MaduraRules

User Guide

Document Revision: Revision: 1

Document Revision: Revision: 1

Table of Contents

1. Change Log

Author	Date	Comment
roger.parkinson	Thu May 01	[maven-release-plugin] prepare release madura-rules-2.2.2

2. References

[1] Spring Framework
[2] JAXB Plugins
[3] JAXB
[4] Hyperjaxb3
[5] JSR-303
[6] MaduraObjects
[7] GPL V3 licence
[8] Commercial Madura Rules Licence 1.0
[9] Apache Licence V2.0

3. The Concept

Madura Rules is a rules engine designed to work closely with Madura Objects. Where Madura Objects provides a transparent validation and metadata facility, Madura Rules is a plugin to Madura Objects that extends it to provide cross-field validation, dynamic metadata and dynamic data generation.

Let's look at a real scenario. You have an object called Customer and to that is linked some Address objects and some Invoice objects. These are all just simple Java beans with getters and setters. Well, they look like that at first. You actually defined them using an XSD file and generated the Java objects using JAXB[3]. Even better you have used the HyperJAXB3[4] and Madura Objects plugins to JAXB to add some extra hooks to those objects. You did not have to write the objects, you just had to define them in an XSD file.

So far this gives you the following features:

- The objects can be serialised to/from XML which is really handy if they need to be passed to PDF tools like FOP, and also useful for web services.
- The objects can be saved and fetched to a database with JPA.
- The objects will self validate. If you attempt to set a value that is incorrect they will throw an exception and the value will not be kept. For example you can set a range on a numeric value and this will automatically be checked.
- You can query the objects for metadata information, specifically for choices available.

If you add Madura Rules to this mix then you extend the validation to cross-field validations, as well as rules to manipulate the metadata. The rules operate totally transparently, except when they throw violation exceptions. All your applications see is a set of Java objects (POJOs). The rules are also able to manipulate metadata. For example they can eliminate some of the available options, set fields to inactive and active, or read only or required.

Each of the generated Java classes therefore exposes an interface for fetching this metadata. So your application can generate a drop-down list of the currently available options, or disable a field if it has been switched to inactive.

The rules are able to generate new data, for example deriving a discount rate based on a customer type, channel and current sale details.

4. The Rule Engine

Rules are just small pieces of code, usually conditional code, that can run independently. A simple example of a rule is 'if the customer type of this customer is A then set his business type to AG'. This needs to be translated into a syntax that the rules parser understands but for now notice that this is just a statement that we always want to hold true.

While the customer type has no value the rule can do nothing, it does not apply. Similarly if the customer type is set to B the rule cannot run, it still does not apply to the situation. But the moment customer type is set to A then the rule fires automatically and the business type is set to AG. The application code doesn't have to do anything to make this happen, it just has to set the customer type value on the object.

If the value of customer type is later changed to something else, ie not A, then the rule must be 'unfired'. That means that the values derived from it have to be unset, so the business type must be cleared, and this must happen automatically too.

Now imagine that there are many rules and that the action of one rule can cause other rules to fire, and that unfiring a rule can cause those rules to unfire as well. This is why we use a specialised engine to manage the rules.

It is important to remember that the rules engine decides what rules to fire and what order to fire them in. It is a common mistake to try to trick the engine into firing the rules in a particular order. This is not necessary. The rules are atomic and the engine can work out the interrelationships more easily than you can yourself.

What happens when there is already a value for the business type? If our example rule fires it will attempt to set a new value. If the new value is the same as the old value then nothing interesting happens. But if it is different then we have a problem. The problem is inconsistent data. The operator may have already given a value for the business type and that value does not work for the customer type they later gave. In this situation the rules engine throws an exception. The result of the exception is that the last value the user attempted to set is rejected and any values we derived from it are rolled back, leaving the session state as it was before.

Keeping the session state correct, even if it may still be incomplete, is part of the 'truth maintenance' system the rules engine implements. The other part is that when the rules can derive a value using what has already been set then this automatically happens.

To use Madura Rules we first assume you use Madura Objects already. So you have an XSD that you used to generate your business objects and these are now Java classes. If you want you could hand-annotate your Java classes but that is harder and gives no extra benefits. You want to do this the easy way, right?

The process of adding rules to your application requires the following steps which are detailed in later sections:

- Write the rules. This involves editing a text file that holds the rules. You can have multiple files if you want.
- Optionally define any external functions your rules require.
- Use the XJR maven plugin or ant task to generate Java from the rules file(s). You could write this Java by hand but it would be boring and hard to get the cross references right.
- Optionally define a file to hold your decision tables.
- Modify your Spring beans file to include the Madura Rules beans.

