriate appender configuration can be created to direct their output to the same place as the Transformer application log (e.g., the Messages Pane in the GUI).

11.1 Configuration for Transformer GUI

Log4j configuration in the GUI is primarily via a log4j.properties file included in the Transformer GUI jar as part of the installation, which has the following settings.

```
############## Log Pane Appender ######################
```

```
# Set root logger level to FATAL and its only appender to ConsoleAppender.
log4j.rootLogger=ERROR, basicAppender
log4j.appender.basicAppender=org.apache.log4j.ConsoleAppender
log4j.appender.basicAppender.layout=org.apache.log4j.PatternLayout

# Logging messages to be shown in the logpane
log4j.appender.LogPane=com.tracegroup.transformer.gui.logging.LogPaneAppender
log4j.logger.com.tracegroup.APP=INFO, LogPane
log4j.additivity.com.tracegroup.APP=false

############## Logger Mapper ######################
# Logging Log mapper to be shown in the logpane
log4j.logger.com.tracegroup.IMP_DIAG=ALL, LogPane
log4j.additivity.com.tracegroup.IMP_DIAG =false
```

A supplementary file, which must also be called log4j.properties, can be created in the install directory, to manage the logging level etc. for Mappers written by the Transformer user.

Log4j itself is included in the Transformer GUI installation.

11.2 Configuration at Runtime

There is no pre-provided mechanism within Transformer to direct the Transformer application log output at runtime.

Responsibility rests with the deployer, as follows:

(a) To ensure that log4j is available on the classpath. Trace has used Log4j version 1.2.16 for development and testing of Transformer version 3.2.0

(b) To define a log4j configuration using normal log4j configuration options.

11.3 Alternative to Log4J

Transformer has had its logging mechanism abstracted so that users can plug in their own logging framework at deployment time. To achieve this Transformer makes uses of the Simple Logging Façade for Java (SLF4J) and more details can be found out about it here: http://www.slf4j.org/

Transformer's SLF4J configuration has been tested with Log4J.

To alter a logging system to make it pluggable into Transformer the user must implement the `LoggingConfigurationReloader` interface. This will cause the logging system to be reconfigured as per the implementation. The logging system could be reconfigured via files, system settings or any other means that the implementation is written to perform. The implementation requires just a single method:

```
    void reconfigure();
```

The following is an example of a Logback implementation:

```
public class LogbackLoggingConfigurationReloader implements LoggingConfigurationReloader {

  private static final Logger LOGGER =
LoggerFactory.getLogger(LogbackLoggingConfigurationReloader.class);

  public void reconfigure() {
    LoggerContext lc = (LoggerContext)LoggerFactory.getILoggerFactory();

    JoranConfigurator configurator = new JoranConfigurator();
    configurator.setContext(lc);

    // Reset the configuration
    lc.reset();

    // Now reload the configuration
    try {

      URL logbackFile = getClass().getResource("/logback.xml");
      if (logbackFile != null) {
        configurator.doConfigure(logbackFile);
      } else {
        LOGGER.warn("Could not find a logback configuration file to reload from");
      }
    }
    catch (JoranException e) {
      LOGGER.error("Failed to reload logging configuration 'logback.xml' from classpath", e);
    }
    StatusPrinter.printInCaseOfErrorsOrWarnings(lc);
  }
}
```

To use the new logging mechanism the user must install it into TraceEnv:

```
TraceEnv.put(LoggingConfigurationReloader.class, new LogbackLoggingConfigurationReloader());
```

The jar or class files defining the new logging mechanism must be added to the classpath.

12 Streaming API for Transformer (SAT)

The Streaming API for Transformer is a mechanism allowing Transformer definitions to be applied to an arbitrarily large file (or stream) of messages. Although Transformer's other features will allow a file to be treated as a single large message in which the records appear as a repeating group, this approach is liable to exhaust the available memory on the system if the file is arbitrarily large.

Let us consider a typical use case for Transformer. A client has a file of SWIFT messages – each message in the file is separated by the $ character. Each of these messages must be transformed into a CSV style message and written out to new line in a specified file.

A number of data **sources** (SATSources) are provided to read different types of input stream (or SATFlavors) – string based, byte array based and streams of MomNodes. At runtime, as a source discovers each separate message in the stream it calls a **client** (a SATOperator) to process that message. The client is responsible for taking the input message, doing something with it and then (optionally) passing it on to a **result handler**, which is also a SATOperator. This process allows SATOperators to be chained together to perform multiple operations on the same input message.

The supplied example of a client uses one or more Coarse API Service Operations to process header, trailer and data messages.

Several result handler implementations are provided, e.g., to write results to a file.

12.1 Terminology

- **SATSource** – a stream of input messages. Note that the term "message" is used in this context to denote something which is to be treated as a single message from a requirements point of view. I.e., it is a loose term. A SATSource class is responsible for opening, reading, and closing the stream, and breaking the stream into individual messages, each of which is handed on to its client SATOperator.
- **SATFlavor** – a data type that a Source or Operator supports, identifying a Java class which can represent the content of a message from the stream. Supported Flavors are:
    - String
    - byte[]
    - MomNode
    - MomNodeBasket
- **SATSourceState** – this class keeps track of the state of the stream i.e. what is the record number of the message being processed and is it the last message in the stream.
- **SATRecord** – an object used to represent a single message (either input or output). The content of the message can be provided in one or more SATFlavors.

- **SATOperator** – something which can process an individual message from a source stream. The SATOperator interface applies to two roles – Client and Result Handler.
- **Client** - a Client is a SATOperator called by a SAT Source which hands it a single input message.
- **ResultHandler** – a ResultHandler is a SATOperator called by another SATOperator to output the results of the operation, or provide further processing.
- **SATServiceOperator** is a SATOperator implementation which uses the stream message as the input parameter to a Transformer operation (ExposedServiceOperation). The result of the operation is passed to a ResultHandler if one has been set.
- **SATException**
  - **SATSourceException** – thrown by the SATSource during the reading of the stream. It can be used to wrap up I/O and SQL Exceptions for example.
  - **SATOperationException** – thrown by the Client SATOperator during processing – it can wrap up any Transformer Exceptions.
  - **SATResultException** – this is thrown by the ResultHandler SATOperator during the writing of the output – like the source exception it wraps up I/O and SQL Exceptions.
- **SATExceptionHandler** - an exception handler associated with a SATSource to which SATOperationExceptions and SATResultException will be given. If the exception handler chooses to throw the exception (or if there is no exception handler specified for the source), the source processing will terminate abruptly by throwing the exception.

12.2 Typical Operation

The user (the API programmer) must decide on which SAT operator to use and which SATSource will provide the data.

The programmer creates a SATServiceOperator instance to act as the client, and loads Transformer ExposedServiceOperation definitions for Data messages and optionally for Header and Trailer messages.

If appropriate, the programmer creates a ResultHandler instance to perform the task of writing out the results of the operation. Depending on the type of ResultHandler this may require that the programmer has already created some other object, such as a FileWriter. For operations that do not have any output (Database inserts for example) the ResultHandler is not required.

The programmer uses the SATServiceOperator “set” accessors to associate the client instance with the ExposedServiceOperaions and the result handler instance.

The programmer creates an appropriate SATSource instance. For this example we’ll assume that the user is using a ReaderSATSource, and so the Reader instance must already have been created, and must be in an appropriate state (e.g., just created, rather than already at end of file).

The programmer passes the client instance into the source via the setClient() method.

The programmer can now call the provide() method on the source.

To summarise, the tasks the API programmer has to complete.
- Create the SATSource, by first creating a Reader.
- Create the Client SATOperation
- (Optionally) create the Result Handler, by first creating a Writer
- Call setResultHandler() on the client SATOperation passing in the ResultHandler SATOperation

- Call setClient() on the SATSource passing in the client SATOperation
- (Optionally) call setExceptionHandler() on the SATSource passing in an ExceptionHandler instance
- Call provide() on the Source.

12.3 Threading and Lifecycle Considerations

SATSource and SATOperator methods are not required to be thread-safe, and the implementations provided will not be so.

SATSource and SATOperator instances are expected to be short-lived for the duration of processing one particular input stream, and their lifecycle is managed by the API programmer.

The API programmer is also responsible for managing the state of objects on which a SATSource or SATOperator depends, e.g., closing the Writer in the above example.

ExceptionHandler implementations may follow the same pattern as above. The SAT package places no requirement on ExceptionHandler implementations to be thread-safe. There are no exception handler implementations provided, however an example could be an implementation which writes an error message to the same Writer being used by the ResultHandler. Clearly instances of an example like that would have to be short-lived and under the control of the API programmer.

12.4 Context/State

Context pertaining to the in-process stream is available via two basic means.

One is via SATSourceState object. The SATSource is responsible for maintaining SATSourceState, which is available to SATOperators and ExceptionHandlers. The SATSourceState incorporates a link to the SATSource instance, so that custom SATSource implementations could provide additional context by this means. Also, since SATSource provides “get” accessors for Client and ExceptionHandler, SATOperators and ExceptionHandlers can be aware of each by this means.

The second means is via a RuntimeContext object which can be specified on the SATSource. This is used by the SATServiceOperator implementation when calling any Transformer MappingDefinitions or Rules, and therefore provides a means by which context/state can be shared with actions inside a MappingDefinition or Rule. For example, if a particular field from the header message needed to be available inside the MappingDefinition for a data message, this value could be stored in the runtime context.

12.5 SATException

This has three subclasses responsible for the errors that occur at different points in the processesing.

12.5.1 SATSourceException

This exception is used to wrap up any I/O and SQL exceptions that are thrown by the various SATSources. SATSourceExceptions are not passed to an exception handler, but will always terminate the provide() method abruptly.

12.5.2 SATOperationException

A SATOperator implementation can create SATOperationExceptions to wrap up any exceptions that are thrown during the operation, relating to the processing of the message.

The SATServiceOperator implementation will wrap the following exceptions resulting from performing message validation and mapping:

- MomNodeException
- ActionException
- UnrecognizedMessageException
- ConfigFileStructureException
- ConfigFileContentException

SATOperationExceptions will be passed to the exception handler (if there is one), which is allowed to rethrow. If it does rethrow, or if there is no exception handler, the source provide() method will terminate abruptly. If it does not rethrow, the source will continue with the next message.

12.5.3 SATResultException

A SATOperator implementation can create SATResultExceptions to wrap up any exceptions thrown relating to the output of the operation result.

SATResultExceptions are passed to the exception handler in the same way as SATOperationExceptions.

In many cases it is very likely that the exception handler may rethrow SATResultExceptions even if it is dealing with SATOperationExceptions, e.g., if output is to a FileWriter and there is an IO exception on the write.

12.6 interface SATSourceState

This class is used to keep track of the state of the records in the stream.

12.6.1 int getRecordNumber()

The position of the record in the file. The first record is a position 0.

12.6.2 boolean isLastMessage()

Is the record the last in the file?

12.6.3 boolean isFirstMessage()

Is the record the first in the file?

12.6.4 SATSource getSource()

Returns the source.

12.7 class SATFlavor

The SATFlavor is a means of keeping track of the type of the data. Each SATSource and SATOperator will be declared to work with certain data flavors. A source which only works with data in byte arrays can only be used with a SATOperator that supports the byte array flavor. The predefined data flavors are:

- TEXT – a java string
- BYTES – a byte array
- MOM – a single MomNode
- BASKET – a MomNodeBasket

The SATRecord (see below) will use the data flavour so that only a single accessor is required to access the different types of data.

12.8 interface SATRecord

The SAT Record represents the information passed from a SATSource to its Client or from any SATOperator to its ResultHandler.

This is an interface.

12.8.1 SATFlavor[] getSupportedFlavors()

Return the array of supported flavors.

12.8.2 Object getContent(SATFlavor)

Return the type appropriate to the SATFlavor, or throw IllegalArgument if not supported

12.8.3 SATSourceState getSourceState()

Return the current source state.

12.8.4 SATRecord getFormerRecord()

When SATOperators are chained together, by configuring one SATOperator as the ResultHandler for another, each link in the chain is given a new SATRecord, but with a link to the SATRecord passed into its predecessor.
E.g., the SATRecord passed to the result handler which writes out the result of a Mapping allows access to the corresponding input message via this method.

12.9 interface SATSource

The interface that defines all SAT Sources.

12.9.1 SATFlavor[] getSupportedFlavors()

Return the array of supported flavors.

12.9.2 set/getClient(SATOperator)

Set/Get the client SATOperator on the Source. If the client does not support any of the same SAT Flavors as the Source then an IllegalArgumentException will be thrown.

12.9.3 set/getOperationExceptionHandler

Sets/Gets the operation exception handler.

12.9.4 set/getResultExceptionHandler

Sets/Gets the result exception handler.

12.9.5 setRuntimeContext(RuntimeContext)

Sets the User context.

12.9.6 getRuntimeContext()

Retrieve the user context.

12.9.7 provide() throws SATException

This prepares the source stream for reading, for example, by opening an input file and then reads from and breaks up the stream and then calls the process() method on the client SATOperator.

12.10 class ReaderSATSource implements SATSource

This is the implementation of SATSource that supports the TEXT SATFlavor.

12.10.1 setReader(Reader reader)

Set the Stream reader.

12.10.2 setRecordSeparator(String separator, boolean isSeparatorPartOfRecord)

Set the separator string for the stream. This method also specifies if the separator is part of the data – i.e. should it be removed before the data is given to the client. The separator is used by the Input Source as the delimiter for each incoming message.

12.10.3 setRecordSeparator(String separator)

Set the separator string for the stream. This method assumes that the separator is not part of the record – i.e. the separator will be removed. The separator is used by the Input Source as the delimiter for each incoming message.

12.10.4 String getRecordSeparator()

Return the separator.

