JARE

JAva **R**ule **E**ngine

version 0.62

by: uwe geercken
web: www.datamelt.com
email: uwe.geercken@web.de

last update: 2014-10-12

Table of Contents

Introduction

General

A business rule engine – written in Java – has been implemented called: JARE – Java Rule Engine. The basic idea behind a business rule engine is to have a set of data – in a file or database - and a set of rules that gets executed against the data, using certain test criteria (checks). At the end there is a list of results and messages that are produced when the rules failed or passed the tests.

„A set of data" mentioned above means that the user has one or multiple sets of data of the same structure which have to be tested. E.g. a comma separated file which contains a list of all airports in the world. Or the user could have a database with all addresses of his customers, that he wants to check against certain criteria. Or maybe you have a million records of telecommunication bills that need to be checked for consistency to assure the quality of the data.

The program comes with a couple of samples that you can run, to understand how it works.

Web application

Also, a web application is available to define the rules and their logical connection to each other and also actions in an easy fashion. If you use the web application to construct rules, the technical details of how the XML file is built (further below) are not so important to know.

Connected rules

Rules often do not come standalone. Many times there are multiple rules that belong together and build a logical unit. So multiple conditions are chained together. E.g. „if the first name of a persons details is empty AND the last name of the persons details is empty then this is invalid". Another example would be: „if the person is from Munich OR if the person is from Frankfurt OR if the person is from Paris AND his age is greater than 62, then this is an error" (because a certain service is not offered in this cities from a certain age onwards).

As you see, rules can be logically connected using [and] or [or] statements. This quickly gets quite complex and is difficult to follow and understand. At the same time, a set of rules might be used in multiple different contexts. The example above with the last name and the first name of a persons details being empty - that causes an error - is usable in different contexts, such as when filling out a registration form in a web application or when adding a new customer to a database. So rules and sets of rules need to be combined in a flexible manner and also need to be re-usable. Re-using rules saves you from defining them twice and – more important – saves you from maintaining them twice.

Uses of the Rule Engine

When all the business logic is either hard coded in programs, spread over different places or even both, then updating the logic, in case it changes, is difficult. Maybe you have to re-compile your program. Or the logic is in different locations and you first need to collect all before you understand how they work together; are you sure you have collected them all? Testing of your changed logic will be difficult, if you do not have the logic in one place and in a standard syntax.

Duplication of your rules or business logic in multiple places is a problem. If it changes, you have to update it in multiple places. If you can not – because somebody else is responsible for the maintenance of part of the logic – then you have to wait or you update your part and wait until the rest is done some time later. Both cases are a quality issue. Consistency is not guaranteed because you have different versions of the same logic or you are using outdated logic.

The amount of data that we handle every day, grows at an enormous speed. So does the number of applications. It gets very difficult to check the data for consistency especially if it comes e.g. from outside your company and is used in multiple places. Again, if you have no central logic of checking the data, this is causing repeated effort in multiple places and makes it difficult to guarantee quality.

A central place to store re-usable rules, which are not incorporated in programs but exist on their own, helps to solve the above mentioned problems. It makes the rules portable and it is easy to create a backup of them. Additionally it allows you to analyze the data for the occurrence of certain patterns or content by using regular expressions and pattern matching. The rule engine decouples the logic from the implementation, makes the program more flexible and helps to improve quality.

If business logic is in your program, in the ETL logic or other places usually not accessed by the business users, then this logic is in-transparent to the users. They have no way to look at it and thus, it is the responsible IT guy who will be called for additions or changes to the logic.

Another area for using the rule engine is data discovery. You can write a rule and log the result to a file in case the rule fails or in case the rule is true. This way you can detect values above a given limit or you can find unusual values. If you run your rules against a large set of data, you can find out how many of the data records fail or pass your rules.

Parts of the Business Rule Engine

To run the business rules engine, multiple parts are required: The rule engine itself (represented by a Java jar file), a file with data or data from a database and rules that are defined in one or multiple XML files. XML is the native format for the rules, but as indicated further above, there is a web application available for writing the rules and orchestrating them, so that there is no need to write XML files by hand. Finally you need to use some tests or checks which test for equality, smaller or greater, null, empty or other criteria.

The rule engine comes with several predefined checks that can be used - which are explained in detail further below. You can also define your own checks to extend the functionality of the business rule engine.

Rule Engine

The engine is a Java program. Thus Java is required to run it - either the runtime environment (JRE) or the development kit (JDK or SDK). It can be run from the console or command line or from within a development environment such as NetBeans or Eclipse. You may run it standalone or integrate it e.g. in your own projects or web applications. The engine can also be used by you favorite scripting language such as e.g. Beanshell, Groovy or Jython - just to name some.

The Jare jar file contains already several predefined checks or tests – as shown below - that the rule engine can use to check the data against the rules. E.g. if a number is smaller than an expected value, if a value matches a certain pattern or if a field is

empty or null, equal to another field or value, numeric, in a list of specified values or maybe a prime number.

Checks

A check defines a certain type of test. Together with the data the check evaluates, if the data corresponds to the expected result. E.g. the CheckIsEqual routine, checks for equality of values. If the data that you want to check is not according to the defined expected result then this would result in a rule that fails.

Checks are used in XML rule files to define how the data should be tested. Below find a list of all the predefined checks currently available. And you are free to implement your own checks if you want to.

For more details, especially on methods and their arguments of the below checks, consult the API.

Name of Class	Description	Applies to
CheckIsEqual	Checks for equality of values	Long, Integer, Boolean, String
CheckIsNotEqual	Checks if two values are not equal	Long, Integer, Boolean, String
CheckContains	Checks if one String contains another String	String
CheckNotContains	Checks if one String does NOT contain another String	String
CheckIsUppercase	Checks if a given string contains upper case characters only	String
CheckIsLowercase	Checks if a String contains lower case characters only	String
CheckEndsWith	Checks if a String ends with a certain String	String
CheckNotEndsWith	Checks if a String NOT ends with a certain String	String
CheckStartsWith	Checks if a String starts with a certain String	String
CheckNotStartsWith	Checks if a String NOT starts with a certain String	String
CheckIsGreater	Checks if a numeric value is greater than the other one. In case of Strings checks if the length of the String is greater than the other.	Integer, Long, Double, Float, String
CheckIsGreaterOrEqual	Checks if a numeric value is greater or equal than the other one. In case of Strings checks if the length of the String is greater or equal than the other.	Integer, Long, Double, Float, String
CheckIsInList	Checks if a string is contained in a list of values. The list is represented by a string where the individual values are separated by a comma (,)	String
CheckIsNotInList	Checks if a string is not contained in list of values. The list is represented by a string where the individual values are separated by a comma (,)	String
CheckIsBetween	Checks if a numeric value is between two given values	Integer, Long
CheckIsNotBetween	Checks if a numeric value is not between two given values.	Integer, Long
CheckIsSmaller	Checks if a numeric value is smaller than the other one. In case of Strings checks if the length of the String is smaller than the other.	Integer, Long, Double, Float, String