Notice that you did not have to change your application code. You can add more rules and change existing ones without having to make your application code aware of any changes at all. You can dynamically modify entries in the decision tables without even having to rebuild your application.

5. Writing Rules

5.1. Simple Rules

There are three types of rules:

- Rule: These are the classic if/then style of rule.
- Constraint: A condition that must be either true or not yet evaluated. If data is presented that makes this condition return false then the data is rejected.
- Formula: A simple algebraic formula which sets a value.

The syntax for these is very like Java. This is deliberate.

Best to start with an example:

```
formula: Customer "Count Items in list"
{
 invoiceCount = count(invoices);
}
```

This is the simplest kind of rule and consists mostly of one algebraic expression. If there is enough information to evaluate the expression the rule will fire and set the resulting value.

The `Customer` is the scope. There is a Customer class defined and any fields referenced in this rule are on some Customer object. There is also a message or comment associated with the rule. The formula in this case includes a function and functions are discussed later.

```
rule: Customer "Determine business from customerType"
{
 if (customerType == "A")
 {
  business = IndustryType.AG;
 }
}
```

This is the classic if/then rule. Like the formula it has a scope and a message. The body of the rule has a condition and one or more actions, ie formulae, to perform if the rule can be fired.

The rule engine detects any Customer object whose customerType field gets set to A and fires the rule automatically.

```
constraint: Customer "check the count: {0}" [oneInvoice.amount]
{
 !(invoiceCount > 2L);
}
```

A constraint also has a scope and a message. The body of the rule is a single condition. It can be a complex condition but it is just one condition. In this case it is saying the count of invoices must not be greater than 2. If we try to change the count to something greater than 2 then the constraint will fire and deliver an exception.

Each kind of rule has a message. The message is used if the rule attempts to fire but cannot and it has to deliver an exception. The message text ends up in the exception. Madura Rules actually makes use of Java's MessageFormat facility for this. Although you put a readable message into the rules file the rule generator takes your message and puts it into a properties file which ends up looking something like this:

```
nz.co.senanque.objecttestrules.R1=Count Items in list
nz.co.senanque.objecttestrules.R2=Determine customerType from name
nz.co.senanque.objecttestrules.R3=check the count: {0}
```

You are then free to translate this file into other languages for different locales. The exception generator will look up the key to get the actual string whenever it generates an exception.

Also note the use of arguments in the message strings. For R3 we specified a message with an argument. The arguments for the message are specified in the rules file just after the message. There is an optional list of fields comma separated and surrounded by brackets.

In our example we saw:

```
constraint: Customer "check the count: {0}" [oneInvoice.amount]
```

When the message is generated the `oneInvoice.amount` field is used as the argument. In this case `oneInvoice` is a field that points to an Invoice object and that has an `amount` field on it. You can refer to fields of owned objects but no further level of indirection is supported. This encourages encapsulation.

The general operation of the engine is that values are set from outside. The engine detects which rules ought to be fired because of those values and fires them. That, in turn, sets other values which causes other rules to fire etc. When all the rules that ought to fire are done then control is returned to the calling application (which doesn't actually know it called any rules).

If a constraint fails or if one of the other rule types has a problem with the data at any point the rule that detected the problem will throw an exception containing the message. The engine will roll back all changes, ensuring the current state is always valid and deliver the exception to the caller.

Apart from constraints failing there can be other reasons for rules to fail. For example if we set customerType to A but we had already set the business to something other than IndustryType.AG. We also fail if we have derived a value from the rules and then the caller attempts to set it directly.

If a value set by the caller changes then the rules are unfired and, possibly, refired to accommodate the change.

5.2. Functions

5.2.1. Internal Functions and Operators

There are many built in functions that you can use in your rules. These handle lists, dates and conversions. The conversion functions are often inserted automatically by the rules parser when it generates the Java form of the rule.

- sum(list.float) sums all the properties in a list eg total = sum(invoices.amount)
- count(list) counts all the properties in a list eg count = count(invoices)
- anyTrue(list) tests if any value is true eg x = anyTrue(invoices.flag)
- countTrue(list) counts all the true properties in a list eg count = countTrue(invoices.flag)
- allTrue(list) tests if all the properties in a list are true eg x = allTrue(invoices.flag)
- unique(list) tests if every item in a list is unique eg x = unique(invoices)
- match(list,list) tests if two lists match eg x = match(invoices,otherInvoices)
- yearsSince(date) number of years elapsed since the date eg years = yearsSince(dateOfBirth)
- monthsSince(date) number of months elapsed since the date eg months = monthsSince(dateOfBirth)
- daysSince(date) number of days elapsed since the date eg days = daysSince(dateOfBirth)
- subtractDays(date,days) eg date = subtractDays(dateOfBirth)
- addDays(date,days) eg date = addDays(dateOfBirth)
- toNumber(any) converts a value to a number if possible
- toDate(any) converts a value (inevitably a String) to a date
- toLong(any) converts a value to a long
- toString(date) convert a value to a String

- isNotKnown(any param) returns true if this parameter was explicitly set to Not Known. See *6*.