12.10.5 provide()

The implementation of the provide method does the following:

- Creates a SATSourceState object and initialises the record count to zero
- Opens and reads from the stream. As it goes it splits the stream into individual messages by breaking on the separator, update the SATSourceState object with the current count and then call the client SATOperator's process() method. To do this it must "read ahead" in the stream to know when it is dealing with the last message.
- If an exception is encountered during reading then it wraps it up as a SATSourceException and throws it
- If the process method throws either a SATOperationException or a SATResultException then the appropriate ExceptionHandler is called, if there is one, otherwise the exception is thrown.

12.11 class InputStreamSATSource implements SATSource

This is the implementation of SATSource that supports the BYTES SATFlavor.

12.11.1 setInputStream(InputStream stream)

Set the Input Stream.

12.11.2 setRecordSeparator(byte[] separator, boolean isSeparatorPartOfRecord)

Set the separator array for the stream. This method also specifies if the separator is part of the data – i.e. should it be removed before the data is given to the client. The separator is used by the Input Source as the delimiter for each incoming message.

12.11.3 setRecordSeparator(byte[] separator)

Set the separator array for the stream. This method assumes that the separator is not part of the record – i.e. the separator will be removed. The separator is used by the Input Source as the delimiter for each incoming message.

12.11.4 byte[] getRecordSeparator()

Return the separator.

12.11.5 provide()

The implementation of the provide method does the following.

- Creates a SATSourceState object and initialises the record count to zero
- Opens and reads from the stream. As it goes it must split the stream into individual messages by breaking on the separator string, update the SATSourceState object with the current count and then call the SATOperator's process () method. To do this it must "read ahead" in the stream to know when it is dealing with the last message.
- If an exception is encountered during reading then it wraps it up as a SATSourceException and throws it
- If the process method throws either a SATOperationException or a SATResultException then the appropriate ExceptionHandler is called, or throw if there is none

12.12 class QueryDefinitionSATSource implements SATSource

This is an implementation of SATSource that only provides the MOM SATFlavor. Its constructor must take SqlQueryDefinition object to use as the input. The source will create the SQL query, perform the select operation and create the MomNodes for each row in the result set.

12.12.1 setWhereParameters(object[] params)

This method allows the callers to set any required where parameters.

12.12.2 provide()

The implementation of the provide method does the following.

- Creates a SATSourceState object and initialises the record count to zero
- Open the SqlQueryDefinition.
- Get a database connection. This should this be taken from the context (if it has been set).
- Perform the SQL select and for each row in the result set create a MomNode, update the SATSourceState object and call the client SATOperator's process() method.
- If an exception is encountered during the SQL select then it wraps it up as a SATSourceException and throws it.
- If the process method throws either a SATOperationException or a SATResultException then the appropriate ExceptionHandler is called.

12.13 interface SATExceptionHandler

A user defined ExceptionHandler to deal with any SAT Exceptions thrown during the processing. Three different types of handler can be set on the SATSource one for each type of SATException.

12.13.1 handleException(SATException)

Deal with the Exception as the user requires.

12.14 interface SATOperator

SATOperators have a twofold responsibility. They can be used in the client capacity to process a single record that has been read from a stream by calling, say, a Transformer operation. Or they can be used as a Result Handler where they are passed the results of an Operation.

12.14.1 process(SATRecord result) throws SATOperationException, SATResultException

The user defined method to process the SAT Result object.

12.14.2 setResultHandler(SATOperator handler)

Set the user provided Result Handler SATOperator.

12.14.3 SATFlavor[] getSupportedFlavors()

States the flavors supported by this operator, preferred flavour first.

12.15 class SATServiceOperator implements SATOperator

This is the concrete implementation of a SATOperator and is used as a SAT Client (i.e. its process method is called by a SATSource). For each incoming record it will call a specified Transformer service operation.

12.15.1 setDataServiceOperation(ExposedServiceOperation op)

This method sets the ExposedServiceOperation that should be called each time a data record is encountered.

12.15.2 ExposedServiceOperation getDataServiceOperation()

Return the data Service Operation.

12.15.3 setHeaderServiceOperation(ExposedServiceOperation headerOp)

This method sets the ExposedServiceOperation that should be called each time a header record is encountered.

12.15.4 ExposedServiceOperation getHeaderServiceOperation()

Return the header Service Operation.

12.15.5 setTrailerServiceOperation(ExposedServiceOperation trailerOp)

This method sets the ExposedServiceOperation that should be called each time a trailer record is encountered.

12.15.6 ExposedServiceOperation getTrailerServiceOperation()

Return the trailer Service Operation.

12.15.7 process(SATRecord result)

This method is inherited from SATOperator.

- The SATSourceState is retrieved from the SATRecord.
- The correct data type is retrieved from the SATRecord using a supported SATFlavor.
- If the SATSourceState indicates that the first record is being processed and a HeaderServiceOperation has been set then its service performer is called.
- If the SATSourceState indicates the last record is being processed and a TrailerServiceOperation has been set then its service performer is called
- Otherwise the DataServiceOperation's service performer is called.
- If the operation throws exception then wrap up the exception as a SATOperationException and throw it.
- If a Result Handler SATOperator has been set
    - Create a new SATRecord and fill it with the result of the operation.
    - Supported flavours on that new SATRecord?
    - Add the original SATRecord to the new SATRecord as its former record.
- Call the Result Handler's process() method passing in the new SATRecord.

12.16 class SATFileWriter implements SATOperator

This is the concrete implementation of a SATOperator and is used as a SAT Result Handler (i.e. its process method is called by a Client SATOperator). Each incoming record is written out to a specified file.

12.16.1 Constructor

The constructor will take a file location and a separator as its arguments. It will create a file writer object and open the file.

12.16.2 process(SATRecord result)

Each output record will be written to the file. If the record is not the last one the separator will also be written. If the last record is received the file will be closed.

12.16.3 SATFlavor[] getSupportedFlavors()

Just return String initially.

12.17 Example

In this example, records are read out of a file a line at a time. The first (header) record is passed to the “Header” Exposed Service Operation and then all subsequent records are passed to the “Data” Operation. All of the outputs are sent to another file, each of the records in the output are separated by a “$” symbol.

```
//Load the Project and the Exposed Service Operations
Project p = …
ConfigFile dataCf = p.getConfigFile("ExposedServices/SatTestService/Data");
ExposedServiceOperation dataService = (ExposedServiceOperation)p.getRunTimeObject(dataCf);
ConfigFile headerCf = p.getConfigFile("ExposedServices/SatTestService/Header");
ExposedServiceOperation headerService = (ExposedServiceOperation)p.getRunTimeObject(headerCf);

//Set up the Source
ReaderSATSource source = new ReaderSATSource();
source.setReader(new FileReader("C:/temp/testFile.txt"));
//The input records are separated by Windows EOL characters
source.setRecordSeparator("\r\n");

//Set up the SAT Client
SATServiceOperator sso = new SATServiceOperator();
//Add the Operations to the Client
sso.setDataServiceOperation(dataService);
sso.setHeaderServiceOperation(headerService);
//Set up the Result Handler – the output records are separated by a $ symbol
SATFileWriter writer = new SATFileWriter(new File("c:/temp/output.dat"),"$");
sso.setResultHandler(writer);
source.setClient(sso);

//Run it
source.provide();
```

12.18 XML SAT

The call/caller relationships involved in processing XML don’t fit with the idea of a SATSource. To get around this problem the SAXSATSource and SAXSATOperator have been introduced. The SAXSATSource contains its own XML Decoder client so it can process an XML document looking for elements of interest. Once found an element of interest can then be given to a SAXSATOperator to be processed.

The SAXSATSource class is used to process large XML messages. If the XML message contains a repeating structure, or only certain elements in the message are of interest then SAXSATSource can be used to apply specified SAXSATOperators to the intesting elements.

The SAXSATSource takes a map as the only parameter to its constructor, the keys to the map are the paths to the XML elements of interest, and the values in the map are the SAXSATOperators that should be applied to the elements.

Three types of path can be used for the key to the map. If the key does not contain a "/" then all the elements with that name will be processed. If the key starts with a "/" then for an element to be processed it must exactly match the path in the key. If the key contains a "/" but doesn't have one at the start then the path to the element must end with the key for it to be processed. For example in the following XML Structure:

```
<Complex>
  <Header>
    <SeqNum>345</SeqNum>
  </Header>
  <First>
    <Body>
      <Name>Keith</Name>
      <DoB>1973-09-12</DoB>
    </Body>
  </First>
  <Second>
       <Body>
      <Name>Bob</Name>
      <DoB>1974-09-12</DoB>
    </Body>
  </Second>
</Complex>
```

The key “Body” will process both Body elements, the key “/Complex/First/Body” will only process the Body element containing the Name Keith and the key “Second/Body” will only process the Body containing Bob.

Two types of SAXSATOperators are available. The SAXSATServiceOperator takes an ExposedServiceOperation as the only parameter to its constructor, it will apply that Operation to its elements of interest. The GlobalSAXSATOperator takes a String as the only parameter to its constructor, the String should be the name of a global variable in the current RuntimeContext. The element of interest will have its content copied to the value of the specified global variable.

An Example of how the SAXSATOperators can be used in conjunction with a SAXSATSource is given below.

```
    Project p = ...

     //Set up the Runtime Context that is used by the mapping
    RuntimeContext context = new RuntimeContextImpl();

     //Set up output writer
    SATFileWriter writer = new SATFileWriter(_outputFile,"\r\n");

     //Get an ExposedServiceOperation
    ExposedServiceOperation inner = (ExposedServiceOperation)p.getRunTimeObject(
            p.getConfigFile(INNER_OPERATION));

     //Start by creating a SAX SAT Operator to call the mapping
    SAXSATServiceOperator sso = new SAXSATServiceOperator(inner);
     //Give the output writer to the SAXSATOpertor
    sso.setResultHandler(writer);

     //Create a GlobalSAXSATOperator to store data in the GLOBAL1 variable
    GlobalSAXSATOperator glob = new GlobalSAXSATOperator("GLOBAL1");
```

```
 //Create the map to store the keys and Operators
HashMap map = new HashMap();
 //GLOBAL1 will store the value in the SeqNum element
map.put("SeqNum", glob);
 //When the Body element is encountered the sso opertation will be called
map.put("Body", sso);

 //Create the Source and add the RuntimeContext
SAXSATSource source = new SAXSATSource(map);
source.setRuntimeContext(context);

 //Get the source data
ByteArrayInputStream stream = new ByteArrayInputStream(INPUT_DATA.getBytes());
source.setInputSource(stream);

 //Start the processesing
source.provide();

 //Get value of GLOBAL1
String out = (String)context.getUserObject("GLOBAL1");
```

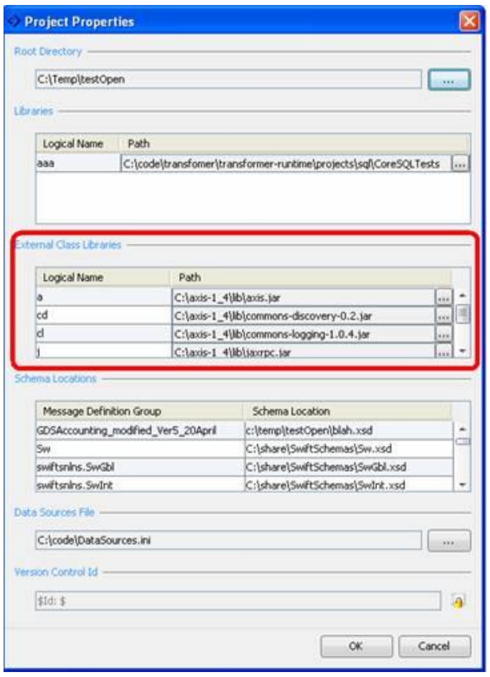
13 TBean API

Transformer provides a means of extending its functionality by allowing many of its tasks to be performed by TBeans. A TBean usually performs a well defined task like a Mapping action or a Decider function in a Condition. In addition to the large number of TBeans that are supplied with the product, Transformer can utilise any additional TBeans provided by the API programmer to the user of the GUI design tool.

A two stage approach is required when creating new TBeans. An XML configuration file (conventionally with a .tra suffix) containing the definition of the TBean must be created and placed in the correct place along with the other configuration files making up a Transformer project. The underlying Java class must also be available both at the design time and in the deployment.

A designated place for additional Java classes is provided within Transformer's project structure, viz. the project's `TBeans/classes` directory. Classes placed in here will be automatically loaded by Transformer without any explicit addition to the classpath, and since they are included in the built version of a project they will be automatically deployed at runtime along with the .tra files.

If additional jar files are required by the Transformer GUI they can be added to the Classpath by editing the "External Class Libraries" table on the project properties screen (see screen shot below). To bring up the Project Properties Dialog go to the "Project" menu and select "Properties".