Name of Class	Description	Applies to
CheckIsSmallerOrEqual	Checks if a numeric value is smaller or equal than the other one. In case of Strings checks if the length of the String is smaller or equal than the other.	Integer, Long, Double, Float, String
CheckIsNumeric	Checks if a value is numeric, meaning all characters are numbers	String
CheckIsEmpty	Checks if a String value is empty, meaning of zero length	String
CheckIsNotEmpty	Checks if a String value is NOT empty, meaning its length is greater than zero	String
CheckIsNull	Checks if a value is Null	String
CheckIsNotNull	Checks if a value is NOT Null	String
CheckLength	Checks if a String has a defined length. In case of an integer, checks the number of digits the integer consists of	String, Integer
CheckMatches	Checks if a String matches another, using a regular expression pattern	String
CheckNotMatches	Checks if a String does NOT match another, using a regular expression pattern.	String
CheckSoundsLike	Checks if a String sounds like another, using the soundex algorithm	String
CheckNotSoundsLike	Checks if a String does not sound like another, using the soundex algorithm	String
CheckIsNegativeNumber	Checks if a number is smaller than zero (0)	Integer, Long, Double, Float

Some checks allow additional parameters to be passed to it. E.g. the check for equality allows an additional parameter to ignore (or not to ignore) the case of the Strings. Details can be found further below (under "Parameter tag") or can be reviewed in the API documentation.

Rule Group Actions

After all rules of a group are run against the data, the user might want to apply some actions to the objects involved. Usually either a database record or a row of data from a CSV or text file is used with the rule engine. Both the database record and the row object provide methods to update its fields. This way an action can be used to update e.g. one field based on a value of another field. Another example would be to check if a certain condition is found and then changing the value of that same field. Or one could – based on the results of the rules of the group – set a field to a fixed value.

All this is possible by defining actions. The program comes with some predefined actions to e.g. concatenate fields, to prepend or append text to fields or to apply certain calculations.

But the actions architecture is designed in a way, that a developer can simply add new actions and call them from the rule engine. Any Java class and method may be used. This allows unlimited flexibility in the way of what an action does based on the outcome of the rules of a group and gives the user the ability to tailor the usage of the program to his needs.

When looping over a number of records or over the lines in a text file, the actions are applied to each record or line. The user can steer if the actions are applied when the group fails or when the group passes.

If multiple actions are defined for a rule group, one action after the other is executed. The actions can be run against different objects. So e.g. action 1 may update field A and action 2 may update field B.

So actions can be seen as a way of applying multiple changes to the data after it has been determined that a group of rules has passed or failed. Another possibility – instead of modifying the data – would be to e.g. email or output the results to a file.

Rule Engine properties file

The user may specify a properties file, named "engine.properties", that comes with the rule engine and is used to define several parameters of how the engine runs and behaves. Optional parameters do not need to be specified. In this case the program uses a default value instead.

Properties are defined as key/pair values. E.g.

templates_folder=/home/user1/test

Below the possible values are listed:

Property key	Value
replacements_file	Optional. The full path and name of the replacements file to use.
templates_folder	Optional. The full path to the folder where the templates for messages and rule logic are stored.
messages_template	Optional. The name of the template to be used for messages output. If none is specified, then a default output will be generated. Template must be located in the *templates_folder*.
rule_logic_template	Optional. The name of the template to be used for rule logic output. If none is specified, then a default output will be generated. Template must be located in the *templates_folder*.
output_file	Optional. The full path and name of the output file to be used. Messages will be stored in this file. If none is specified, all output will go to the console.
output_file_timestamp	Optional. The format of the timestamp that will be added to the name of the output file. The format follows the specification of the Java SimpleDateFormat class.

Property key	Value
output_type	Optional. Defines which messages will be output, only failed ones (0), only passed ones (1) or both (2). Must be 0 or 1 or 2.
object_label	Optional. Each object in the test gets a running number to identify the object when e.g. outputting the reults to a log file. This label will be used to identify every object processed.
object_label_format	Optional. Each object in the test gets a running number to identify the object. This value defines the format of the running number. Follows the java syntax of the DecimalFormat class. Default is [00000].
actions_output_file	Optional. The full path and name of the actions output file to be used. Results of the actions can be stored in this file. If none is specified, all output will go to the console.
actions_output_file_timestamp	Optional. The format of the timestamp that will be added to the name of the actions output file. The format follows the specification of the Java SimpleDateFormat class.

The data

Data can come from different locations. Most of the times the data comes from a database or alternatively from a file. Database can be any database that is accessible through JDBC. So any of the known database sources such as e.g. Microsoft SQL Server, Oracle or MySQL may be used - just to name some.

Files come in different shape and form, but most of the times – at least when containing data – comma separated (CSV) files are used. In these files each row represents a record and the fields of each row are separated by a delimiter. Usually a TAB character, a semicolon (;) or a comma (,). These files may also come in other formats that are more complicated but the formatting follows a similar pattern.

Here is an excerpt from the *airport.csv* example file:

```
KABE;LEHIGH VALLEY INTL;00393;N40390750;W075262690
KABI;ABILENE RGNL;01791;N32244075;W099405483
KABR;ABERDEEN RGNL;01302;N45265660;W098251860
KABY;SOUTHWEST GEORGIA RGNL;00197;N31320785;W084114010
KACK;NANTUCKET MEM;00048;N41151099;W070033665
KACT;WACO RGNL;00516;N31364064;W097134987
..
..
```

Explanation: Semicolons (;) are used to separate the fields. The first column contains the 4-character ICAO (International Civil Aviation Organization) code of the airport, which is a unique identifier. In the second field comes the name and in the third field the elevation. Lastly two fields follow, which contain the latitude and longitude position values of the airport.

Another alternative is, to use fixed length ASCII file. Each row in such a file has the same length. The fields of the row are defined by a starting position and a fixed length. This way a delimiter is not required because – by using the defined position of a field – the value of the field can be retrieved.

In the case of fixed length ASCII files, a file needs to be defined that describes where each field starts and its length. This way the program can extract the values correctly from the data file. Below is an example of the format of the file:

```xml
<xml>
      <row>
              <field name="icao" description="ICAO Code 4-Letter" start="0" length="4"/>
              <field name="name" start="4" length="40"/>
              <field name="elevation" start="44" length="5"/>
              <field name="latitude" start="49" length="9"/>
              <field name="longitude" start="58" length="10"/>
      </row>
</xml>
```

Here is an excerpt from the *airport_fixed.txt* example file:

```
KABELEHIGH VALLEY INTL               00393N40390750W075262690
KABIABILENE RGNL                     01791N32244075W099405483
KABRABERDEEN RGNL                    01302N45265660W098251860
KABYSOUTHWEST GEORGIA RGNL           00197N31320785W084114010
KACKNANTUCKET MEM                    00048N41151099W070033665
KACTWACO RGNL                        00516N31364064W097134987
..
..
```

Explanation: Each field in each row has a defined start position and length. The first field contains the 4-character ICAO (International Civil Aviation Organization) code of the airport, which is a unique identifier. In the second field comes the name and in the third field the elevation. Lastly two fields follow, which contain the latitude and longitude values of the airport. They are all well aligned. Fields that contain less data then the maximum length of the field – like in this case the description field - are filled with trailing spaces.