Similar to functions are the operators. The usual group are available:

```
+ - * / ^ % && || ! == => >= < >
```

These mean the same as they do in Java, with the exception of '^' which invokes the pow() function, eg 2^3 is 8.

Also note that dates and strings can be compared with the compare operators instead of calling a method.

5.2.2. External Functions

You can add your own functions if you want to. Here is what you do.

First, write your function as a static method on some class. It should look something like this:

```
@Function
public static Double combine(Number a, Number b)
{
    return a.doubleValue() + b.doubleValue();
}
```

Note that the function is annotated with `nz.co.senanque.rules.annotations.Function` and that the arguments are Number rather than Double or Long or double or long. This keeps the function flexible enough to handle multiple data types. You really only want to pass simple arguments such as:

- Number
- String
- Boolean
- Date

5.3. Decision Tables

The Decision Table feature is a way to represent a large number of relationships which could be done with a lot of if/then rules but it would be tedious to write and maintain.

You define an XML structure that looks like this:

```
<DecisionTable name="business-customerType" type="Customer"
 message="nz.co.senanque.newrules.decisiontable.business-customerType">
 <ColumnNames>
  <ColumnName autoAssign="true">business</ColumnName>
  <ColumnName>customerType</ColumnName>
 </ColumnNames>
 <Rows>
  <Row>
   <Column>AG</Column><Column>A</Column>
  </Row>
  <Row>
   <Column>AG</Column><Column>B</Column>
  </Row>
  <Row>
   <Column>FISH</Column><Column>B</Column>
  </Row>
  <Row>
   <Column>FINANCE</Column><Column>C</Column>
  </Row>
  <Row>
```

```
   <Column>FINANCE</Column><Column>D</Column>
  </Row>
  <Row>
   <Column>FINANCE</Column><Column>E</Column>
  </Row>
  <Row>
   <Column>FINANCE</Column><Column>F</Column>
  </Row>
 </Rows>
</DecisionTable>
```

The decision table has a name, a relevant object (Customer in this case) which is the equivalent of the scope in the other rules. There is also a message identifier which is delivered as an error if an attempt to set an incorrect value is made.

The column names refer to fields in the Customer object.

Below that are rows and columns. Each row specifies a valid combination. So if we set the business to B then the only valid values for business are AG and FISH. If your application examines the metadata for business, say to create a drop down list, then it will give only those values.

If we set the customerType to A then there is only one valid value for business and, because we set the autoAssign attribute in the columnName, then that value will actually be set.

We can, of course, decide to set the business value first and have it decide what options are available for customerType instead. And we can have more than two columns.

Your decision table data comes from the XML file by default, but you can specify a factory to deliver the data. That factory is a Java class you write that implements `nz.co.senanque.rules.factories.DecisionTableFactory`. Your factory will be called when the rules load up, not for every invocation of the decision table.

If you do use a factory to deliver the data you should define a workable data set in the XML anyway and use it for unit testing. That way your unit tests will not be dependent on external data sources. If no factory is configured the XML will be used and, if a factory is configured the factory will be used in preference.

5.4. Constants

Sometimes it is convenient to use soft constants in your rules like this:

```
rule: Customer "Determine business from customerType"
{
 if (customerType == ${xyz})
 {
  business = IndustryType.AG;
 }
}
```

To find the value of xyz the engine will look in an XML document that defines the constant like this:

```
<MaduraValidator>
 <Constants>
  <Constant name="xyz">aaaab</Constant>
 </Constants>
</MaduraValidator>
```

The XML can be changed without having to change the rules, which increases your deployment options. Like the decision tables these constants can be delivered from factories you supply, so the value of xyz might be determined by some Java code you write that runs automatically when the rules load. Note that the factory does not run after that and the value, once established, remains in use as a constant.

But you need to define the constant in the XML file regardless and it is convenient to use the value from there when unit testing.

6. Deploying Rules

This assumes you are deploying a Java application and that you are using Spring to wire it together. There are alternatives but the examples here use Spring. We also assume you have set up your project in line with the description in Madura Objects. So you have already defined your objects in an XSD file and you already have your Spring configuration file for that. All we need to do now is describe what else you have to do.