Root Directory: C:\Temp\testOpen

Libraries

Logical Name	Path
aaa	C:\code\transfomer\transformer-runtime\projects\sql\CoreSQLTests

External Class Libraries

Logical Name	Path
a	C:\axis-1_4\lib\axis.jar
cd	C:\axis-1_4\lib\commons-discovery-0.2.jar
cl	C:\axis-1_4\lib\commons-logging-1.0.4.jar
j	C:\axis-1_4\lib\jaxrpc.jar

Schema Locations

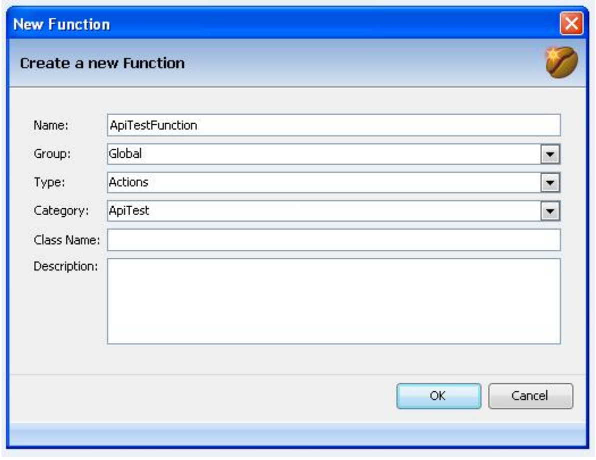
Message Definition Group	Schema Location
GDSAccounting_modified_Ver5_20April	c:\temp\testOpen\blah.xsd
Sw	C:\share\SwiftSchemas\Sw.xsd
swiftsnlns.SwGbl	C:\share\SwiftSchemas\SwGbl.xsd
swiftsnlns.SwInt	C:\share\SwiftSchemas\SwInt.xsd

Data Sources File: C:\code\DataSources.ini

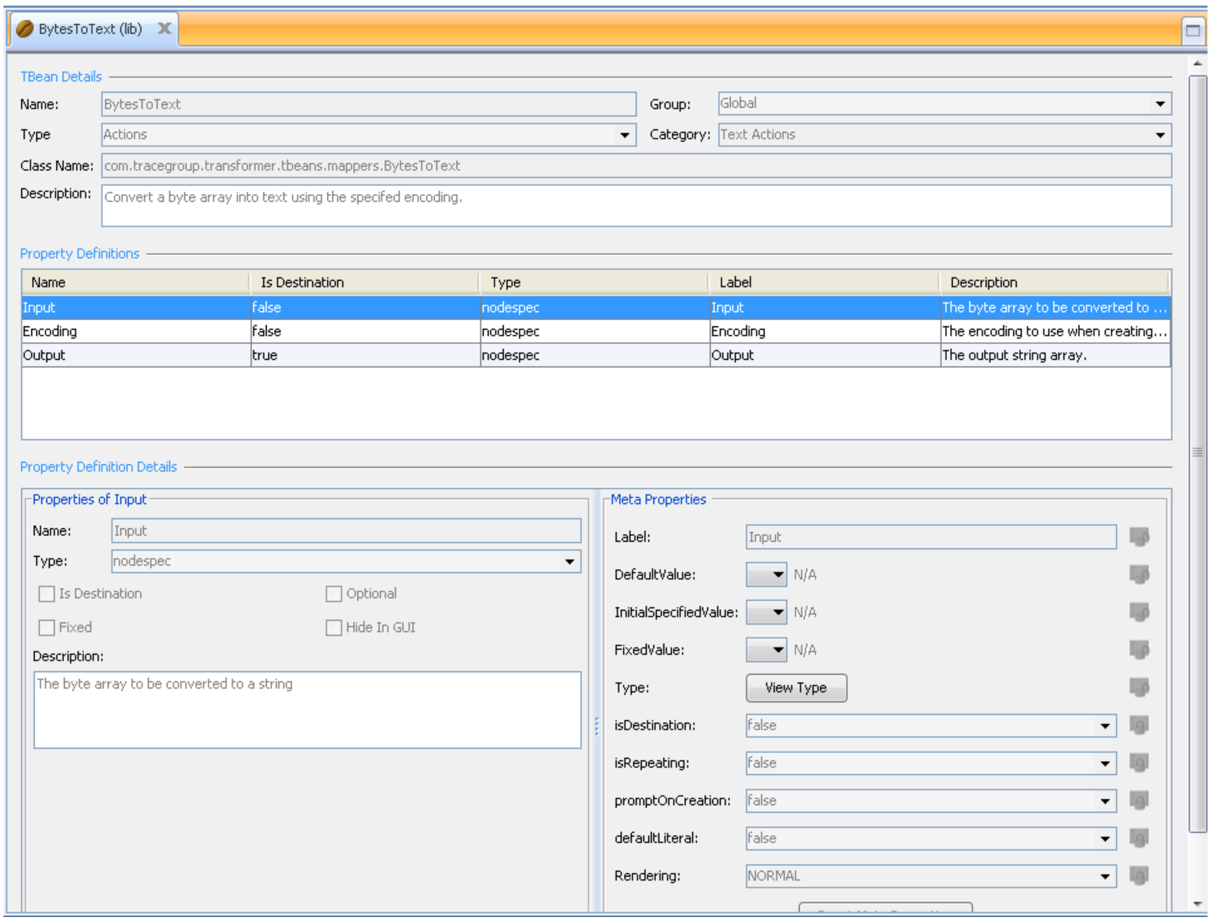
Version Control Id: $Id: $

Each additional jar file must be given a unique identifier prefix when it is added to the table. Please note that this additional Classpath information is only used by the Transformer GUI - at Runtime the jar files must still be added to the Classpath in the normal way.

The skeleton of the .tra file can be created via the “New Function” action from the Transformer GUI. A user must then add definitions of any properties that the TBean requires the user should specify when invoking it. This can be done via the GUI using the TBean screen:



Name:	ApiTestFunction
Group:	Global
Type:	Actions
Category:	ApiTest
Class Name:	
Description:	



BytesToText (lib)

TBean Details

Name:	BytesToText	Group:	Global
Type	Actions	Category:	Text Actions
Class Name:	com.tracegroup.transformer.tbeans.mappers.BytesToText		
Description:	Convert a byte array into text using the specifed encoding.		

Property Definitions

Name	Is Destination	Type	Label	Description
Input	false	nodespec	Input	The byte array to be converted to …
Encoding	false	nodespec	Encoding	The encoding to use when creating…
Output	true	nodespec	Output	The output string array.

Property Definition Details

Properties of Input

Name:	Input
Type:	nodespec

☐ Is Destination ☐ Optional

☐ Fixed ☐ Hide In GUI

Description:

The byte array to be converted to a string

Meta Properties

Label:	Input
DefaultValue:	N/A
InitialSpecifiedValue:	N/A
FixedValue:	N/A
Type:	View Type
isDestination:	false
isRepeating:	false
promptOnCreation:	false
defaultLiteral:	false
Rendering:	NORMAL

Property Definitions generally contain a

1. Name
   Use this field to enter of amend the name of the Property.

2. Type
   This field identifies the Type of the Property. Transformer identifies many standard types of property as well as more specific types of property used in specific situations. Each Property Type defines a set of Meta Properties that control the way in which a Property is defined. These Meta Properties vary by Property Type. For example, all Properties will have a Label Meta Property that controls the literal displayed to the user when the Function is used but a Property Type of Option also has the ability to define the list of values permitted for the Options.

3. Is Destination
   This field indicates whether the Property is viewed as a destination for a mapping action. Setting this will cause the Property to appear on the Output side of a mapping action.

4. Optional
   This field indicates whether the Property is mandatory or optional. When a Function is used, mandatory properties must be entered.

5. Fixed

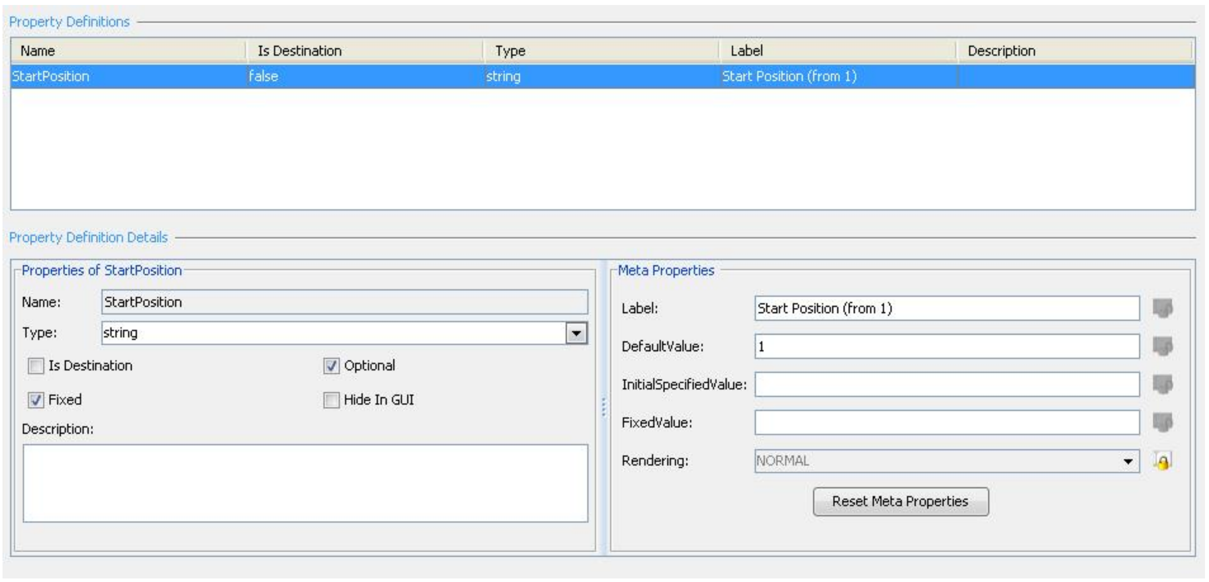
The Fixed field indicates whether the Property has a fixed value and will not be displayed when the Function is used.

6. Hide in GUI
   This field indicates that the Property should not be displayed in the GUI when the Function is used.

7. Description
   Use this field to enter a description of the Property. This is displayed when the Property is selected when the Function is used.

8. Meta Properties Panel
   This panel is used to display any meta properties specific to the selected type.

These can be configured in the Tbean view:



Name	Is Destination	Type	Label	Description
StartPosition	false	string	Start Position (from 1)	

Name: StartPosition
Type: string
☐ Is Destination ☑ Optional
☑ Fixed ☐ Hide In GUI
Description:

Label: Start Position (from 1)
DefaultValue: 1
InitialSpecifiedValue:
FixedValue:
Rendering: NORMAL

Which would look like the following in the tbean .tra file:

```
<PropertyDefinition name="StartPosition" type="integer" fixed="false" optional="false">
   <MetaProperty name="Label" type="string">
      <Value>Start Position (from 1)</Value>
   </MetaProperty>
   <MetaProperty name="DefaultValue" type="integer">
      <Value>1</Value>
   </MetaProperty>
</PropertyDefinition>
```

13.1 Mappers

A Mapper function is used in Transformer to perform a specific task inside Mapping Definitions.

There are a number of sub-classes to the mapper TBeans whose use is determined by exactly what sorts of Objects need to be mapped.

13.1.1 StringMapperFunction

The String Mapper function is used when both the input and the output of the mapping action are String Objects. Typical examples of this type of function are those that appear in the text category on the Action Selector screen.

To create a Java class that extends StringMapperFunction the user must provide one method – the signature of which is given below.

```
        public String mapString( FunctionCall owner,
                                 Map properties,
                                 ActionRuntime runtime,
                                 Object input )
                throws MomNodeException, SkipAction, ActionException
```

The input (assumed to be a String) is the data that is being passed into the action and the returned String is the output from the function.

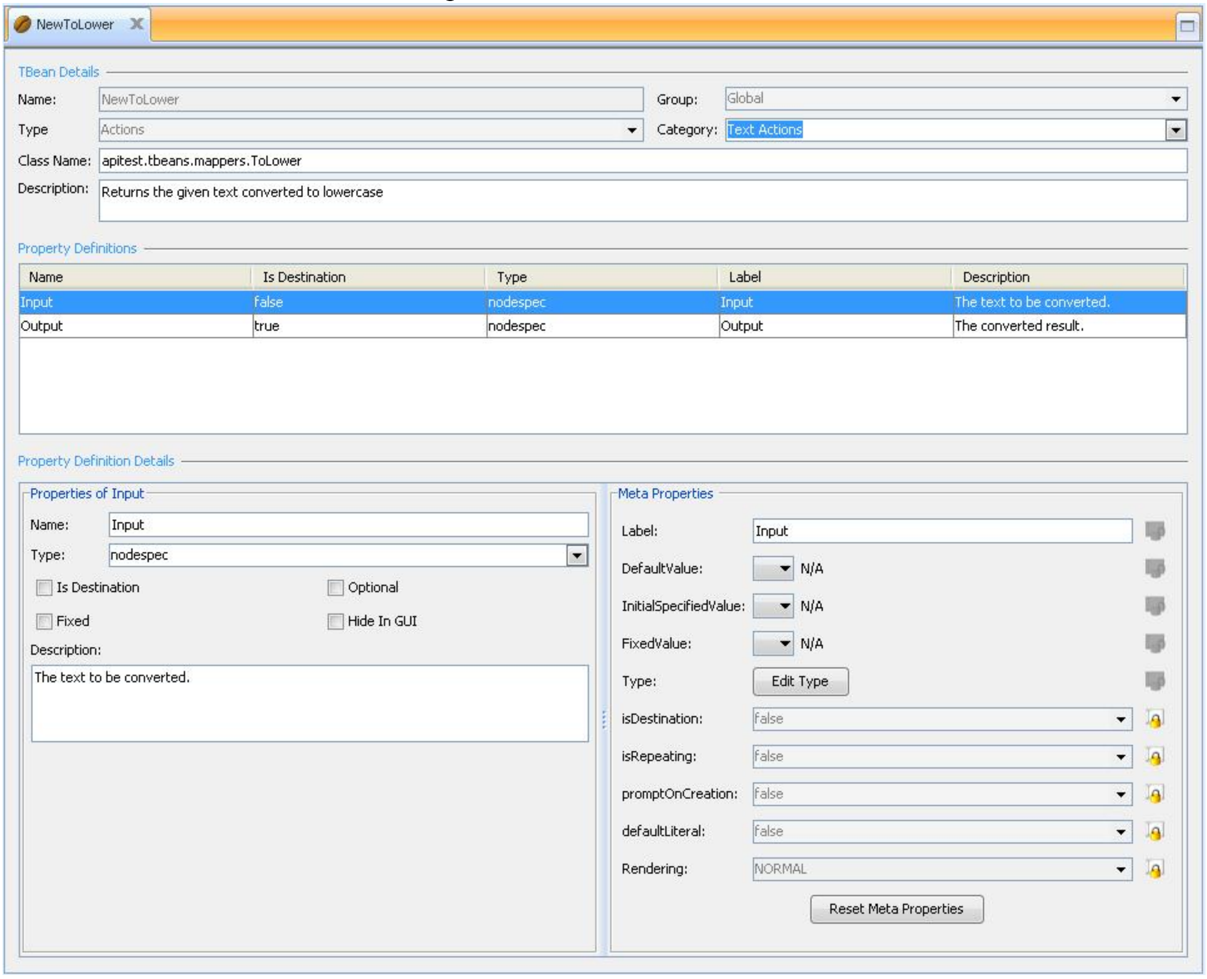
If any properties need to be retrieved from the .tra file then the Map properties can be used to do the look up. The key in the map is name of the property taken from the .tra file.