Both types – databases and files - can be used to feed the data into the rule engine and test it against a set of rules. The data needs to be wrapped in java objects, so that the rule engine can use its methods to test the data. To do so, the rule engine is using the Java reflection mechanism. This can be any java object and the class of the objects and the method to be used is defined in the XML rule file.

Rules and grouping

Lastly some rules are needed (or at least one) that are used to check the data. Rules are written in XML format and define what should be tested or more specifically, what is expected to be returned when a rule is run, what messages are to be output when a rule passes or fails and other details.

Rules are defined in so called subgroups and can be combined using either [and] or [or] conditions - but not both in one subgroup - to logically connect them to each other. So you can construct a subgroup like this: rule1 and rule2 and rule3 and rule4. All rules must be true for the subgroup to pass. Or define a group like this: rule1 or rule2 or rule3 or rule4. This means, that only one rule needs to pass successful for the whole subgroup to pass. These conditions are define in the rule using the "ruleoperator" tag. If you have more complex relations between the rules like e.g. „rule1 or rule2 and rule3" then you have to define them in separate subgroups.

Subgroups are defined in a group. One group – represented in one XML file - can have multiple subgroups, which again can be connected to each other using [and] or [or] conditions. These are defined in the XML file using the "intergroupoperator" tag. This operator is always relative to the previous subgroup. The first subgroup subsequently does not need this operator. Using multiple subgroups any logical condition can be defined to build a group of rules that belong together to test or analyze a certain condition.

The logic of connecting the subgroups with each other is as follows: the first group is compared to the second group. The result is then compared to the third group. The result again is compared to the fourth group, and so on. So the subgroups are chained together, one with the next one.

In subgroup1 for example you have two rules connected with an [or] condition: rule1 or rule2. Then in subgroup2 for example you have two rules connected with an [and] condition. The both groups are connected with an [or] condition. So the overall logic – if one wanted to write it out as a sentence - would be: if rule1 or rule2 of subgroup1 or rule1 and rule2 of subgroup2 passes, then the test would be passed.

Below find some lines from the examples *airports_0001.xml* file:

```xml
<xml>
<group id="group_1" description="rule groups to check airport record consistency">
       <subgroup id="subgroup_1" description="check icao code" ruleoperator="and">
              <rule id="rule_icao_1" description="check for correct icao code spelling">
              <object classname="Row" method="getField" type="string"
              parameter="0" parametertype="integer"/>
              <expected value="[A-Z]{4}" type="string"/>
              <execute value="com.datamelt.rules.implementation.CheckMatches">
              </execute>
              <message type="failed" text="ICAO code field of $1 does not match
              to expected value: $0" />
              <message type="passed" text="Correct match of ICAO code field to
              expected value: $0" />
              </rule>
              ...
       </subgroup>
       <subgroup id="subgroup_1" description="check lat-long values"
       intergroupoperator="and" ruleoperator="and">
              ...
              <rule id="rule_longitude_0" description="check for correct longitude
              value">
              <object classname="Row" method="getField" type="string"
              parameter="4" parametertype="integer"/>
              <expected value="[WE][0-9]{9}" type="string"/>
              <execute value="com.datamelt.rules.implementation.CheckMatches">
              </execute>
              <message type="failed" text="Longitude field $1 does not match to
              expected value: $0" />
              <message type="passed" text="Correct match of longitude field to
              expected value: $0" />
              </rule>
       </subgroup>
</group>
</xml>
```

Rule fails – group passes

An advantage of this concept of grouping rules in groups and subgroups is, that you can have one or multiple rules that fail a certain test, but the group as a hole passes it. This is also valid for subgroups. A subgroup may fail, because the conditions are not met, but the group – constructed of multiple subgroups – may pass.

Lets take the example of a car renting company that wants to check if somebody is of a certain age – say 28 - before it rents a nice, shiny Ferrari to somebody. But at the same time another rule of the company says, that a person who is younger than 28 may rent the car, if he or she is willing to pay an additional fee.

Now if e.g. A 25 year old girl comes and wants to rent the Ferrari, then the first rule will fail, because she is too young. But the second will pass because she indicates that she does not care how much it cost, as long as she gets that fabulous car.

So rule 1 fails, but rule 2 passes. The result is that the girl may rent the car, although one of the companies rules failed. Through the combination of rules and subgroups we can construct a group which will do exactly what we saw in this example.

Rules Zipfile

Rules can be put together in a zip file. Simply use the tool of your choice to add all relevant files to an zip archive. The rule engine will process the rules from the zip archive and execute them just the same way as if you would specify a folder containing the rule files. This allows you to put all rules belonging together in one single file, respectively to collect your rules for different projects and purposes in different zip files.

The web application automates this process: you write the rules and actions and you combine the rules in a logical way and finally you export the file as a zip file to be used with the rule engine.

Having all rules in one single file has the advantage that you can use it to create versions of your rule logic. You could have one zip file for a certain period of time and another one – with different rule logic – for another time period.

When the Rule Engine runs

Once all the parts are together, the rule engine can be run. It scans first a given directory for XML rule files (or a zip file containing the rules) and parses them. Next it takes a data record and runs all rules against it. The engine will check the conditions of [and] or [or] that exist between the rules and between the subgroups and determine if the data passes the logic. When a rule passes or when a rule fails, a message may be output to indicate what went wrong, including the parameters that were used to run the rule. Output may be to the console or into a file.

Note that a check will fail if the expected type does not correspond with the type of the data. E.g. if an integer is expected, but the data contains a string value. In this case the resulting message will indicate a conversion type error.

When the process is complete all data is tested against the whole set of rules. It can be determined what went wrong and where. If a rule passed the tests, if its subgroup passed the tests and if the group as such passed or failed the tests.

The format of the output of the results can be defined transparently and flexible using templates. Maybe you want all results to be output, including all rules that failed. But maybe in another situation, you only want to know if the group as a whole passed the test. All this can be defined in the template. The rule engine uses a template engine which allows you to define the output in any format that is text based, such as HTML, CSV, XML, JSON and others.

Business Rules definition

The rules - as already indicated - are written in a XML format. XML is an ASCII based text format. So you can write a rules definition file with a simple text editor.

The file follows the standard XML syntax. If you have no experience of how to write XML files, I would recommend to get a good book or check the web. There is a lot of information available to get you started. Or you use the web application available to write your rules and actions in an easy fashion and you don't need to know anything about XML at all.

The XML file is the point where the groups, subgroups and rules are defined. Additionally there is a so called replacements file. It is an optional properties file with key/value pairs. When you have to repeat values frequently in the XML file, you can replace it with a variable – that you define in the replacements file. This way, you have to write less and if the original value changes, you have to change it in one place only.

When the rule engine parses the XML rule file(s). All occurrences of variables in the file are replaced with their real values from the replacements file.