6.1. Building with Ant

First you need to generate your rules. This is done using an Ant task called XJR. Define the Ant task like this:

```
<taskdef name="xjr" classname="nz.co.senanque.generate.XJR">
  <classpath>
   <fileset dir="${basedir}/temp/lib" includes="*.jar" />
 <pathelement location="${basedir}/bin"/>
  </classpath>
</taskdef>
```

This assumes you have the madura-rules.jar file in temp/lib. The /bin entry is optional but you might want this if you have external function classes. The xjr task needs to have those on its class path. Now to invoke the task you do this

```
<xjr destdir="${basedir}/generated"
 packageName="nz.co.senanque.objecttestrules"
 rules="${basedir}/test/nz/co/senanque/rulesparser/ObjectTest.txt"
 schema="${basedir}/sandbox.xsd"
 xsdpackageName="nz.co.senanque.madura.sandbox">
 <classReference name="nz.co.senanque.sandbox.SampleExternalFunctions"/>
</xjr>
```

There are several arguments specified here:

- destdir is where the resulting Java files will be generated. They will be put into the package specified in packageName.
- rules specifies the file containing your rules.
- schema specifies the xsd file.
- xsdpackageName is only needed if your xsd package is different from your rules packagename.
- The optional classReference entry specifies a class containing your external functions as described in *5.2.2*. If you have multiple classes then add multiple classReference entries.

Once this task has run you will need to compile the generated Java using the usual javac task supplied with Ant.

6.2. Building with Maven

The one extra thing you have to do with Maven is generate the rules from your rules .txt file. There is a Maven plugin to do this and it is invoked by adding this to your pom file

```
<plugin>
    <groupId>nz.co.senanque</groupId>
    <artifactId>madura-rules-maven-plugin</artifactId>
    <version>1.0</version>
    <dependencies>
        <dependency>
            <groupId>com.sun.xml.bind</groupId>
```

```
            <artifactId>jaxb-xjc</artifactId>
            <version>2.2</version>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <type>jar</type>
            <version>0.9.24</version>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <goals>
                <goal>xjr</goal>
            </goals>
            <configuration>
                <targetDirectory>target/generated-sources/xjc</
targetDirectory>
                <rules>PizzaOrderRules.txt</rules>
                <schema>PizzaOrder.xsd</schema>
                <packageName>nz.co.senanque.pizzaorder.rules</packageName>
                <xsdpackageName>nz.co.senanque.pizzaorder.instances</
xsdpackageName>
            </configuration>
        </execution>
    </executions>
</plugin>
```

There are several arguments specified on the xjr plugin:

- destdir is where the resulting Java files will be generated. They will be put into the package specified in packageName.
- rules specifies the file containing your rules.
- schema specifies the xsd file.
- xsdpackageName is only needed if your xsd package is different from your rules packagename.
- The optional classReference entry specifies a class containing your external functions as described in *5.2.2*. If you have multiple classes then add multiple classReference entries.

Both of the schema file and the rules file are always in the directory `src/main/resources`.

6.3. Spring Configuration

This is what you add to the Spring configuration[1] you put together for Madura Objects:

```
<bean id="validationEngine"
 class="nz.co.senanque.validationengine.ValidationEngineImpl">
 <property name="metadata" ref="metadata"/>
 <property name="plugins">
  <list>
   <ref bean="MaduraRulesPlugin"/>
  </list>
 </property>
</bean>
```

We have added a bean to the list injected into the plugins property on the validation engine bean we already had for madura objects. The bean is defined like this:

```
<context:component-scan base-package="nz.co.senanque.objecttestrules"/>
```

```
<bean id="MaduraRulesPlugin" class="nz.co.senanque.rules.RulesPlugin"/>
```

The component scan allows Spring to find all the rules in the package named. You can scan as many packages as you like.

Madura objects needs a message source defined and you need to add another properties file to that:

```
<bean id="messageSource"
 class="org.springframework.context.support.ResourceBundleMessageSource">
 <property name="basenames">
  <list>
   <value>TestMessages</value>
   <value>nz/co/senanque/objecttestrules/messages</value>
  </list>
 </property>
</bean>
```

That extra properties file is generated when the Java for the rules is generated. It allows you to apply I18n to the messages you defined in your rules. You do have to write any extra properties files yourself to cope with other languages, but the first one is done.