The ActionRuntime contains those things that are scoped to each invocation of a mapping i.e. a RepetitionVariableManager, a set of temporary variables and any “global” variables.

13.1.1.1 *ToLower Mapper Example*

The ToLower mapper takes a String and converts it to its lower case equivalent.

The ToLower .tra file looks something like this:


NewToLower

TBean Details

Name:	NewToLower	Group:	Global
Type	Actions	Category:	Text Actions
Class Name:	apitest.tbeans.mappers.ToLower		
Description:	Returns the given text converted to lowercase		

Property Definitions

Name	Is Destination	Type	Label	Description
Input	false	nodespec	Input	The text to be converted.
Output	true	nodespec	Output	The converted result.

Property Definition Details

Properties of Input

Name:	Input
Type:	nodespec

☐ Is Destination ☐ Optional

☐ Fixed ☐ Hide In GUI

Description:

The text to be converted.

Meta Properties

Label:	Input
DefaultValue:	N/A
InitialSpecifiedValue:	N/A
FixedValue:	N/A
Type:	Edit Type
isDestination:	false
isRepeating:	false
promptOnCreation:	false
defaultLiteral:	false
Rendering:	NORMAL

```
<ActionInfo group="Global" name="NewToLower" category="Text Actions"
javaClassType="MapperFunction">
    <PropertyDefinition name="Input" type="nodespec">
        <Description>The text to be converted.</Description>
        <MetaProperty name="Label" type="string">
            <Value>Input</Value>
        </MetaProperty>
        <MetaProperty name="Type" type="nodetype">
            <Value>
                <NodeType>
                    <Bot>String</Bot>
                </NodeType>
            </Value>
        </MetaProperty>
    </PropertyDefinition>
    <PropertyDefinition name="Output" type="nodespec">
        <Description>The converted result.</Description>
        <MetaProperty name="Label" type="string">
            <Value>Output</Value>
        </MetaProperty>
        <MetaProperty name="isDestination" type="boolean">
            <Value>TRUE</Value>
        </MetaProperty>
        <MetaProperty name="Type" type="nodetype">
            <Value>
                <NodeType>
                    <Bot>String</Bot>
                </NodeType>
            </Value>
        </MetaProperty>
    </PropertyDefinition>
    <Description>Returns the given text converted to lowercase</Description>
    <JavaClassName>apitest.tbeans.mappers.ToLower</JavaClassName>
</ActionInfo>
```

The corresponding Java class:

```
package apitest.tbeans.mappers;

import com.tracegroup.transformer.tbeans.*;
import com.tracegroup.transformer.dad.*;

import java.util.*;


public class ToLower extends StringMapperFunction
{
  public String mapString( FunctionCall owner,
                           Map properties,
                           ActionRuntime runtime,
                           Object input )
  {
    if ( input == null || input == NodeSpec.ABSENT_VALUE) {
      return "";
    } else {
      return ((String)input).toLowerCase();
    }
  }

}
```

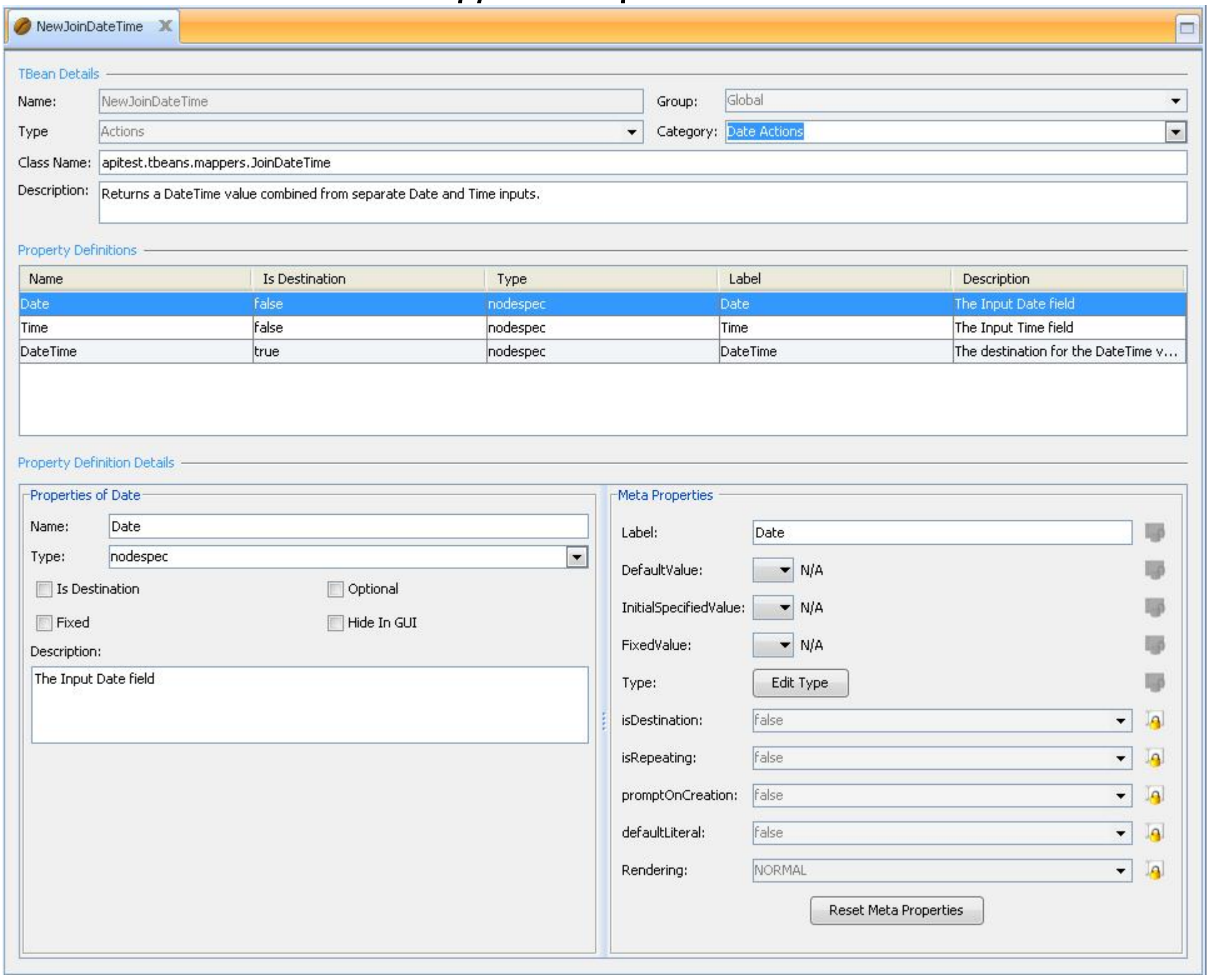
13.1.2 ObjectMapperFunction

The Object Mapper function is used when both the input and the output of the mapping action are things other than String Objects.

To create a Java class that extends ObjectMapperFunction the user must provide one method – the signature of which is given below.

```
       public Object mapObject( FunctionCall owner,
                                Map properties,
                                ActionRuntime runtime,
                                Object input )
               throws MomNodeException, SkipAction, ActionException
```

13.1.2.1 JoinDateTime Mapper Example


NewJoinDateTime

TBean Details

Name: NewJoinDateTime
Group: Global
Type: Actions
Category: Date Actions
Class Name: apitest.tbeans.mappers.JoinDateTime
Description: Returns a DateTime value combined from separate Date and Time inputs.

Property Definitions

Name	Is Destination	Type	Label	Description
Date	false	nodespec	Date	The Input Date field
Time	false	nodespec	Time	The Input Time field
DateTime	true	nodespec	DateTime	The destination for the DateTime v...

Property Definition Details

Properties of Date

Name: Date
Type: nodespec
Is Destination: ☐
Optional: ☐
Fixed: ☐
Hide In GUI: ☐
Description: The Input Date field

Meta Properties

Label: Date
DefaultValue: N/A
InitialSpecifiedValue: N/A
FixedValue: N/A
Type: Edit Type
isDestination: false
isRepeating: false
promptOnCreation: false
defaultLiteral: false
Rendering: NORMAL

```
<ActionInfo group="Global" name="NewJoinDateTime" category="Date Actions"
javaClassType="MapperFunction">
    <PropertyDefinition name="Date" type="nodespec">
        <Description>The Input Date field</Description>
        <MetaProperty name="Label" type="string">
            <Value>Date</Value>
        </MetaProperty>
        <MetaProperty name="Type" type="nodetype">
            <Value>
                <NodeType>
                    <Bot>Date</Bot>
                </NodeType>
            </Value>
        </MetaProperty>
    </PropertyDefinition>
    <PropertyDefinition name="Time" type="nodespec">
        <Description>The Input Time field</Description>
        <MetaProperty name="Label" type="string">
            <Value>Time</Value>
        </MetaProperty>
        <MetaProperty name="Type" type="nodetype">
            <Value>
                <NodeType>
                    <Bot>Time</Bot>
                </NodeType>
            </Value>
        </MetaProperty>
    </PropertyDefinition>
    <PropertyDefinition name="DateTime" type="nodespec">
        <Description>The destination for the DateTime value.</Description>
        <MetaProperty name="Label" type="string">
            <Value>DateTime</Value>
        </MetaProperty>
        <MetaProperty name="Type" type="nodetype">
            <Value>
                <NodeType>
                    <Bot>DateTime</Bot>
                </NodeType>
            </Value>
        </MetaProperty>
        <MetaProperty name="isDestination" type="boolean">
            <Value>TRUE</Value>
        </MetaProperty>
    </PropertyDefinition>
    <Description>Returns a DateTime value combined from separate Date and Time
inputs.</Description>
    <JavaClassName>apitest.tbeans.mappers.JoinDateTime</JavaClassName>
</ActionInfo>
```

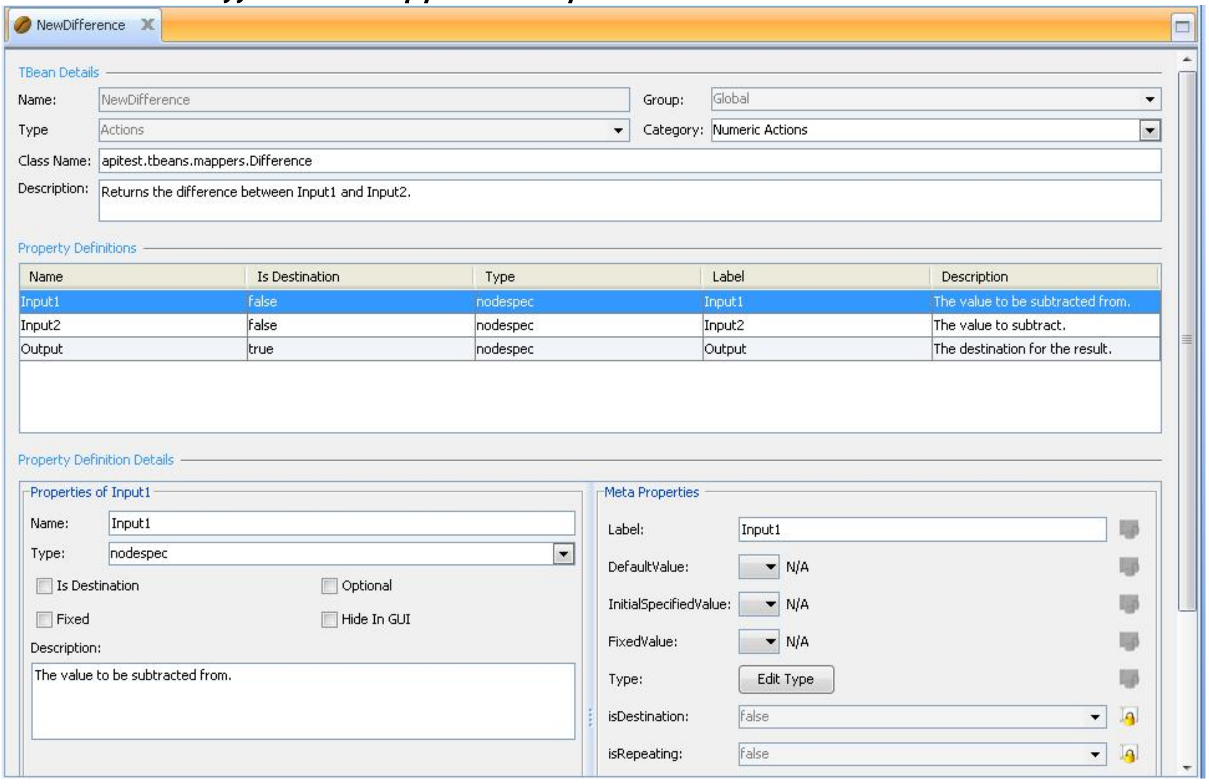
The corresponding Java class:

```
package apitest.tbeans.mappers;

import com.tracegroup.transformer.tbeans.*;
import com.tracegroup.transformer.mom.bots.*;
import com.tracegroup.transformer.dad.*;

import java.util.*;

/**
 * JoinDateTime updates a DateTime field from a Date field and a Time field.
 * It requires three properties:<OL>
 * <LI>a NodeSpec of type DateTime
 * <LI>a NodeSpec of type Date, which can be used as a destination
 * <LI>a NodeSpec of type Time, which can be used as a destination
 * </OL>
 */
public class JoinDateTime extends ObjectMapperFunction
{
  public Object mapObject(FunctionCall owner, Map properties,
                          ActionRuntime runtime, Object input)
  {

   Object[] inputArray = (Object[])input;
     // First I have to get the original date & time. (These are both
     // subclasses of Java Date).
    DateWithoutTime inDate = (DateWithoutTime)inputArray[0];
    TimeWithoutDate inTime = (TimeWithoutDate)inputArray[1];

    if (inDate == null || inTime == null) return null;

     // Then I have to construct 2 GregorianCalendars & initialise them (separately)
    // to these values.
    GregorianCalendar dc = new GregorianCalendar();
    dc.setTime( inDate );
    GregorianCalendar tc = new GregorianCalendar();
    tc.setTime( inTime );

     // Now I can extract the 3 interesting fields from time & set them in date.
    dc.set( Calendar.HOUR, tc.get( Calendar.HOUR_OF_DAY ) ); // NOT 'HOUR' which uses 12-hour
clock!!
    dc.set( Calendar.MINUTE, tc.get( Calendar.MINUTE ) );
    dc.set( Calendar.SECOND, tc.get( Calendar.SECOND ) );

     // Finally I can construct a new Date from the modified GregorianCalendar.
    return dc.getTime();
  }
}
```

13.1.3 NumericMapperFunction

NumericMapperFunction is an extension of the ObjectMapperFunction which has some extra methods to manipulate integers and decimals.