For example in the replacements file you put in a line:

$AIRLINE_NAME_LX=SWISS

And to use the variable in the XML file, just write down the variable value with a dollar sign ($) in front of it. So in this example put $AIRLINE_NAME_LX in the XML file.

Below find a list and of the tags that are used with a description of its meaning and their attributes.

XML tag

On the highest level there are the *XML* start and end tags. All groups, subgroups and rules will be located between these two tags.

XML tags by the way always consist of two tags: one opening tag and one closing tag, as can be seen in the example below.

```
<xml>
  ...
</xml>
```

Group tag

Next comes the *group* tag. Only one *group* tag is allowed per file. It groups all subgroups (see below) together.

```
<xml>
  <group id="group_1" description="check record consistency" outputafteractions="true">
    ...
  </group>
  <action>
  </action>
</xml>
```

The group tag can have following attributes:

Name	Description
id	Required. Uniquely identifies the group.
description	Optional. Describes the purpose of the group
outputafteractions	Optional. After all actions are run the objects toString() method is called and the result si output to the console or file

Subgroup tag

Next comes the *subgroup*. It is a sub- or child-tag of the *group* tag. One group can have one to many subgroups underneath it.

```
<xml>
  <group id="group_1" description="check record consistency" outputafteractions="true">
    <subgroup id="subgroup_1" description="check icao code" ruleoperator="and">
     ...
    </subgroup>
  </group>
  <action>
  </action>
</xml>
```

Subgroups have following attributes:

Name	Description
id	Required. Uniquely identifies the subgroup.
description	Optional. Describes the purpose of the subgroup
ruleoperator	Optional. Specifies how the rules that belong to one subgroup are logically connected to each other. Possible values are [and] or [or]. Default is [and]
intergroupoperator	Optional. Specifies how the subgroup is logically connected to the previous subgroup. Possible values are [and] or [or]. Default is [and] Note: for the first subgroup in a group this attribute is ignored, because it has no previous subgroup

Rule tag

A subgroup itself can contain one to many rules.

```
<xml>
  <group id="group_1" description="rule groups to check airport record consistency">
    <subgroup id="subgroup_1" description="check icao code" ruleoperator="and">
      <rule id="rule_icao_1" description="check for correct icao code spelling">
        ...
      </rule>
    </subgroup>
  </group>
  <action>
  </action>
</xml>
```

Rules can have following attributes:

Name	Description
id	Required. Uniquely identifies the rule
description	Optional. Describes the purpose of the rule

Object tag

Inside a rule tag there must be an object tag. The object is a reference to a Java class that contains the actual data that is to be checked. The method of the object with the given parameter (argument) is executed - in the example below: row.getField(0) - and the result will be compared to the expected value (explained further below).

Actually one can have two *object* tags per rule. In this case, the *expected* tag has to be left out. In this case the test is not made between the object and the expected value, but between the two objects. This allows to check two fields of an object against each other and to evaluate the result of the comparison.

```xml
<xml>
  <group id="group_1" description="rule groups to check airport record consistency">
    <subgroup id="subgroup_1" description="check icao code" ruleoperator="and">
      <rule id="rule_icao_1" description="check for correct icao code spelling">
        <object classname="Row" method="getField" type="string" parameter="0"
        parametertype="integer"/>
        ...
      </rule>
    </subgroup>
  </group>
  <action>
  </action>
</xml>
```

The object tag can have following attributes:

Name	Description
classname	Required. Name of the class that will be instantiated. This is the object that contains the data.
method	Required. Name of the method to execute for the above mentioned object.
type	Required.The return type of the method that will be executed. E.g. string, int, boolean, etc. Corresponds to the regular Java objecs. Case insensitive; you can e.g. write String or string.
parameter	Required. The parameter to pass to the method, in case one is required. Variables from the replacements properties file may be used instead of the real value.
parametertype	Required. The type of the parameter to be passed to the method.

Expected tag

The expected value (of the *expected* tag) is that value, that – when the method on the object, as described above, is executed, is expected to be returned from the method. It is the value that you test for. When this *expected* tag is left out, then the test that is run is a test that does not compare two values (the object value to the expected value), but does a simple check only. E.g. a check for Null.

```
<xml>
  <group id="group_1" description="rule groups to check airport record consistency">
    <subgroup id="subgroup_1" description="check icao code" ruleoperator="and">
      <rule id="rule_icao_1" description="check for correct icao code spelling">
        <object classname="Row" method="getField" type="string" parameter="0"
        parametertype="integer"/>
        <expected value="[A-Z]{4}" type="string"/>
        ...
      </rule>
    </subgroup>
  </group>
  <action>
  </action>
</xml>
```

The expected tag can have following attributes:

Name	Description
value	Required. The value that is expected to be returned from executing the method of the object. Variables from the replacements properties file may be used instead of the real value.
type	Required. Type of the value. E.g. string, integer, boolean, etc.

Execute tag

The execute tag specifies which check to run on the data. Checks (or tests) are defined in java classes. The rule engine already comes with a large set of checks that have been predefined for the user. One can implement his own checks by implementing the GenericCheck class. Please consult the API documentation for more details.

```
<xml>
  <group id="group_1" description="rule groups to check airport record consistency">
    <subgroup id="subgroup_1" description="check icao code" ruleoperator="and">
      <rule id="rule_icao_1" description="check for correct icao code spelling">
        <object classname="Row" method="getField" type="string" parameter="0"
        parametertype="integer"/>
        <expected value="[A-Z]{4}" type="string"/>
        <execute value="com.datamelt.rules.implementation.CheckMatches">
              ...
        </execute>
        ...
      </rule>
    </subgroup>
  </group>
  <action>
  </action>
</xml>
```

The *execute* tag can have following attributes:

Name	Description
value	Required. The name of the class that implements the class GenericCheck and has to be executed on the data. The evaluate method of the rule will be executed with both the value of the object created and the expected value. The result must be a true or false.

Parameter tag

The *execute* tag may have an additional sub- or child-tag *parameter*, in case the check class expects one or multiple additional parameters to be passed to it. For example the check for equality (CheckIsEqual) accepts an additional boolean parameter, if during the check the case of the values is to be ignored or not.

```xml
<xml>
  <group id="group_1" description="rule groups to check airport record consistency">
    <subgroup id="subgroup_1" description="check icao code" ruleoperator="and">
      <rule id="rule_icao_1" description="check for correct icao code spelling">
        <object classname="Row" method="getField" type="string" parameter="0"
        parametertype="integer"/>
        <expected value="[A-Z]{4}" type="string"/>
        <execute value="com.datamelt.rules.implementation.CheckMatches">
          <parameter type="boolean" value="true"/>
        </execute>
        ...
      </rule>
    </subgroup>
  </group>
  <action>
  </action>
</xml>
```

The *parameter* tag under the *execute* tag can have following attributes:

Name	Description
type	The type of the parameter. E.g. string, integer, boolean, etc.
value	The value of the parameter. Variables from the replacements properties file may be used here instead of the real value.