There are some advanced options for configuring the rules engine:

```
<bean id="MaduraRulesPlugin" class="nz.co.senanque.rules.RulesPlugin">
 <property name="operations">
  <bean class="nz.co.senanque.rules.OperationsImpl">
   <property name="today"><value>07/18/2010</value></property>
  </bean>
 </property>
 <property name="decisionTableDocument" value="classpath:choices.xml"/>
  </bean>
 </property>
 <property name="constantsDocument" value="classpath:choices.xml"/>
  </bean>
 </property>
 <property name="decisionTableFactoryMap">
  <map>
   <entry key="business-customerType" value-ref="myDTFactory"/>
  </map>
 </property>
 <property name="constantFactoryMap">
  <map>
   <entry key="xyz" value-ref="myConstantFactory"/>
  </map>
 </property>
</bean>
```

All of these are optional. You can rewrite the OperationsImpl and inject your own, though you probably won't want to. In this example we are injecting the property 'today' with a fixed date. This is helpful in unit testing, allowing you to keep the date constant when functions like daysSince are running.

The decisionTableDocument is the XML file that contains all of the decision table data and the constantsDocument contains the constants. In practice you normally combine all of this information into one file and inject it into the choicesDocument property of the AnnotationsMetadataFactory bean you defined for madura objects.

If you want to use a factory to deliver the data for one or more decision tables you define a decisionTableFactoryMap. The key is the name of the decision table, as defined in the XML document. The bean is an implementation of `nz.co.senanque.rules.factories.DecisionTableFactory`. Because the name of the

decision table is passed to the factory the factory itself might be coded to handle more than one table. Your choice.

If you want to use a factory to deliver the data for one or more constants you define a constantFactoryMap. The key is the name of the constant, as defined in the XML document. The bean is an implementation of `nz.co.senanque.rules.factories.ConstantFactory`. Because the name of the constant is passed to the factory the factory itself might be coded to handle more than one constant. Your choice.

7. Designing Rule Based Systems

To do this properly you have to think a little differently to conventional applications.

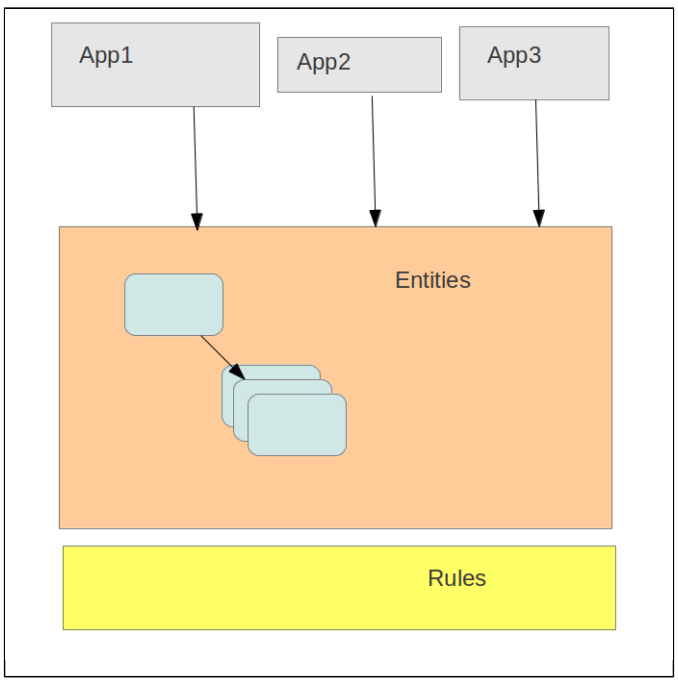

Figure (1) Architecture

Figure (1) shows the concept. The entities, or business objects are all available to the various applications that need to use them. There might also be some support code as well such as DAOs and helpers which is not shown here. The important part is that all the applications will use the same entities.

Those entities must, therefore, be quite free of application-specific code. This is one reason why the Madura approach is to generate them with JAXB. You can inject code using JAXB but it has to be done very consistently.

Using Madura we smarten up the entities by adding validation rules and (optionally) more complex rules. Validation rules are obvious enough and it is easy to see that the entities would want the same validation rules regardless of the application code that uses those entities.

It is important to notice that the application code never references the rules underlying the entities. This means you can change those rules without having to change any application code, and that means you can manage keeping the entities consistent across multiple applications as shown.

Just to make this clear consider the alternatives. The three applications are, for example, a web application, a web services application and a swing application. All of them create a Customer object and load various fields in it such as name, address, etc. These fields all need the same validation and these three applications would need to implement validation specific to their technologies. Except that they don't now because the validation rules handle it for them. All they see is an exception when they get something wrong. JSR-303[5] does this as well, but the three applications would have to call a JSR-303 provider to find out if the data is valid. With Madura this happens transparently.