13.1.3.1 Difference Mapper Example



Name: NewDifference
Group: Global
Type: Actions
Category: Numeric Actions
Class Name: apitest.tbeans.mappers.Difference
Description: Returns the difference between Input1 and Input2.

Name	Is Destination	Type	Label	Description
Input1	false	nodespec	Input1	The value to be subtracted from.
Input2	false	nodespec	Input2	The value to subtract.
Output	true	nodespec	Output	The destination for the result.

Name: Input1
Type: nodespec
Is Destination
Optional
Fixed
Hide In GUI
Description: The value to be subtracted from.

Label: Input1
DefaultValue: N/A
InitialSpecifiedValue: N/A
FixedValue: N/A
Type:
isDestination: false
isRepeating: false

```
<?xml version="1.0"?>
<ActionInfo group="Global" name="NewDifference" category="Numeric Actions"
javaClassType="MapperFunction">
    <PropertyDefinition name="Input1" type="nodespec">
        <Description>The value to be subtracted from.</Description>
        <MetaProperty name="Label" type="string">
            <Value>Input1</Value>
        </MetaProperty>
        <MetaProperty name="Type" type="nodetype">
            <Value>
                <NodeType>
                    <Bot>AnySimple</Bot>
                </NodeType>
            </Value>
        </MetaProperty>
    </PropertyDefinition>
    <PropertyDefinition name="Input2" type="nodespec">
        <Description>The value to subtract.</Description>
        <MetaProperty name="Label" type="string">
            <Value>Input2</Value>
        </MetaProperty>
        <MetaProperty name="Type" type="nodetype">
            <Value>
                <NodeType>
                    <Bot>AnySimple</Bot>
                </NodeType>
            </Value>
        </MetaProperty>
    </PropertyDefinition>
    <PropertyDefinition name="Output" type="nodespec">
        <Description>The destination for the result.</Description>
        <MetaProperty name="Label" type="string">
            <Value>Output</Value>
        </MetaProperty>
        <MetaProperty name="isDestination" type="boolean">
            <Value>TRUE</Value>
        </MetaProperty>
        <MetaProperty name="Type" type="nodetype">
            <Value>
                <NodeType>
                    <Bot>AnySimple</Bot>
                </NodeType>
            </Value>
        </MetaProperty>
    </PropertyDefinition>
    <Description>Returns the difference between Input1 and Input2.</Description>
    <JavaClassName>apitest.tbeans.mappers.Difference</JavaClassName>
</ActionInfo>
```

The corresponding Java class:

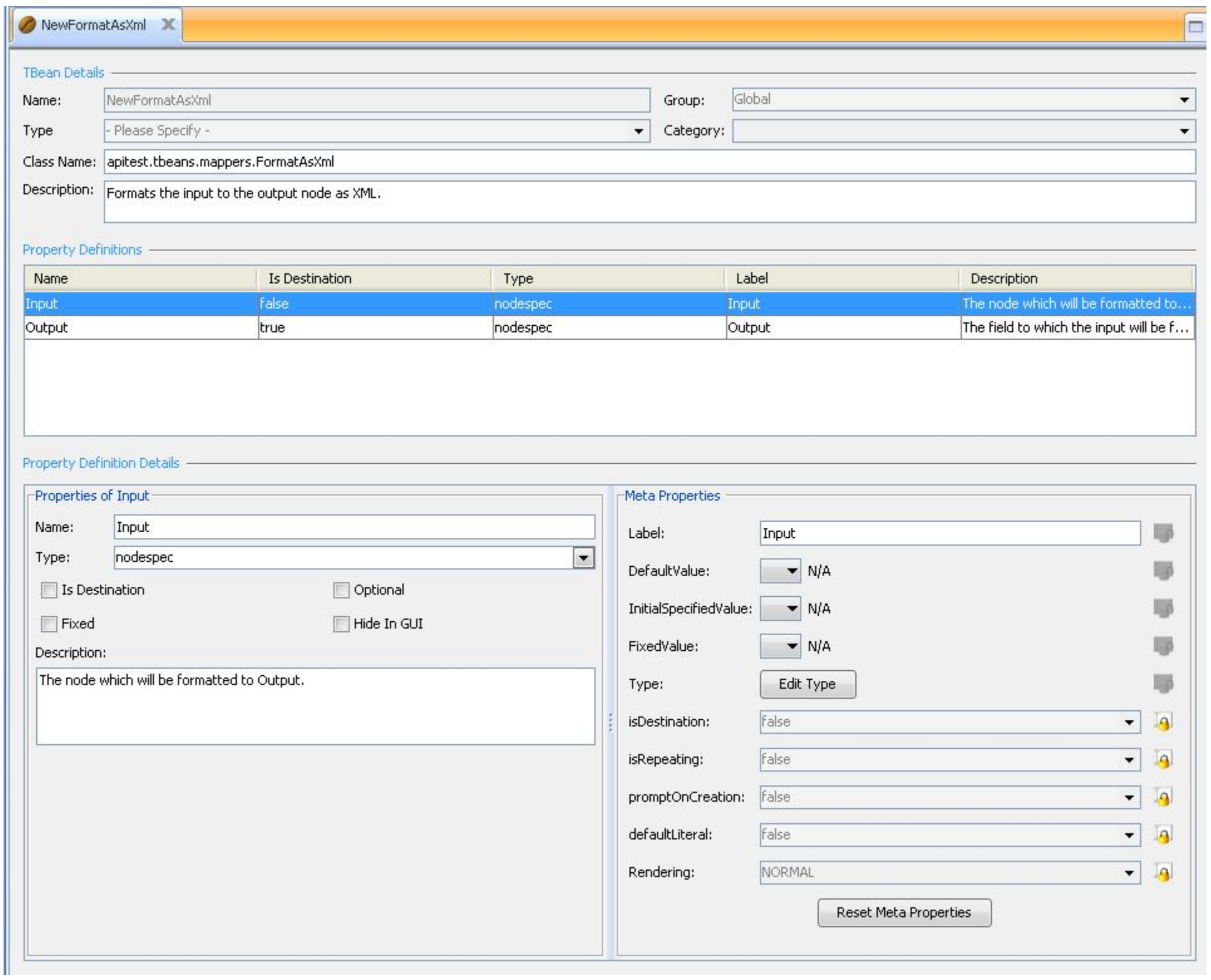
```
package apitest.tbeans.mappers;

import com.tracegroup.transformer.tbeans.*;
import com.tracegroup.transformer.mom.*;
import com.tracegroup.transformer.dad.*;

import java.util.*;


public class Difference extends NumericMapperFunction
{
  public Object mapObject( FunctionCall owner,
                           Map properties,
                           ActionRuntime runtime,
                           Object inputObject )
      throws MomNodeException, ActionException, SkipAction
  {
    Object inParam[] = (Object [])inputObject;

    if ( resultIsDecimal( inParam ) ) {
      return paramAsDecimal( inParam[0] ).subtract( paramAsDecimal( inParam[1] ) );
    } else {
      return paramAsInteger( inParam[0] ).subtract( paramAsInteger( inParam[1] ) );
    }
  }
}
```

13.1.4 NodeMapperFunction

Node Mapper functions are used when the mapper has a single input and output that are both NodeSpecs. The method that must be provided has the following signature:

```
        public void mapNode( FunctionCall owner,
                             Map properties,
                             ActionRuntime runtime,
                             MomNode input,
                             MomNode output)
                       throws MomNodeException, SkipAction, ActionException
```

13.1.4.1 FormatAsXml Mapper Example


NewFormatAsXml

TBean Details

Name:	NewFormatAsXml	Group:	Global
Type	- Please Specify -	Category:	
Class Name:	apitest.tbeans.mappers.FormatAsXml		
Description:	Formats the input to the output node as XML.		

Property Definitions

Name	Is Destination	Type	Label	Description
Input	false	nodespec	Input	The node which will be formatted to...
Output	true	nodespec	Output	The field to which the input will be f...

Property Definition Details

Properties of Input

Name: Input

Type: nodespec

☐ Is Destination ☐ Optional

☐ Fixed ☐ Hide In GUI

Description:

The node which will be formatted to Output.

Meta Properties

Label:	Input
DefaultValue:	N/A
InitialSpecifiedValue:	N/A
FixedValue:	N/A
Type:	Edit Type
isDestination:	false
isRepeating:	false
promptOnCreation:	false
defaultLiteral:	false
Rendering:	NORMAL

```
<ActionInfo group="Global" name="NewFormatAsXml" category="Advanced Actions"
javaClassType="MapperFunction">
    <PropertyDefinition name="Input" type="nodespec">
        <Description>The node which will be formatted to Output.</Description>
        <MetaProperty name="Label" type="string">
            <Value>Input</Value>
        </MetaProperty>
    </PropertyDefinition>

    <PropertyDefinition name="Output" type="nodespec">
        <Description>The field to which the input will be formatted.</Description>
        <MetaProperty name="Label" type="string">
            <Value>Output</Value>
        </MetaProperty>
        <MetaProperty name="isDestination" type="boolean">
            <Value>TRUE</Value>
        </MetaProperty>
        <MetaProperty name="Type" type="nodetype">
            <Value>
```

```
                <NodeType>
                    <Bot>string</Bot>
                </NodeType>
            </Value>
        </MetaProperty>
    </PropertyDefinition>
    <Description>Formats the input to the output node as XML.</Description>
    <JavaClassName>apitest.tbeans.mappers.FormatAsXml</JavaClassName>
</ActionInfo>
```

The corresponding Java class:

```
package apitest.tbeans.mappers;

import com.tracegroup.transformer.tbeans.*;
import java.util.Map;
import com.tracegroup.transformer.dad.*;
import com.tracegroup.transformer.mom.*;
import com.tracegroup.transformer.util.XMLSerializer;
import com.tracegroup.transformer.configfiles.ConfigObject;

/**
 * <code>FormatAsXML</code> takes the XML representation of the input node, including
 * all the markup structure, and sets it as the string value of the output node.  The .tra
 * file must specify that the output is <string>.
 */
public class FormatAsXml extends NodeMapperFunction {


    public void validateConfiguration(ConfigObject owner, TBeanConfiguration configuration) {
    Object o = configuration.getPropertyValue(INPUT);

    if ( o == null || !(o instanceof NodeSpec)) {
      configuration.setReferenceUnusable(INPUT +" Must Be Specified");
    }

    NodeSpec ns = (NodeSpec)o;
    DadNode dad = ns.dgetDadNode();

    if (dad.getDefiningGroup() == null) {
      configuration.setPropertyUnusable(INPUT,"The input cannot be a simple BOT");
    }

  }


  public void mapNode(FunctionCall owner,
                      Map properties,
                      ActionRuntime runtime,
                      MomNode input,
                      MomNode output) throws MomNodeException, SkipAction, ActionException {

    try {
      String s = input.formatAsXML(XMLSerializer.SERIALIZE_FORMATTED);
      SimpleNode sn = (SimpleNode)output; // rely on .tra file
      sn.setStringValue(s);
    }
    catch (WriteMessageContentException ex) {
        // Unexpected - only if SAX not set up properly in the JDK environment or something
      // like that.
      Throwable t = ex.getCause();
      if (t != null) {
        throw new RuntimeException(t);
      } else {
        throw new RuntimeException(ex);
      }
    }
  }

  private static final String INPUT = "Input";

}
```

13.1.5 NodeSpecs

Mapper TBeans often have to interact with input or output properties that are defined as being of the NodeSpec type. A NodeSpec is a property type that allows the user to address a field from the input or output tree of a mapping (or a temporary variable, a value generator, a literal etc.). NodeSpecs have already featured in some of the examples earlier in this section but are fully covered here.

It is common practice to set the value of a NodeSpec in the setProperties method of a mapper and then use it in the doFunction (or mapNode method etc.). See example below taken from the remove node mapper.

```
private static final String OUTPUT_NAME = "Output";

private NodeSpec _outputNS;

@Override
public void setProperties(ConfigObject owner,
                          Map properties,
                          TBeanConfiguration configuration) {
  _outputNS = (NodeSpec)properties.get(OUTPUT_NAME);
}

@Override
public void doFunction( FunctionCall owner,
                        Map properties,
                        ActionRuntime runtime )
        throws MomNodeException, DataStateException {
  _outputNS.rremoveNode(runtime, ExceptionMask.MaskEverything);
}
```

The ActionRuntime class has had a number of convenience methods added to it that can be used to retrieve information from NodeSpecs. The following methods are available.

- String getStringFromNodeSpec(NodeSpec mns) – returns the String value from the NodeSpec
- Object getObjectFromNodeSpec(NodeSpec mns) – returns the Object value from the NodeSpec.

These mappers are fairly simple wrappers around methods on NodeSpec itself. These methods can also be called from a mapper (or a decider – see next section). The nodespec methods that should be used at runtime are all prefixed with the letter “r” and many require the ActionRuntime to be passed into them.