Message tag

The *message* tag identifies the message that is to be generated, when the rule either passes or fails. So each rule has two messages.

```xml
<xml>
  <group id="group_1" description="rule groups to check airport record consistency">
    <subgroup id="subgroup_1" description="check icao code" ruleoperator="and">
      <rule id="rule_icao_1" description="check for correct icao code spelling">
        <object classname="Row" method="getField" type="string" parameter="0"
        parametertype="integer"/>
        <expected value="[A-Z]{4}" type="string"/>
        <execute value="com.datamelt.rules.implementation.CheckMatches">
          <parameter type="boolean" value="true"/>
        </execute>
        <message type="failed" text="ICAO code field of $1 does not match to
        expected value: $0" />
        <message type="passed" text="Correct match of ICAO code field to expected
        value: $0" />
      </rule>
    </subgroup>
  </group>
  <action>
  </action>
</xml>
```

Actions

The Xml syntax allows for zero, one or many actions to be defined. Actions are executed sequentially and for each record, row or other object. As the rule engine is open to accept any Java object, the user may use any Java class as an object. But in many cases, as said before, a typical example of an object would be a row (from a file) or a resultset from a database.

By implementing action classes one can create methods that do something with the values of an object, in case the rule group fails (or passes) and thus allows the user to create - in a very simple way - an extendible suit of actions that full fill specific tasks.

There are several possibilities of how to apply actions to the objects involved:

a) Provide an action class name, method and value plus an object defining a setter method. This will instantiate the action class and set the object using its setter method to the value given in the action class.

b) Provide an action class name, method, an object defining a getter method and an object defining a setter method. This will instantiate the action class, get the value of the object defining the getter object and then apply the value to the object defined as the setter object.

c) Provide an action class name and method. This will instantiate the action class and invoke the named method. But it will not update any objects.

d) Provide an object. The object method will be invoked without using an action class. In this case you have to specify an action class tag in the xml file without class name and method.

So these are four options that can be used to invoke actions. If your input (object) e.g. is a record from a database, than it might come as part of a Java resultset. The resultset comes with methods to retrieve the values of fields of the current record. An action allows e.g. to retrieve the value of a field of the record and apply it to another field. So depending on what your object is, it provides methods to get and set values or to execute other methods.

If you want to set a certain field to a constant value then option a) above would be a valid approach. In the xml rule file an action with a value is defined – along with a class name and a method name - that the relevant field of the object is set to, in case the rule group fails or passes.

Option b) allows the user to get the value from one field and use it to set the value of another field. Together with an action class and method, it is possible e.g. to get the value of one field modify it and then apply it to the other field. Samples for modifications are to prepend or append values, set the date or time of a field, to do certain calculations, to add leading zeros or spaces, to trim the value (remove leading or trailing spaces) etc.

In case of option c) above, there are no objects involved, which means, there is no resulting value, that a certain field is set to. Anyway, this type of action can be used to e.g. create some output to a file or the console or other processing tasks.

Option d) allows to directly call the method of a given class. E.g. when using an updatable resultset, a first action could be to update a field of the resultset and another action to call the update() method of the resultset to apply the changes. In this case an action class is not required.

The Java Rule Engine comes with a class "SimpleAction". This class defines several methods that can be used to update data, to set a field to a date, to prepend or append a value to a field, to add a value to a field, to add leading zeros, etc.

It is quite easy to write your own classes, which then can be used with the rule engine to execute actions.

Action tag

The *action* tag defines the action(s) to be taken after a group fails or passes.

```xml
<xml>
  <group id="group_1" description="rule groups to check airport record consistency">
    <subgroup id="subgroup_1" description="check icao code" ruleoperator="and">
      <rule id="rule_icao_1" description="check for correct icao code spelling">
        <object classname="Row" method="getField" type="string" parameter="0"
        parametertype="integer"/>
        <expected value="[A-Z]{4}" type="string"/>
        <execute value="com.datamelt.rules.implementation.CheckMatches">
          <parameter type="boolean" value="true"/>
        </execute>
        <message type="failed" text="ICAO code field of $1 does not match to
        expected value: $0" />
        <message type="passed" text="Correct match of ICAO code field to expected
        value: $0" />
      </rule>
    </subgroup>
  </group>
  <action classname="com.datamelt.rules.core.action.SimpleAction" method="prependValue"
      executeif="failed">
  </action>
</xml>
```

The *action* tag can have following attributes:

Name	Description
classname	Optional. Name of the class to execute.
method	Optional. Name of the method to execute
executeif	Optional. Indicator to execute the action if the group failed or passed.
valuetype	Optional. Type of the value. E.g. string, integer, boolean, etc.
value	Optional. A constant value to be used to udpate an action object.

Action Parameter tag

The *action* tag can optionally have one to many parameter tags. These parameters define additional arguments that have to be passed to the method that is defined in the action tag.

```xml
<xml>
  <group id="group_1" description="rule groups to check airport record consistency">
    <subgroup id="subgroup_1" description="check icao code" ruleoperator="and">
      <rule id="rule_icao_1" description="check for correct icao code spelling">
        <object classname="Row" method="getField" type="string" parameter="0"
        parametertype="integer"/>
        <expected value="[A-Z]{4}" type="string"/>
        <execute value="com.datamelt.rules.implementation.CheckMatches">
          <parameter type="boolean" value="true"/>
        </execute>
        <message type="failed" text="ICAO code field of $1 does not match to
        expected value: $0" />
        <message type="passed" text="Correct match of ICAO code field to expected
        value: $0" />
      </rule>
    </subgroup>
  </group>
  <action classname="com.datamelt.rules.core.action.SimpleAction" method="prependValue"
      executeif="failed">
    <parameter type="string" value="passed-"/>
  </action>
</xml>
```

The *action* parameter tag can have following attributes:

Name	Description
type	The type of the parameter. E.g. string, integer, boolean, etc.
value	Value of the parameter.

Action Object tag

The *action* tag can have zero, one or two object tags. The objects are instantiated when the action is run and the relevant method is executed. Object may be used to retrieve (get) values or may be used to modify (set) values.

```xml
<xml>
  <group id="group_1" description="rule groups to check airport record consistency">
    <subgroup id="subgroup_1" description="check icao code" ruleoperator="and">
      <rule id="rule_icao_1" description="check for correct icao code spelling">
        <object classname="Row" method="getField" type="string" parameter="0"
        parametertype="integer"/>
        <expected value="[A-Z]{4}" type="string"/>
        <execute value="com.datamelt.rules.implementation.CheckMatches">
          <parameter type="boolean" value="true"/>
        </execute>
        <message type="failed" text="ICAO code field of $1 does not match to
        expected value: $0" />
        <message type="passed" text="Correct match of ICAO code field to expected
        value: $0" />
      </rule>
    </subgroup>
  </group>
  <action classname="com.datamelt.rules.core.action.SimpleAction" method="prependValue"
      executeif="failed">
    <parameter type="string" value="passed-"/>
      <object classname="com.mysql.jdbc.ResultSet" type="getter" returntype="string"
          method="getString">
      </object>
  </action>
</xml>
```

The *action object* tag can have following attributes:

Name	Description
classname	Name of the class to instantiate
method	Name of the method to execute.
type	Type of the method: *getter* or *setter*

Action Object Parameter tag

The *object* tag can optionally have one to many parameter tags. These parameters define additional arguments that have to be passed to the method that is defined in the object tag.