By adding Madura Rules we can have more complex rules which handle cross-field validation. We can also have rules that derive values for other fields. For example if we are building up an order from several order items with different prices we might figure the total using rules. Adding another item or changing its value will automatically update the total. All three applications can use this logic without change. If you then need to add a tax calculation to figuring the total you can do this without having to work the code into the three applications. It will be transparent and consistent.

This means that there are a lot of things you might expect to have to code into the applications that no longer need to be there. It also means that some of the things you still do need in the applications can be done differently. Consider these examples:

- We already discussed the rule that figures the total of the invoice. So all you need in the application is some way to display the field containing the total. You do not need any code that adds it up. You still need code to add order items to the order, or whatever equivalent your application needs. Creating and removing objects is always an application operation, so the application creates the order item and adds it to the order. But then the rules kick in and add the order item amount to the total. If the order item amount changes (either because the application set it or because another rule set it because of something else) then the total will be refigured automatically.
- Setting a value in a field on the order item might cause the price of that order item to be set. For example if we have a size field on the order item then different sizes of this item might have different prices. Once the price is determined by the rules then the total will be revised. All the application had to do was pick the size.
- The size might also influence the availability of other fields. Perhaps size=Medium is only compatible with two of the five available flavours. If the application sets Medium then it must set one of those compatible flavours. If it sets one of the others you get an exception. Again there is no application code needed for this.
- You might want to present the current list of available values for flavour, though and that does need application code. Building a list of available values for a Swing application is totally different from building a Select box in a web application and in a web service application you probably don't do this because you have all the picks in the request. Any of these applications can query the Madura Objects API to find the available values and implement whatever makes sense to that technology. But what the application code did *not* have to do is figure out which available values there are. It just calls for the current list, whatever it is. Again, if you want to change the size/flavour relationships you don't have to change the application code. It is actually normal for those kinds of relationships to be driven by XML files, database tables or similar.
- What if we want to enable some user function using the rules? For example we might want to signal that this customer needs a credit check and that might be signalled by the order being over some critical value. We might want the user to see a credit check button enabled if that happens. But, clearly, that would have the rules tied to the UI and we never do that. What we do instead is set a boolean field in the Customer to true and tie the button enable to that field. Yes, you need application code to query the boolean and enable the button, but the application code does not have to decide what triggers the enable, it only knows about the boolean. Naturally you use the same field in the Swing application and the Web application. Notice that we did not put anything specific about the UI in the entities or the rules. But we did put enough information in there for different UI technologies to do what they need to.

At this point you may have noticed that the collection of entities and their fields are always in a valid *State*. The rules enforce this. Attempts to set invalid values (anything which violates the rules) results in the values being rejected and an exception thrown. The State remains valid although it may be incomplete.

This incompleteness is what the 'required' attribute is for. Some fields are required and some are not. Some may be dynamically made required by rules. An application can easily check for fields that are required but not completed and adjust the UI appropriately. Usually this means disabling the submit button.

There are some traps for the unwary in all this. Here they are and how to avoid them.

- Because there are things happening under the covers you can unwittingly strike performance problems unless you take some care. Be aware that every business object you bind into a session has an overhead. There are rules to fire and monitoring objects created to manage every field in the business objects. So, depending on your hardware you will probably manage hundreds of objects okay. But when it grows to thousands of objects you may strike problems.
- Performance again: adding a business object to a collection causes a bind, of course. Removing one causes an unbind. Binding and unbinding are expensive operations so don't bind, unbind and rebind things over and over.
- Java collections like to use the equals() method on the object when locating the target of a remove(). If you have two objects in the collection that have the same values in all their

fields then you may find it removes the wrong object. This is not a Madura problem, it is more of a Java issue. But it is confusing so watch for it. Mostly just make sure you have a unique field in each object.

- The rules are not threadsafe. Don't assume they are.
- You should assume that any rule might fire at any time, and that it might fire multiple times in a session as the user sets values, changes values etc. Beware of this if you decide to implement a custom operator. Custom operators should not perform expensive operations such as calling external web services. Probably database queries are okay as long as you enable caching.

We have already mentioned that the decision tables can be updated from any data source you want to use, this can be done while the application is running. The other rules are not so dynamic because they require Java generation. However the most flexible way to deploy rules into a running application is to use Madura Bundles.

Madura Bundles allows you to build a small jar file containing Java classes and resources and dynamically deploy it to a running application. The details are out of scope of this document, but this approach would permit you to hold a consistent set of rules, including decision tables, into a jar file and deploy them. You can arrange for existing sessions to continue using the rules they started with and new ones pick up the new rules.