- MomNode rgetNode(ActionRuntime runtime) – this method returns the MomNode address by the NodeSpec, if the NodeSpec contains a path and the target of the path is not present then the method will return null.
- Object rgetSource(ActionRuntime runtime) – this returns the source node of the Nodespec. So if the Nodespec is addressing a single node then it will return a Momnode and if the Nodespec is address a repeating set of nodes (i.e. the path string ends []) then the method returns an IChildnodeset. This method also takes into account the exception scenarios of

the the parent action so it may throw an Exception, or just ignore the result depending on how the Action Exception Rules have been set up for this action.

- void rremoveNode(ActionRuntime runtime) – removes the specified node from its parent
- void rremoveNode(ActionRuntime runtime, ExceptionMask mask) – removes the specified node from its parent taking into account the ExceptionMask
- MomNode rgetOrCreateNode(ActionRuntime runtime) – this will retrieve the addressed node or create it if it is not present. This is usually used on destination addresses to create them if they are not already there.
- IChildNodeSet rgetCNS(ActionRuntime runtime) – this returns the addressed ChildNodeSet (set of repeats)
- IChildNodeSet rgetOrCreateCNS(ActionRuntime runtime) – this returns the addresses ChildNodeSet (set of repeats) or create a new one if not present.
- Object rgetObject(ActionRuntime runtime) - return the value associated with this NodeSpec as a Business Object. This method takes account of the AER rules associated with this node. If this NodeSpec defines a Path, and the path does not address a node which is present, this method will do one of the following, depending on the rules, and on whether or not a default value exists:
    - return a default value
    - return the special value ABSENT_VALUE
    - throw an ActionException

  Similarly, if the path addresses a node which has a BOT Construction exception, it will do one of the following, depending on the rules, and on whether or not a default value exists:
    - return a default value
    - return the special value NOBOT_VALUE
    - throw an ActionException
- String rgetString(ActionRuntime runtime) - return the value associated with this NodeSpec as a String. This method takes account of the AER rules associated with this node – see the previous point for exception scenarios.
- SimpleNode rgetSimpleNode(ActionRuntime runtime) - this method returns a SimpleNode corresponding to this NodeSpec. In normal situations this is the same as the node returned by rgetNode but if there has been a dynamic type override or substitution to a "simple content plus attributes" this method will return the simple content node.

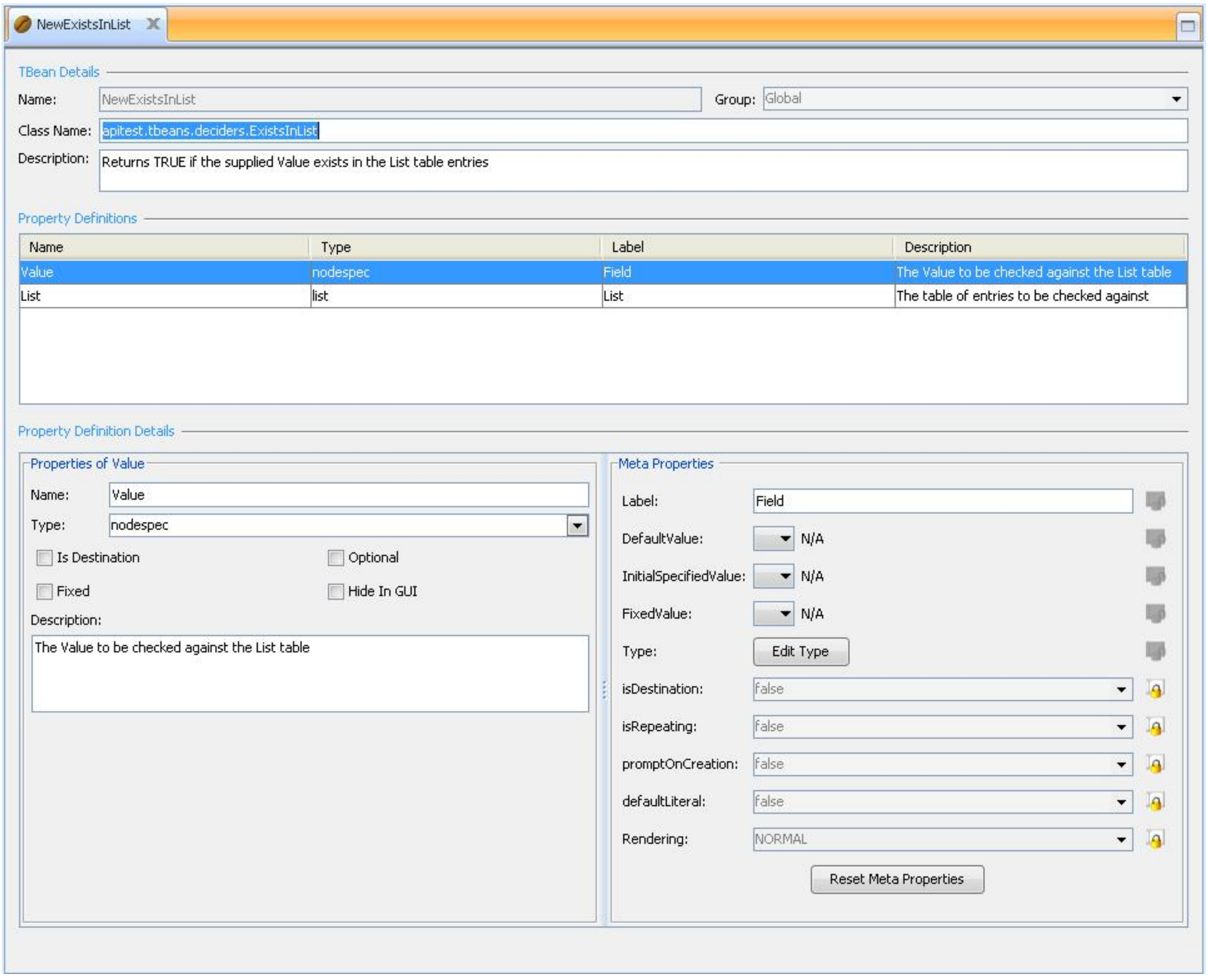
13.2 Deciders

Deciders are used inside of conditions to perform conditional logic on data items. A Java class that is extending Decider must provide the following method:

```
public boolean decide(Condition owner, Map properties, ActionRuntime runtime)
        throws ActionException, MomNodeException, SkipAction
```

13.2.1 ExistsInList Decider Example

The Decider will check to see if the value is in a specified list.


NewExistsInList

TBean Details

Name: NewExistsInList
Group: Global
Class Name: apitest.tbeans.deciders.ExistsInList
Description: Returns TRUE if the supplied Value exists in the List table entries

Property Definitions

Name	Type	Label	Description
Value	nodespec	Field	The Value to be checked against the List table
List	list	List	The table of entries to be checked against

Property Definition Details

Properties of Value

Name: Value
Type: nodespec
☐ Is Destination ☐ Optional
☐ Fixed ☐ Hide In GUI
Description:
The Value to be checked against the List table

Meta Properties

Label:	Field
DefaultValue:	N/A
InitialSpecifiedValue:	N/A
FixedValue:	N/A
Type:	Edit Type
isDestination:	false
isRepeating:	false
promptOnCreation:	false
defaultLiteral:	false
Rendering:	NORMAL

Example .tra file

```
<DeciderInfo group="Global" name="NewExistsInList">
    <JavaClassName>apitest.tbeans.deciders.ExistsInList</JavaClassName>
    <PropertyDefinition name="Value" type="nodespec">
        <Description>The Value to be checked against the List table</Description>
        <MetaProperty name="Label" type="string">
            <Value>Field</Value>
        </MetaProperty>
    </PropertyDefinition>
    <PropertyDefinition name="List" type="list">
        <Description>The table of entries to be checked against</Description>
    </PropertyDefinition>
    <Description>Returns TRUE if the supplied Value exists in the List table
entries</Description>
 </DeciderInfo>
```

The corresponding Java class:

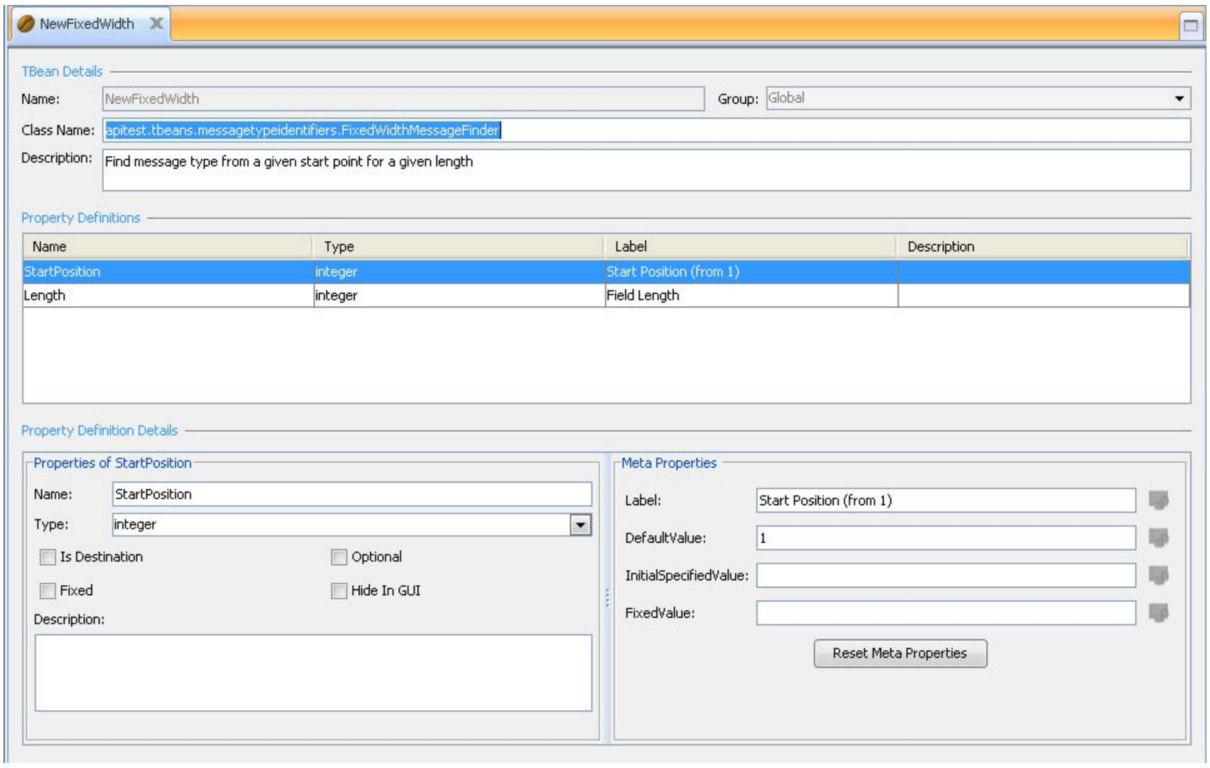
```
package apitest.tbeans.deciders;

import com.tracegroup.transformer.configfiles.ConfigObject;
import com.tracegroup.transformer.dad.*;
import com.tracegroup.transformer.mom.*;
import com.tracegroup.transformer.tbeans.Decider;
import com.tracegroup.transformer.tbeans.propertyeditors.ListPropertyValue;

import java.util.List;
import java.util.Map;

public class ExistsInList extends Decider {
  private NodeSpec _valueSpec;
  private List _list;

  public void setProperties(ConfigObject owner, Map properties,
                            TBeanConfiguration configuration) {
    _valueSpec = (NodeSpec)properties.get(VALUE_NAME);
    ListPropertyValue lpv = (ListPropertyValue)properties.get(LIST_NAME);
    _list = lpv.list;
  }

  public boolean decide(Condition owner,
                        Map properties,
                        ActionRuntime runtime)
    throws ActionException, MomNodeException, SkipAction {
    return _list.contains(_valueSpec.rgetString(runtime));
  }

  public static String VALUE_NAME = "Value";
  public static String LIST_NAME = "List";
}
```

13.3 MTIs

Message Type Identifiers (MTIs) are used to identify a Message from its content – they are the Transformer equivalent of Cloverleaf Trxid procs. An MTI must provide the following 3 methods:

```
public ConfigFile getMessageDef( MessageDefinitionGroup owner,
                                 Map properties,
                                 ReadableMessageContent content )
        throws UnrecognizedMessageException


public ConfigFile getMessageDef( MessageDefinitionGroup owner,
                                 Map properties,
                                 String content)
        throws UnrecognizedMessageException


public ConfigFile getMessageDef( MessageDefinitionGroup owner,
                                 Map properties,
                                 byte[] content)
        throws UnrecognizedMessageException
```

13.3.1 Fixed Width MTI Example


NewFixedWidth

TBean Details

Name: NewFixedWidth Group: Global

Class Name: apitest.tbeans.messagetypeidentifiers.FixedWidthMessageFinder

Description: Find message type from a given start point for a given length

Property Definitions

Name	Type	Label	Description
StartPosition	integer	Start Position (from 1)	
Length	integer	Field Length	

Property Definition Details

Properties of StartPosition

Name: StartPosition

Type: integer

☐ Is Destination ☐ Optional

☐ Fixed ☐ Hide In GUI

Description:

Meta Properties

Label: Start Position (from 1)

DefaultValue: 1

InitialSpecifiedValue:

FixedValue:

```
<MTIInfo name="NewFixedWidth" group="Global">
    <PropertyDefinition name="StartPosition" type="integer" fixed="false" optional="false">
        <MetaProperty name="Label" type="string">
            <Value>Start Position (from 1)</Value>
        </MetaProperty>
        <MetaProperty name="DefaultValue" type="string">
            <Value>1</Value>
        </MetaProperty>
    </PropertyDefinition>
    <PropertyDefinition name="Length" type="integer" fixed="false" optional="false">
        <MetaProperty name="Label" type="string">
            <Value>Field Length</Value>
        </MetaProperty>
    </PropertyDefinition>
        <Description>Find message type from a given start point for a given
length</Description>
        <JavaClassName>apitest.tbeans.messagetypeidentifiers.FixedWidthMessageFinder</JavaClas
sName>
        <Keywords>Test</Keywords>
</MTIInfo>
```