If the object is a setter object (type=setter), then this object is updated by an action. In this case, the parameter tag

```xml
<xml>
  <group id="group_1" description="rule groups to check airport record consistency">
    <subgroup id="subgroup_1" description="check icao code" ruleoperator="and">
      <rule id="rule_icao_1" description="check for correct icao code spelling">
        <object classname="Row" method="getField" type="string" parameter="0"
        parametertype="integer"/>
        <expected value="[A-Z]{4}" type="string"/>
        <execute value="com.datamelt.rules.implementation.CheckMatches">
          <parameter type="boolean" value="true"/>
        </execute>
        <message type="failed" text="ICAO code field of $1 does not match to
        expected value: $0" />
        <message type="passed" text="Correct match of ICAO code field to expected
        value: $0" />
      </rule>
    </subgroup>
  </group>
  <action classname="com.datamelt.rules.core.action.SimpleAction" method="prependValue"
      executeif="failed">
      <parameter type="string" value="passed-"/>
      <object classname="com.mysql.jdbc.ResultSet" type="getter" returntype="string"
          method="getString">
          <parameter type="string" value="icao" settervalue="true"/>
      </object>
  </action>
</xml>
```

The *object* parameter tag can have following attributes:

Name	Description
type	The type of the parameter. E.g. string, integer, boolean, etc.
value	Value of the parameter.
settervalue	Indicates if the parameter is used for updating the object. Required for a setter object that has multiple parameters.

Rule examples

Below find some examples for rules with explanations:

1)

```
<rule id="rule_hostname_2" description="check for correct hostname value">
      <object classname="Row" method="getField" type="string" parameter="30"
      parametertype="integer"/>
      <expected value="" type="string"/>
      <execute value="com.datamelt.rules.implementation.CheckIsEqual">
        <parameter type="boolean" value="true"/>
      </execute>
      <message type="failed" text="hostname $1 does not equals: $0" />
      <message type="passed" text="Correct hostname compared to expected value: $0" />
</rule>
```

Explanation: Checks if the value of the method *getField(30)* executed on the *row* object equals the expected string value „" (empty) using the *CheckIsEqual* routine and passing an additional boolean parameter = true to it (to ignore the case).

2)

```
<rule id="rule_elevation_0" description="check for elevation value greater than 0">
      <object classname="Row" method="getIntegerField" type="integer" parameter="2"
      parametertype="integer"/>
      <expected value="-50" type="integer"/>
      <execute value="com.datamelt.rules.implementation.CheckIsGreater"></execute>
      <message type="failed" text="Elevation field value $1 is not greater than expected
      value: $0" />
      <message type="passed" text="Correct value of elevation greater than expected value:
      $0"/>
</rule>
```

Explanation: Checks if the value of the method *getIntegerField(2)* executed on the *row* object is greater than the expected integer value of -50 using the *CheckIsGreater* routine.

3)

```
<rule id="rule_elevation_0" description="check for correct name syntax">
      <object classname="Row" method="getField" type="integer" parameter="6"
      parametertype="integer"/>
      <expected value="[A-Z]{4}" type="string"/>
      <execute value="com.datamelt.rules.implementation.CheckMatches">
      </execute>
      <message type="failed" text="Name field value $1 does not match expected value of $0"
      />
      <message type="passed" text="Name field correctly matches expected value: $0" />
</rule>
```

Explanation: Checks if the value of the method getField(6) executed on the row object matches the expected string of 4 alphabetic characters using the CheckMatches routine.

4)

```
<rule id="rule_hostname_2" description="check for equal values">
      <object classname="Row" method="getField" type="string" parameter="30"
      parametertype="integer"/>
      <object classname="Row" method="getField" type="string" parameter="31"
        parametertype="integer"/>
      <execute value="com.datamelt.rules.implementation.CheckIsEqual">
        <parameter type="boolean" value="true"/>
      </execute>
      <message type="failed" text="Unequal field values $1 compared to $0" />
      <message type="passed" text="Correct equals value  between fields: $0" />
</rule>
```

Explanation: Checks if the value of the method *getField(30)* and getField(31) executed on the *row* object are equal using the *CheckIsEqual* routine.

Action examples

Below find some examples for actions with explanations. Further above the different types of action scenarios are descibed. The options stated below reference these scenarios.

Note that the class "SimpleAction" comes with the Java Rule Engine. It contains methods that can be used to manipulate or output data.

option a):

```
<action classname="com.datamelt.rules.core.action.SimpleAction" method="setValue"
        executeif="failed" valuetype="string" value="peterpan">
        <object classname="com.mysql.jdbc.ResultSet" type="setter" method="updateString">
                <parameter type="string" value="testfield"/>
                <parameter type="string" settervalue="true"/>
        </object>
</action>
```

Explanation: The method "setValue" of the class "SimpleAction" is invoked by passing the value "peterpan" - of type string – to the objects setter method "updateString". This method requires the name of the field to update ("testfield" - a field in the database) and the value from the action class ("peterpan").

option b):

```
<action classname="com.datamelt.rules.core.action.SimpleAction" method="prependValue"
        executeif="passed">
        <parameter type="string" value="veryhigh-"/>
        <object classname="com.mysql.jdbc.ResultSet" type="getter" returntype="string"
                method="getString">
                <parameter type="string" value="icao"/>
        </object>
        <object classname="com.mysql.jdbc.ResultSet" type="setter" method="updateString">
                <parameter type="string" value="testfield"/>
                <parameter type="string" settervalue="true"/>
        </object>
</action>
```

Explanation: The method "prependValue" of the class "SimpleAction" is invoked. It prepends text to an existing string. The method requires the string of what to prepend to be passed ("veryhigh-") as a parameter. The first object defines the getter object – to retrieve the value from the field "icao". It invokes the "getString" method of the "ResultSet" class. The second object is the object to update (setter object). The name of the method to invoke is "updateString", This method requires the name of the field to update ("testfield" - a field in the database). So the value of the first object is retrieved and prepended by "veryhigh-" and then the second object is set to the resulting value.

option c):

```
<action classname="com.datamelt.rules.core.action.SimpleAction" method="setValue"
        executeif="failed">
        <parameter type="string" value="somevalue"/>
</action>
```

Explanation: The method "setValue" of the class "SimpleAction" is invoked. The method requires one parameter (or argument) to be passed to it. This is the value "somevalue" of type string.

option d):

```
<action executeif="failed">
        <object classname="com.mysql.jdbc.ResultSet" type="setter" method="updateRow"/>
</action>
```

Explanation: The method "updateRow()" of the class "ResultSet" is invoked. Notice that there is no class and method name defined for the action itself, so the method is invoked directly on the object.