The rules sometimes need intermediate values, for the same reason procedural code does. But you do not necessarily want those variables in your objects. As mentioned in the Madura Object documentation, these objects are HyperJAXB objects, which means they can be serialized to a database using JPA or to XML using JAXB. You probably do not want to include serialization of those intermediate fields. The answer is quite simple and delivered from HyperJAXB rather than Madura. Use the `Ignored` annotation like this:

```
<element name="weight" type="double">
 <xsd:annotation>
  <xsd:appinfo>
   <annox:annotate>
    <md:Unknown/>
   </annox:annotate>
   <hj:ignored/>
  </xsd:appinfo>
 </xsd:annotation>
</element>
```

This is a field that will come up in the next section. The point we are drawing attention to here is the `<hj:ignored/>`, which ensures that the field is annotated to ensure serialization is ignored for it.

8. Directed Questioning

Sometimes you have many, many fields for a user to give answers to but in practice only a few are relevant in any one case. It would be nice to allow the user to focus on just those fields and not be distracted by the others.

To achieve this you can use the Directed Questioning mechanism. The way it works is that your program focusses on one value that can be derived from the rules. The engine can work out what information is needed to obtain that one value and can dynamically tell you what fields are needed to obtain it. The fields may change depending on what answers the user gives to previous fields.

Here is a trivial example. Consider the following set of rules which calculate body mass index.

```
formula: Customer "BMI"
{
 bmi = weight / (height * height);
}
formula: Customer "Height Metric"
{
 height = heightMetric;
}
formula: Customer "Height Imperial"
{
 height = (heightFeet * 0.3048D) + (heightInches * 0.0254D);
}
formula: Customer "Weight metric"
{
 weight = weightKilos;
}
formula: Customer "Weight pounds"
{
 weight = weightPounds * 0.453D;
}
```

The main formula is in the first rule but to make it more interesting the basic information can be supplied in either metric or imperial units, whichever the user prefers.

The schema for this needs to have the `@Unknown` annotation on all of the fields mentioned, for reasons that will soon become apparent. So they all look something like this:

```
<element name="weightKilos" type="double">
 <xsd:annotation>
  <xsd:appinfo>
   <annox:annotate>
    <md:Label labelName="Weight (Kilos)" />
    <md:Unknown/>
   </annox:annotate>
  </xsd:appinfo>
 </xsd:annotation>
</element>
```

Now, what you want to know is the `bmi` so your code should look like this:

```
ValidationSession validationSession = m_validationEngine.createSession();
Customer customer = new Customer();
validationSession.bind(customer);
while ((fieldMetadata =
 m_rulesPlugin.getEmptyField(customer.getMetadata().getFieldMetadata("bmi"))) !
= null)
{
```

```
 log.debug("found field {}",fieldMetadata.getName());
 ... ask for the field
 fieldMetadata.setValue(some_value);
 or
 m_rulesPlugin.setNotKnown(fieldMetadata); // if the user indicates they
 don't know the answer
}
```

The first three lines are just normal interaction with the Validation Engine, nothing special. The fourth line tells the Madura Rules Engine to backchain on the field named `bmi` on the `customer` object. The engine will look for rules that output that field and try and fire them, and that will cause it to look for fields that rule needs to fire and so on. When it finds a field that has no output rules and no current value it will return a `FieldMetadata` describing that field.

You write whatever code you need to obtain a value for that field, call the `setValue()` method and loop.

When there are no more empty fields the `bmi` field will hopefully have a value.

The actual order of the questions should be regarded as undefined because you want to be free to change the rules around which will vary what order things are asked in. Also, depending on the earlier answers, some fields will not be asked for. With that in mind, let's walk through the example.

1. heightMetric. Answer is 1.9
2. weightPounds. Answer is 'Don't know', which means the engine has to try and find weight another way.
3. weightKilos. Answer is 90
4. Resulting bmi is 24.

Notice that it did not need to ask for heightInches or heightFeet because it already got height using heightMetric. So it can avoid asking for the fields it doesn't need.

If you want you can preload some answers before starting the sequence. For example if you already have set a value for weight then weight will not be returned by the `getEmptyField()` method and the user will not have to answer it. Users get particularly annoyed at having to answer questions they've already answered.

Now that this might not actually get you a value for `bmi`. If the user answers 'don't know' for too many questions then the rules will not be able to derive a value. In that case you probably have to inform them and go around again.

To go around again you have to reset the 'unknown-ness' of the fields. We have to assume that none of the answers given so far are actually of any use and we must lose the data in the fields as well as any 'don't know' flags we set. To do this you do the following:

```
m_rulesPlugin.clearUnknowns(customer);
```

The API for this calls the rule engine directly rather than just the objects that you saw in Madura Objects. This is because these operations are specific to the engine so it is appropriate for them to be dependent on it rather than being called indirectly and making Madura Objects more complex to handle it. Remember we may have more than one plugin servicing Madura Objects and they will each have their own specialisations.