The corresponding Java class:

```
package apitest.tbeans.messagetypeidentifiers;

import java.util.*;


import com.tracegroup.transformer.configfiles.*;
import com.tracegroup.transformer.dad.*;
```

```java
import com.tracegroup.transformer.mom.*;
import com.tracegroup.transformer.tbeans.*;

/**
 * Extract the message name from a fixed place in the text.
 * Must supply "StartPosition" (from 1) and "Length" parameters
 *
 */
public class FixedWidthMessageFinder extends MessageTypeIdentifier {

  private static String NAME_START_POSITION = "StartPosition";
  private static String NAME_LENGTH = "Length";


   /**
    * @see TBean#validateConfiguration
    * @param owner - not used
    * @param configuration  - supplies the parameter
    */
  public void validateConfiguration(ConfigObject owner, TBeanConfiguration configuration)
  {
    Integer startPos = (Integer)configuration.getPropertyValue(NAME_START_POSITION);
    if (startPos == null) {
      configuration.setPropertyUnusable(NAME_START_POSITION, "not defined");
    } else {
      if (startPos.intValue() < 1) configuration.setPropertyUnusable(NAME_START_POSITION, "
must be greater then zero");
    }
    Integer length = (Integer)configuration.getPropertyValue(NAME_LENGTH);
    if (length == null) {
      configuration.setPropertyUnusable(NAME_LENGTH, "not defined");
    } else {
      if (length.intValue() < 1) configuration.setPropertyUnusable(NAME_LENGTH, " must be
greater then zero");
    }
  }
   /**
    * @see MessageTypeIdentifier#getMessageDef
    * @param owner - definition of MessageDefinitionGroup
    * @param properties - parameters
    * @param content - string or byte array contained in the message
    * @return the message definition to use
    * @throws UnrecognizedMessageException
    */
  public ConfigFile getMessageDef( MessageDefinitionGroup owner,
                                   Map properties,
                                   ReadableMessageContent content )
      throws UnrecognizedMessageException  {
    return doGetMessage( owner, properties, content );
  }

  public ConfigFile getMessageDef( MessageDefinitionGroup owner,
                                   Map properties,
                                   String content )
      throws UnrecognizedMessageException {
    return doGetMessage( owner, properties, new StringRMC( content ) );
  }
  public ConfigFile getMessageDef( MessageDefinitionGroup owner,
                                   Map properties,
                                   byte [] content )
      throws UnrecognizedMessageException {
    return doGetMessage( owner, properties, new ByteRMC_1( content ) );
  }

  private ConfigFile doGetMessage( MessageDefinitionGroup owner,
                                   Map properties,
                                   ReadableMessageContent content )
      throws UnrecognizedMessageException {
```

```
    Integer startPos = (Integer)properties.get(NAME_START_POSITION);
    if (startPos == null)
      throw new UnrecognizedMessageException("Parameter " + NAME_START_POSITION + " not
specified");
    Integer length    = (Integer)properties.get(NAME_LENGTH);
    if (length == null)
      throw new UnrecognizedMessageException("Parameter " + NAME_LENGTH + " not specified");
    int endPos = startPos.intValue() + length.intValue();
    if (endPos > content.getLimit())
      throw new UnrecognizedMessageException("Start + length exceeds message length of " +
content.getLimit());

    String msgType = null;

   try {
     msgType = content.peekString(startPos.intValue(), endPos);
   } catch ( java.nio.charset.CharacterCodingException e ) {
     throw new UnrecognizedMessageException(e.getMessage());
   }

   return getMessageDef(owner, msgType);
  }
}
```

13.4 Value Generators

Value Generators are used by transformer to produce information that is easily looked up e.g. the current time. Two methods must be provided by implementing classes:

```
      public Object generateValue()
```

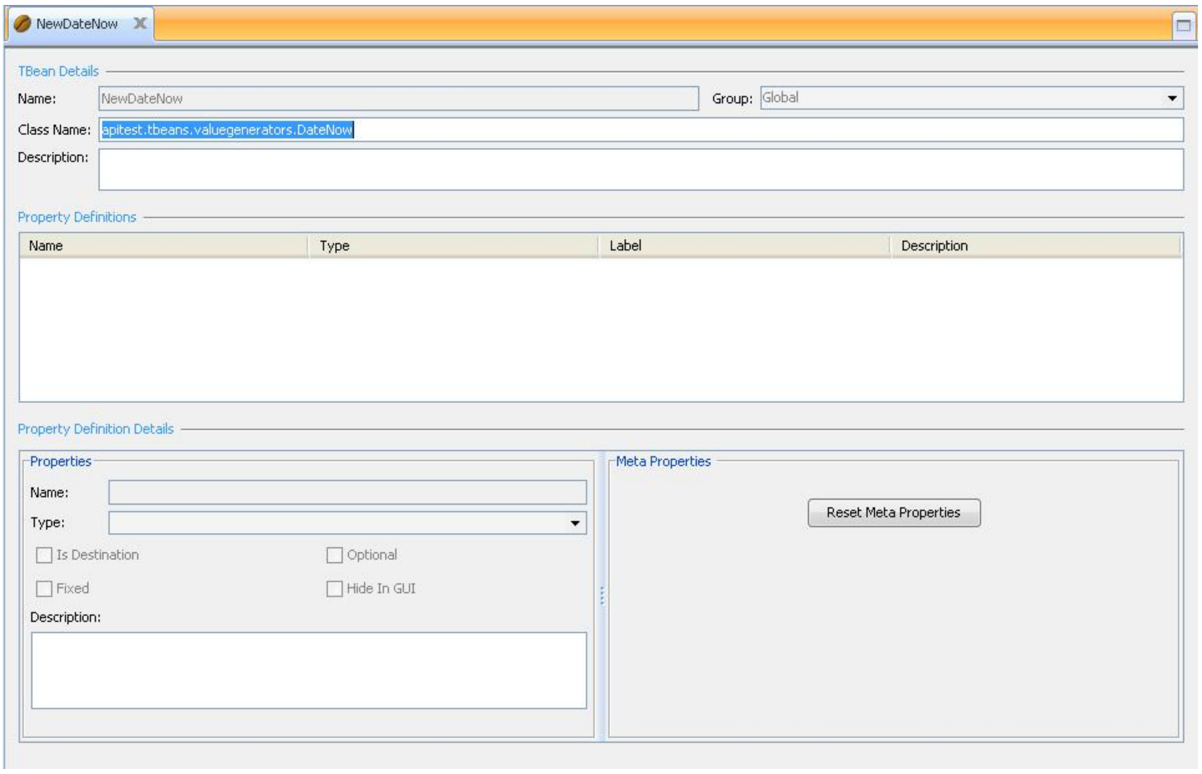
The generateValue returns the value of the information that is being provided

```
      public Class getObjectClass()
```

and the getObjectClass returns the type of Object that the information is stored in.

13.4.1 DateNow Value Generator Example

The NewDateNow generator returns the current Date. Its .tra file is detailed below:



Name: NewDateNow
Group: Global
Class Name: apitest.tbeans.valuegenerators.DateNow
Description:

Name	Type	Label	Description

Name:
Type:
Is Destination
Optional
Fixed
Hide In GUI
Description:

```
<ValueGeneratorInfo group="Global" name="NewDateNow">
    <JavaClassName>apitest.tbeans.valuegenerators.DateNow</JavaClassName>
 </ValueGeneratorInfo>
```

And the corresponding Java class:

```
package apitest.tbeans.valuegenerators;

import com.tracegroup.transformer.tbeans.*;
import com.tracegroup.transformer.mom.bots.*;

import java.util.*;

public class DateNow extends ValueGenerator {
  public Class getObjectClass()
  {
    return DateWithoutTime.class;
  }


  public Object generateValue()
  {
    return DateWithoutTime.createDate( new Date() );
  }
}
```

14 Lookup Tables

There are three situations in Transformer which have to interact with tables of values. The first is lookup tables in mappings where a given input value corresponds to a different output value. The second can be seen in code converters and the exists in list decider – these need to validate that a value exists as a key (or value) in a table. Finally there is the case used in the message edit facility in test data form view where an editor widget presents the user with a list of allowed values.

The interfaces in the `com.tracegroup.transformer.lookup` package represent the solutions to these three situations. From a user perspective they are only really likely to want to be able to write the first of these – a lookup table – we will concentrate on this item.

14.1 Lookup Interfaces

14.1.1 IKeyFinder

`IKeyFinder` is the base class for all Lookups. It has the methods for defining a property definition set and defines the type of the key used.

`Class<K> getKeyClass()` - returns the class of the key

`PropertyDefinitionSet getKeyAndValuePropertyDefinitionSet()` - creates a Property Definition Set representing the inputs and output properties used in the lookup. Typically this Property Definition Set will include an Input and an Output property.

`void resetCache()` - the results of a Lookup operation can be cached. This method is used to reset the cache.

14.1.2 ILookupKeys

`ILookupKeys` describes a Lookup typically used by a Converter. I.e. it checks to see if something is in a list of allowed values. Generally speaking this is the minimal implementation of IKeyFinder. In other words if something implements IKeyFinder it will at the very least implement ILookupKeys.

`boolean containsKey(K key)` - this method returns true if the underlying table contains the supplied key and false otherwise.

14.1.3 IKeyAndValueFinder

`IKeyAndValueFinder extends ILookupKeys` and provides information on the type of the value class.

`Class<V> getValueClass()` - returns the class of the value

14.1.4 ILookupKeysAndValues

`ILookupKeysAndValues extends IKeyAndValueFinder` provides the functionality that a lookup mapper will use - a value will be returned when a key is specified. It is this interface (and `ILookupKeys`) that a user will have to implement if they wish to write a Lookup provider TBean.

`V getValue(K key)` - this method returns a value object when given a specific key object i.e. it performs the look up on the underlying table.

The final interfaces are unlikely to be needed by a user but are provided for completeness.

14.1.5 IDiscoverKeys

`IDiscoverKeys` is used if the Lookup needs to return a list of the key values. This interface is implemented for the third use case – the message edit facility.

`Set<K> getKeys()` – returns the Set of keys for the underlying table

14.1.6 IDiscoverKeysAndValues

`IDiscoverKeysAndValues` can return a map representing all of the keys and values in the table.

`Map<K,V> getMap()` – returns a Map that represents all of the keys and values in the table.

14.1.7 IReverseLookupKeys

`IReverseLookupKeys` – this is used if the values in a table need to be queried

`boolean containsValue(V value)` – this method returns true if the underlying table contains the supplied value and false otherwise.

14.1.8 IReverseLookupKeysAndValues

`IReverseLookupKeysAndValues` – this class will be implemented if a “reverse” lookup is required. i.e. if a the user supplies a value its corresponding key will be returned.

`K getKey(V value)` - this method returns a key object when given a specific value object i.e. it performs the reverse look up on the underlying table.

14.2 LookupProviderTBean

`LookupProviderTBean` is the class that must be extended if the user wishes to create an external lookup table. The only method that has to be provided is the following:

`public ILookupKeys<Object> createLookup()` – this method provides a Lookup that will return a value given a key.

The implementation of `ILookupKeys` does the majority of the work. The `ILookupKeys` interface was described in the previous section.

14.3 Creating a Lookup TBean

To create a Lookup TBean the user must first write a Java Class which extends `LookupProviderTBean` (from the `com.tracegroup.transformer.tbeans` package).

The implementation of the `ILookupKey` interface does the real work. Simply implementing `ILookupKey` is fine if the TBean is only going to be used for Validation purposes i.e. a Value or Key is checked to see if it present in the underlying Map. However, if full lookup functionality is required then `ILookupKeysAndValues` should be implemented as well.

So a simple implementation with no user configurable Properties using a `HashMap` would look something like this:

```
public class ExternalLookup extends LookupProviderTBean {

  @Override
  public ILookupKeys<Object> createLookup() {
    return new ExternalMap(retrieveMap());
  }

  Map<String, String> retrieveMap() {
    //This requires a real implementation.
    return new HashMap<String, String> ();
  }


  class ExternalMap implements ILookupKeys, ILookupKeysAndValues {

    Map<String, String> _map;

    ExternalMap(Map map) {
      _map = map;
    }

    public boolean containsKey(Object key) {
      return _map.containsKey(key);
    }

    public Class getKeyClass() {
      return String.class;
    }

    public PropertyDefinitionSet getKeyAndValuePropertyDefinitionSet() {

      //This method has to define the properties that will appear on screen
      //for this lookup. Typically the user will want a single input and a
      //single output property. A Property has to have a type and this is
      //likely to be a nodespec which means that in the GUI the user can
      //fill in the details of the property with information from the
      //source, destination or temporary trees

      //Each Property must be added to the PropertyDefinitionSet which is
      //returned by the method.

      PropertyDefinitionSet pds = new PropertyDefinitionSet();

      //Set up Input Property with a NodeSpec type
      PropertyDefinition pd = new PropertyDefinition();
      pd.setName(new Name("Input"), null);
      pd.setType(new Name("nodespec"), null);
      //Add it to the PropertyDefinitionSet
      pds.addPropertyDefinition(pd, null);
      //Set up Output Property with a NodeSpec type
      pd = new PropertyDefinition();
      pd.setName(new Name("Output"), null);
      pd.setType(new Name("nodespec"), null);
      //Add an isDestination meta-property to indicate that the property
      //will appear as an output i.e. on the right hand side of the
      //property panel
      Property p = new Property();
      p.setName(new Name("isDestination"), null);
      p.setType(new Name("boolean"), null);
      p.setValue(true, null);
      pd.addMetaProperty(p, null);
      //Add it too
      pds.addPropertyDefinition(pd, null);

      return pds;
    }

    public void resetCache() {
      //We have included this method for future enhancement.
      //Whilst it can clear the cache there is no facility in the
      //Transformer Runtime
      //to call this method
      _map.clear();
    }

    public Object getValue(Object key) {
      return _map.get(key);
    }

    public Class getValueClass() {
      return String.class;
    }
  }

}
```

If you want your lookup to have multiple keys then you will need to set up and add additional inputs in the `getKeyAndValuePropertyDefinitionSet` method. E.g.