Use of templates

In general, templates can be used to define a structure with place holders, which then can be mixed with some data. E.g. in a word processor you define one template and multiple address records for all your customers. Then you let the program merge the two to produce one document for each of the customers. If you want a different output, then you just change the template once – your data is separate from your presentation of it.

Currently there are two places in the business rule engine where you can use templates to define the output of certain functionality exactly the way you would like to have it. One place is the message of a rule. Each rule can have two messages: one in case it passes and another in case it fails. The way how it is output - the format - is defined in a template.

So the user can decide which elements of the group(s), subgroup(s) and rule(s) should appear in the output and the format that they should have. It is completely up to the user to use the results and display them.

If output e.g. goes into a file that will be read by a non-technical person, then the format needs to be different compared to when the output is used to automatically feed another system using a XML format. The user is open to decide how the output is presented, which makes using templates so beneficial. It could be in plain CSV, XML, HTML or any other ASCII based format.

The other place is what I call the rule definition. As described earlier, you can have many rules and many subgroups in one group in an XML file. The logic how these are all chained together is not easily retrievable, especially if you have a complex and large set of rules. So there is functionality in place to display an easy version of the complex logic. And a template is used to format, how the output of this text should look like. Here again, you are free, which parts to display and how to display them.

In both cases, the tool comes with some predefined templates to get you started quickly. Which template to use is defined in a properties file that goes with the rule engine.

I would recommend to have a look at the template engine that is used, if you want to modify the existing templates and create your own ones.

Apache Velocity

The method of using templates is not new. Actually there is a template engine used here, that is called Apache Velocity. It is very fast and easy to use in both applications and web sites. I would recommend it to everybody because it perfectly divides your data from your presentation.

I have developed quite a few websites – both small and larger in size - that work very well, are specifically flexible in terms of changes and extension and that are very fast. The Velocity template engine is very mighty. It is astonishing easy to use, flexible and versatile. Many people use it as a replacement for JSP or other languages. You can use it with shell scripts, scripting languages, for generating content or for migrations.

If somebody is interested, I can provide further details or show you how I implemented this outstanding tool. Otherwise visit the Apache Velocity website at http://velocity.apache.org to find detailed information and to download the tool.

More technical details

There is actually much more going on in respect to the process of running the tests. In the XML file you define which tests to run. As XML is plain text, there are no objects and methods that are pre-compiled that could be used. So they are constructed when the program runs, using the Java reflection mechanism.

Same is valid for the object containing the actual data. In the XML file you define which method has to be invoked on the object and which parameters to pass to the method. And you define what value is expected to be returned from the method. This is the expected result, that is compared to the actual result which is returned from the object.

This is the actual question that one has in regards to the data and the rules: „if I have data and test it using a certain condition, is the result the same that I expect?". The result is a [0] or [1] respectively [true] or [false], indicating in the test failed or not.

E.g. if my data is the word "europe" and my test condition is "is the data longer than 4 characters?", then the result would be a [1] indicating that the test successfully passed or the test condition is [true].

Now as already indicated earlier, the rule engine executes rules using certain checks and conditions. There are a lot of these checks already predefined and come with the program. How data is made available to the rule engine, is up to the user. The sample provided with the program is just one way how to do it. It is based on a CSV file, which is read and fed into the rule engine, one row after the other.

If you want to check data from a database e.g., then all you have to do is to get the data and capsule it in some java objects or you use e.g. a Java resultset. You reference these objects in the XML rule file (in the object tag) and define which predefined checks to use. When running your program you pass the data object to the rule engine, which in turn executes the rules.

Here is an excerpt of a Java code example of how to use the rule engine:

```java
// do some initial stuff here ....

// reader for the data file
BufferedReader reader = new BufferedReader(new FileReader(args[0]));

// variable to contain a row of data from the data file
String line;

// select all files in the given folder (args[1]) containing rule
// the definitions, but only those that end in .xml
File dir = new File(args[1]);
File[] fileList = dir.listFiles(new FilenameFilter() {
public boolean accept(File f, String s) {return s.endsWith(XML_FILE_EXTENSION);}
});

// create an engine object, passing a reference to the
// properties file (args[2)
BusinessRulesEngine engine = new BusinessRulesEngine(fileList, args[2]);

// splitter object will split the row from the datafile into
// its fields using - in this case - the default semicolon (;) seperator
Splitter splitter = new Splitter(Splitter.TYPE_COMMA_SEPERATED);
```

```
        // loop over all rows of the data file
        // output will go to the file as defined in the properties file or else
        // to the console
        while ((line=reader.readLine())!=null)
        {
                // only if the line is not empty
                if(!line.trim().equals(""))
                {
                        // get a row object containing the fields and data
                        Row row = splitter.getRow(line);
                        // run rule engine on this data
                        // the first parameter gives the object a label
                        engine.run("row: " + counter, row);
                }
        }

        // output the total number of rules
        int numberOfRules = engine.getNumberOfRules();

        // do some finalizing stuff here ....
```

The rule engine accepts either a single object or a collection of objects as parameter to its *run()* methods. The results of the execution of the rules are stored in the RuleExecutionCollection inside the rule engine. It contains RuleExecutionResult objects, with information on what was executed and the results of the execution. You can output the results or you may use them for further processing.

During execution of the rule engine, it outputs messages either to the console or to a file. The format of the output is dependant on how it is defined in the relevant message template. As rules belong to subgroups and subgroups belong to a group, in many cases you want to output messages that display if the group failed, if subgroups failed and if rules failed. Sometimes you might want to output also the case when the group passes or even both. You are free to define the output of the results by providing your own template or modifying the existing one.

The sample data

There is some sample data that comes with the business rule engine, that allows you to run it and see what it does. Once you have understood the basics, you can create your own small or bigger projects based on the business rule engine.

Data format

A file with a little bit more than 700 airports is available in the *samples/airport/data* folder. It contains the details of airports – one per row. The fields in each row are divided by semicolons (;). Airports have a so called ICAO code. This is a 4-letter code assigned to the airport which uniquely identifies the airport in the world. Each airport also has a name and an elevation which can be negative (meaning below sea level) in some parts of the world and lastly there is information available on the latitude and longitude position of the airport.

As you can see we can easily make out some rules that apply to the airport data as described above. E.g. the ICAO code field must be 4 characters in length. If the file contains data where this field is longer or shorter then that would be an error. We don't know much about the name field but maybe we want to make sure that it is not longer than a certain length, because the data will go into a system that has a limitation on the length of this field. Elevation can certainly be tested, if it is indeed a numeric value. We should also test, that the value is not too high and at the same time not too low. If

somebody entered a value but made a mistake and that system did not catch it, then we can do that here and ensure the quality of the data.