Also relevant in this context is a `isNotKnown()` function that may be used in the rules.

```
rule: Customer "isnotknown"
{
 if (isNotKnown(weightKilos))
 {
  address = "not known rule fired";
 }
}
```

What this does is probably obvious enough. If `weightKilos` is set to NOT KNOWN, which is an explicit setting by the user then this rule will fire. NOT KNOWN and differs from it being UNKNOWN, which just means we don't have a value yet. The `clearUnknowns()` method sets the fields to UNKNOWN, which means none of them are now NOT KNOWN.

A tip worth mentioning here is that there is a hardly mentioned annotation available for your use. This is `@MapField`. You can annotate any field with this and supply it with text and you might use this to drive how you ask for the field.

Most likely, though, you just want to generate a prompt and an input field. The prompt can be fetched from `FieldMetadata.getlabel()`. If there is a list of possible values on this field you can fetch it using `FieldMetadata.getChoiceList()`.

Beware of using this with rules that cross objects. The `clearUnknowns()` method only clears one object and this may confuse things if there is a second object contributing information. You can clear multiple objects if you want but you do have to know what objects to clear.

You can also, of course, ask for multiple fields using the `getEmptyField()` method, just one at a time though. For example you might ask for `bmi` and then ask for a Ponderal Index (which is just BMI but the height is cubed). The fields that contributed to the BMI would also contribute to the PI and would not need to be reprompted. If one or two extra fields were needed for a PI then they would be prompted for, but not the others.

9. The Rules API

There is almost no API to consider here. The rules are invoked transparently as part of Madura Objects. There are just two exceptions to this. One is the Directed Questioning feature fully described in *6* and the other is a subtle side effect of the rules that you need to be aware of.

In Madura Objects a typical sequence in the life cycle of an object involves:

- Fetch or create an object and possibly attached objects.
- Create a Validation Session.
- Bind the object to the Session. This will bind any attached objects as well.
- Perform various rule-monitored operations. Other objects may be attached or removed during this phase and they will be added or removed to/from the Session as well.
- Save the resulting object structure.
- Close the Session. This will clean up the various internal structures associated with the Session and its monitored objects. It will not remove the monitored objects but it *will* reset them.

The particular thing to be aware of here is that closing the session does change the contents of the object and it ought not to be reused after that. For example you would definitely not want to swap the last two steps or you will save empty objects.

A. Licence

Madura Rules is licenced with the GPL V3 licence [7] by default. This means that any code you develop that makes use of Madura Rules must also be GPL, ie not proprietary code.

However, for projects that require a proprietary option we also offer Commercial Madura Rules License version 1 for a fee. If you are developing code that is not open source and uses Madura Rules you must accquire a valid License for all Developers who use Madura Rules in your project. Madura Rules may be used in many projects simultaneously without additional payments. The resulting project may be copied an unlimited number of times and deployed to an unlimited number of computers without additional payments.

All dependencies of Madura Rules are licenced with Apache Licence V2.0 or a compatible licence as specified in their pom files.

B. Release Notes

2.2.2

Upgraded Spring dependency.

2.2.1

Converted build to maven.

2.2

Added pom file for maven projects.

Fixed incorrect ivy conf for javax.xml.bind

2.1

Built with Java 1.7

2.0

Added Directed Questioning mechanism.

Reorganised tests to remove unnecessary interdependencies between them.

Improved handling of divide by zero.

Switched ProxyField references in rule generation to RuleProxyField references and added an exception trap for Unknowns.

Added the getEmptyField(FieldMetadata fm) method on RulesPlugin, which enables directed questioning.

1.1

Removed use of MessageSourceAccessorFactory because it does not play well with Madura Bundles.

Fixed a problem with rules attempting to attach to the wrong object type. Where this occurs the rule context is ignored, which gives the correct result. It happens when you have rules applying to an owner object, but you have chosen to not create the owner object, just the child. The net result is that the child rules are active but the owner rules are not.

1.0

Initial release

0.1

Fixed failure to fire relevant rules when an object is removed from a collection.

Fixed memory leak when removing objects from the session.

Fixed issues with the table constraint.

Rules can now refer to inherited fields.

Known problem: One of the tests in AllTests fails intermitently. I'm assuming this is some config issue I haven't quite solved in the project. It always runs fine when invoked directly from Eclipse. It also runs on, at most, the third try using ant. (This seems to be fixed, has not presented in recent builds)