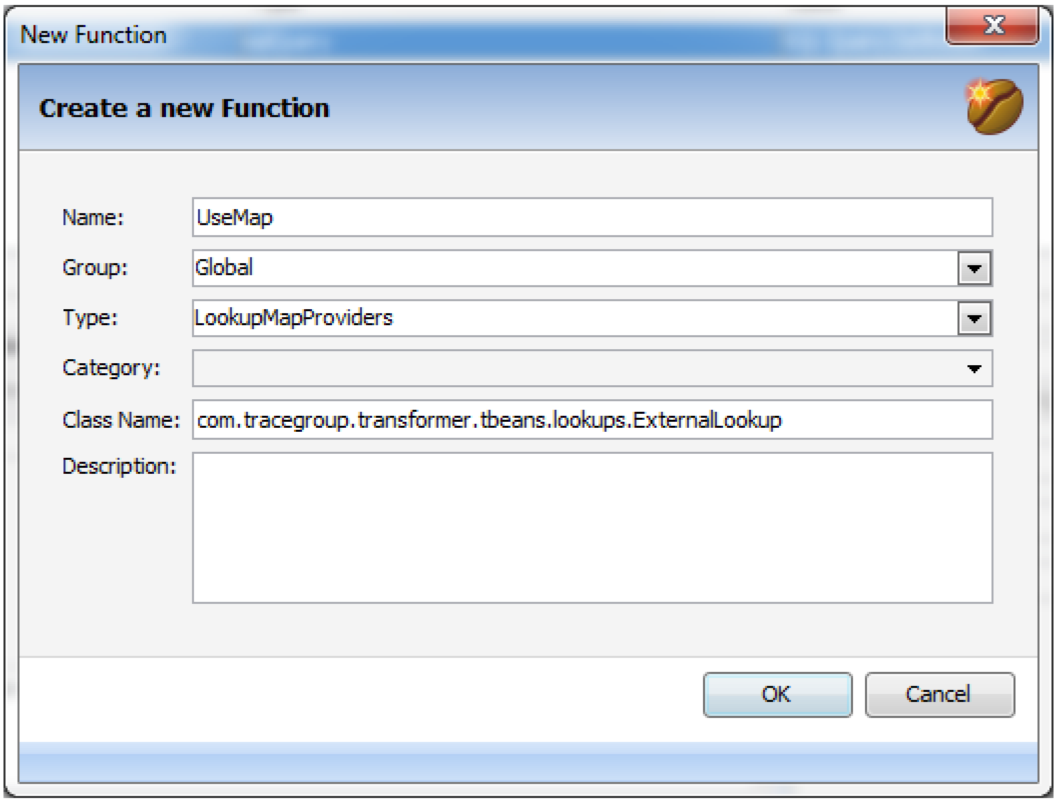
```
      //Set up a second Input Property with a NodeSpec type
      PropertyDefinition pd2 = new PropertyDefinition();
      pd2.setName(new Name("Input2"), null);
      pd2.setType(new Name("nodespec"), null);
      //Add it to the PropertyDefinitionSet
      pds.addPropertyDefinition(pd2, null);
```

In this case the key Object passed into the getValue method will be an array of Objects rather than a simple Object.

The class files should be added to the required project by adding them to the “Java Class Libraries” Table in the Project Properties Dialog. To access this choose the “Project Properties” action from the Project menu.

To add the TBean tra file that references the Java class do the following. Open the “Entire Project” tab in the Navigator (right click at the top of the Navigator and select New Tab -> Entire Project). Expand the tree and Navigate to Global/Functions/LookupMapProviders. Select the LookupMapProviders folder right click and select “New LookupProviders”.

In the resulting dialog fill in the Name of the Provider and the fully qualified class name.

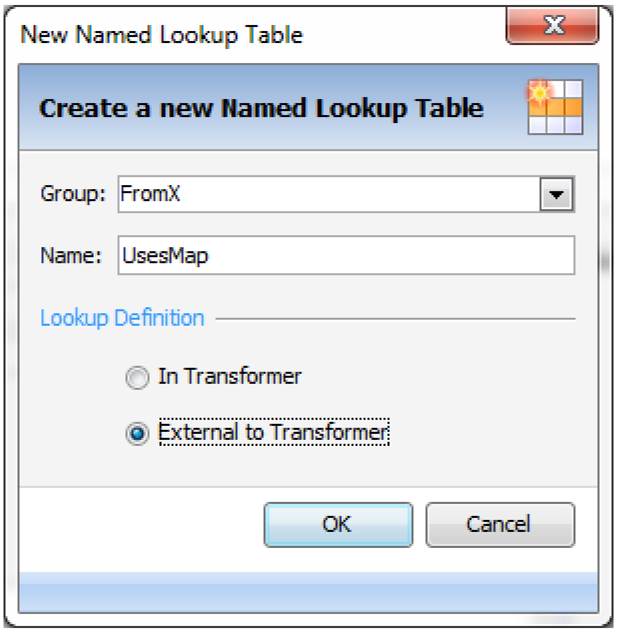

New Function

Create a new Function

Name:	UseMap
Group:	Global
Type:	LookupMapProviders
Category:	
Class Name:	com.tracegroup.transformer.tbeans.lookups.ExternalLookup
Description:	

Save the resulting Config file.

To use the new provider in a mapping group - add a New Lookup Table to the mapping Group. Give it a Name and choose “External To Transformer” as the Lookup definition:


New Named Lookup Table

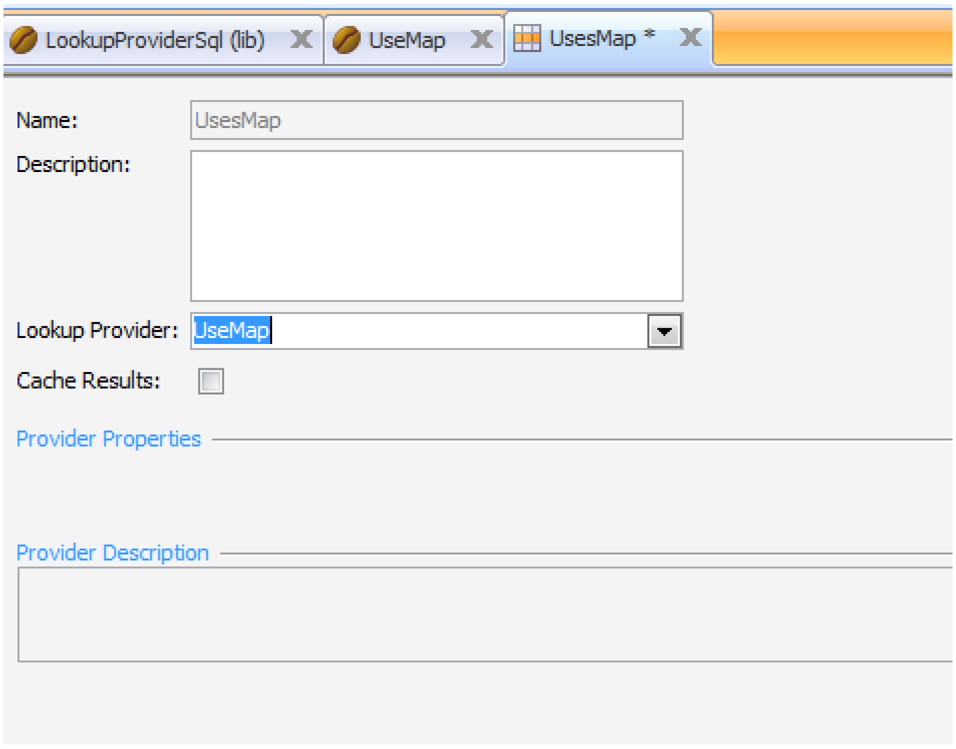
Create a new Named Lookup Table

Group:	FromX
Name:	UsesMap

Lookup Definition

In Transformer [✓ External to Transformer]

In the resulting form choses the new Lookup Provider.



Name: UsesMap

Description:

Lookup Provider: UseMap

Cache Results: ☐

Provider Properties

Provider Description

15 Glossary

BusinessObjectType (BOT) – the fundamental building blocks of Transformer. At the lowest level everything is built up of combinations of Strings, Integers, and Decimals etc. A simple Component or Component Type will be based (although not necessarily directly) on a BOT.

Built Project – a jarred or zipped Transformer project that has been created by performing a Build Project action from the GUI design tool.

Component – A Component is a DadNode that is a building block of a message. It is comparable to an XML Element in XML Schema terms.

ComponentType – A ComponantType is a DadNode that is a building block of a message. It is comparable to an XML Simple or Complex Type in XML Schema terms.

ConditionDefinition – A ConditionDefinition could be thought of as a reusable If condition. If some complex logic is required to work out if a statement is true or false this logic could be encapsulated in a ConditionDefinition for later reuse.

ConfigFile – the Java class representing a Configuration file. A valid ConfigFile will contain a hierarchy of ConfigObjects.

ConfigObject – The Configuration Object that is held within a ConfigFile. ConfigObjects represent a hierarchy within a ConfigFile, and ConfigFiles represent a hierarchy within a Project. All the items that can be produced and edited by the Transformer GUI will have a corresponding ConfigObject.

DadNode - A part of a ***D***iction***A***ry ***Definition*** (DAD), which represents a constituent of the structure of a particular message type. For example, a MT540 Message in SWIFT 15022, or a Complex Type in an XML Schema.

EnrichmentDefinition – An enrichment definition is like a mapping but the returned object is the same definition as the input. An enrichement is used to embellish the message with additional information usually from an external system like a database table for example.

EnrichmentRule - An Enrichment Rule acts on a Component, Component Type or Message and is used to embellish the DadNode with additional information usually from an external system.

ExposedService – this is a collection of Exposed Service Operations. An Exposed Service provides a mechanism to expose its operations to a calling system or architecture. For example, one exposed service may be set up so that its operations are callable as Plain Old Java Objects (POJOs) via the use of the CoarseAPI ServiceBuilder whilst another may exposed its operations as RESTful Web Services via the RESTful ServiceBuilder.

ExposedServiceOperation – encapsulate a mapping or validation and allow it to be called from an external system as a single method call.

Library – A Built Transformer Project that contains pre-defined NamedComponent, NamedComponentType and Message Definitions.

Mapper – A TBean that is reponsible for performing a Mapping action task. Typical examples are CopyText, DateTimeToText etc.

MappingDefinition – A Mapping Definition or Mapping is a set of transformation rules for creating a Component, Component Type or Message from a different Component, Type or Message.

MappingDefinitionGroup – Generally Mappings are stored in Groups. A Mapping Definition Group describes a set of mappings going from one standard (Message Definition Group) to another. For example you might have a Mapping Definition Group that contains all of the mappings from FpML to FIX.

MessageContent Factory – A software component (TBean) which provides message content objects that can be parsed or formatted, including dealing with byte/character encoding and decoding. Configurable for each Message Definition Group.

MessageDefinition – a DadNode representing the entire structure of a message. In XML Schema terms it is comparable with a Root Level Element.

MessageDefinitionGroup – A Message Definition Group (MDG) is collection of Components, Component Types and Messages representing an entire message standard. For example you might have an MDG representing all of the FIX messages. A MessageDefintionGroup is comparable to an entire Xml Schema.

MessageTypeIdentifier (MTI) – these are a mechanism to identify different messages in the same standard. There are usually different MTIs for different message statndards. For example, a CSV style message could be indentified by the contents of the first field within it, an XML message could be identified by its Root Element and a Swift MT message could be identified by the appropriate field in Block 1. Each of these examples would correspond to a different MTI in Transformer terms.

MomNode – A part of ***M***essage ***O***bject ***M***odel (MOM), which represents the run-time instance of a particular message. A MomNode contains the DadNode to which it refers, the data associated with the Node and any validation or error messages.

MomNodeBasket – A Mom Node Basket is a collection of Mom Nodes that represents the inputs or outputs to or from a Mapping (or Enrichment, Condition etc). The Input basket is constructed from the Mapping that is about to be called and subsequently populated and the output basket is produced as a result of calling the Mapping.

NamedActionSet – the base class for Mapping Definitions, Enrichment Definitions, Condition Definitions and Rules.

NamedComponent – a DadNode that represents a single aspect of a Message. Components can be simple or complex and are built up by combining BOTs either directly or indirectly. Components may be based on Component Types. Referred to in the GUI as a “Component”

NamedComponentType – a DadNode that can be thought of as a building block for creating NamedComponents, other NamedComponentTypes or MessageDefinitions. NamedComponentTypes can be simple or complex and are built up by combining BOTs either directly or indirectly. Referred to in the GUI as a “ComponentType”

Project - a self contained set of user definable configuration files named according to a logical directory type structure.

Project Key – a unique string used to ideantify a Project. The Project’s Project Key is stored inside the jar when the project is built and is used when the project is loaded at runtime.

Rule – A rule acts on a Component, Component Type or Message and can be used to either validate it (a ValidationRule) or enrich it (an EnrichmentRule).

RuleSet – is a Collection of Rules. The user is allowed to specify Validation RuleSets or Enrichment RuleSets.

RuntimeContext – a means of storing additional information that can be used when a Mapping (or Rule, Condition etc) is running. A typical example of this is storing of a Database Connection or meta-data item.

SqlQueryDefinition – A Transformer configuration item used to query a Database in a similar way to an SQL select statement.

TBean – A Transformer configuration item that performs a specific role e.g. Mapper, Parser, ValueGenerator etc.

Tra File – the configuration files as they appear on disk.

ValidationRule – A Validation Rule acts on a Component, Component Type or Message and is used to check it for errors. Generally speaking Validation Rules go beyond simple syntax error checking and are used for inter-field error checking. E.g. a typical Validation Rule might have logic along the lines of: If Field A is populated then Field B must be absent.

ValueGenerator – a TBean used to generate information in a mapping e.g. Current DateTime.