The latitude and longitude fields follow a certain formatting: the first field is the indicator for the direction, meaning north (N), south (S), east (E) or west (W). For the latitude field two numbers follow indicating the degrees, then two numbers indicating the minutes and then four numbers indicating the seconds and hundredth seconds. E.g. N32215055 means: North 32 degrees, 21 minutes and 50.55 seconds. Because longitude degree value go up to 180, the longitude field is one digit longer: three numbers indicating the degrees, two numbers indicating the minutes and four numbers indicating the seconds and hundredth of seconds. E.g. W064475480 means: West 64 degrees, 47 minutes and 54.80 seconds.

Now that we know the formatting of the latitude and longitude fields, there are quite a few checks we can do with the data. For example there are only four letters for the four directions that are used. If e.g. The value of the latitude field starts with a the letter „G" than that would be an error. Likewise, if we found a latitude degree to be higher than 90, we would want to know because it is simply not possible and would point us to a problem in the data.

Important note*: The data in the sample files is **not** for navigational purposes. Don't use it for other purposes than the examples shown here.*

CSV Reader

The CsvReader class is a sample class of how to work with the business rule engine. It allows you to quickly get started and determine, if the rule engine is useful for your purposes.

The example CsvReader reads the airports.csv file and creates Row objects. The row class is a very easy class that basically just serves as a container for the data. These row objects contain the fields of the rows which are divided by a separator (in this case a semicolon). So each field devided by a semicolon from the other is afterwards a field of the row object.

The reader instantiates the rule engine, which parses the XML files containing the rules and then executes the data against the rules. The output of the results is written into a file in the samples/airports/output directory.

I would recommend to look at the example data, the properties files and the rule file and of course to run the sample. Once you familiarized yourself, go ahead and change some data – specifically that data that is referenced in the rule XML file and re-run the sample. You will find different results/output depending on your changes. You are also free to change the sample by adding your own rules.

Apart from this CsvReader class, you are of course free to implement your own class and run the rule engine with it. Maybe you have a different format of your data that comes from file or a database. Or you might want to integrate the rule engine directly in your projects.

Sample console run

Below is the database sample shell script (*run_airport_check_database.sh*) as invoked from the console. It gives you an overview of what the rule engine did. The next section (*Sample Output*) shows the messages that are written to file when the program runs.

```
airports$ ./run_airport_check_database.sh
start:                    Sun Jul 06 19:10:58 CEST 2008
parsing xml rule files...
number of groups:         2
number of rules:          9
group logic:              group_2:  subgroup_1 ( rule_elevation_1 )
group logic:              group_1:  subgroup_1 ( rule_icao_0 and rule_icao_1 and
rule_name_0 and rule_name_1 and rule_elevation_0 and rule_elevation_1 ) and subgroup_2
( rule_latitude_0 and rule_longitude_0 )
number of lines of data:  737
output to        :        output/messages.txt
total number of rules:    6633
number of rules passed:   6629
number of rules failed:   4
number of groups failed:  4
end:                      Sun Jul 06 19:10:59 CEST 2008
elapsed time:             1 second(s)
end of process.
```

Sample Output

Find below the sample output of the rule engine, as run against the database sample. Please note that the output displays both if the group(s) failed and if the rule(s) failed. Which messages are displayed – failed ones, passed ones or both – is defined in the *engine.properties* file The output can be changed to your personal preferences as it is based on the Apache Velocity template engine.

```
row: 171 group_2: failed=[true]
       subgroup subgroup_1 failed: [true]
               rule rule_elevation_1 failed: [true] - Elevation field value [7014] is
               not smaller or equal than expected value: [7000]
row: 171 group_1: failed=[true]
       subgroup subgroup_1 failed: [true]
               rule rule_icao_0 failed: [false] - Correct uppercase of ICAO code: [KFLG]
               rule rule_icao_1 failed: [false] - Correct match of ICAO code field to
               expected value: [[A-Z]{4}]
               rule rule_name_0 failed: [false] - Correct length of name greater than
               expected value: [0]
               rule rule_name_1 failed: [false] - Correct length of name smaller than
               expected value: [41]
               rule rule_elevation_0 failed: [false] - Correct value of elevation
               greater than expected value: [-50]
               rule rule_elevation_1 failed: [true] - Elevation field value [7014] is
               not smaller or equal than expected value: [7000]
               subgroup subgroup_2 failed: [false]
               rule rule_latitude_0 failed: [false] - Correct match of latitude field to
               expected value: [[NS][0-9]{8}]
               rule rule_longitude_0 failed: [false] - Correct match of longitude field
               to expected value: [[WE][0-9]{9}]

row: 270 group_2: failed=[true]
       subgroup subgroup_1 failed: [true]
               rule rule_elevation_1 failed: [true] - Elevation field value [7284] is
               not smaller or equal than expected value: [7000]

row: 270 group_1: failed=[true]
       subgroup subgroup_1 failed: [true]
               rule rule_icao_0 failed: [false] - Correct uppercase of ICAO code: [KLAR]
               rule rule_icao_1 failed: [false] - Correct match of ICAO code field to
               expected value: [[A-Z]{4}]
               rule rule_name_0 failed: [false] - Correct length of name greater than
               expected value: [0]
               rule rule_name_1 failed: [false] - Correct length of name smaller than
               expected value: [41]
               rule rule_elevation_0 failed: [false] - Correct value of elevation
               greater than expected value: [-50]
               rule rule_elevation_1 failed: [true] - Elevation field value [7284] is
               not smaller or equal than expected value: [7000]
       subgroup subgroup_2 failed: [false]
```

```
              rule rule_latitude_0 failed: [false] - Correct match of latitude field to
              expected value: [[NS][0-9]{8}]
              rule rule_longitude_0 failed: [false] - Correct match of longitude field
              to expected value: [[WE][0-9]{9}]
```

Future plans

The web application which is available to define rules and actions needs to be enhanced and to be made more user friendly. Also a security model needs to be implemented.

Then of course bugs that are detected will be fixed and some new features will be made available.

Provide us with your feedback, so we can enhance the tool for the benefit of all.

Contact

If you find bugs or have comments or suggestions in regards to the rule engine or the documentation, please feel free to contact me:

Uwe Geercken
Datamelt
email: [uwe.geercken@w](mailto:uwe.geercken@web.de)eb.de
web: www.datamelt.com

Alphabetical Index

Action...8

Actions...8

Airports..4, 29

Apache Velocity......................................27, 28

Beanshell..5

Business logic..4, 5

Chained..4, 12, 27

Check..6

Checks.................................4, 5, 6, 8, 17, 28

Collection..29

Contact..33

Csv file..28, 30

CsvReader...30

Database.........................4, 5, 10, 11, 28, 30

Eclipse..5

Groovy..5

Intergroupoperator.........................11, 12, 15

Java code example..28

Jython...5

Message...............13, 18, 21, 22, 23, 24, 27, 29

Microsoft SQL Server....................................10

MySQL...10

NetBeans...5

Oracle..10

Quality...5

Reflection..11, 28

Regular expressions.......................................5

Replacements file...14

Row object...24

Ruleoperator. 11, 12, 15, 16, 17, 18, 21, 22, 23

SimpleAction...19

Subgroup..11, 12, 13

Subgroups..11, 12, 13

Templates..13, 27

Zip file..13