MaduraObjects

User Guide

Document Revision: Revision: 1

Document Revision: Revision: 1

Table of Contents

1. References

[1] Spring Framework
[2] slf4j
[3] JAXB Plugins
[4] JAXB and Hibernate
[5] JAXB
[6] Hibernate
[7] Hyperjaxb3
[8] JSR-303
[9] Annox
[10] MaduraObjects
[11] MaduraBundles
[12] MaduraRules
[13] Apache Licence 2.0

2. Change Log

Author	Date	Comment
roger.parkinson	Sun Dec 01	[maven-release-plugin] prepare release madura-objects-2.2.1

3. The Concept

3.1. Business Objects

Imagine Business Objects looking like ordinary Java Objects, specifically they look like Java objects generated from JAXB[5]. That means there are setters and getters for the properties, there are no constructor arguments (so they are simple beans so far) and they use the array stuff that JAXB generates.

Since they actually *are* Java objects generated from JAXB they are backed by a schema file and they can be serialised to XML easily.

And since they are actually generated using the HyperJAXB3[7] plugin they are Hibernate[6] compatible.

So far, so standard. Programming with these beans takes ordinary Java skills. Designing the Business Objects in XSD takes a little more but nothing that cannot be picked up in an hour or two, especially if you already know Hibernate. Knowing Hibernate is a more serious requirement but Hibernate skills can be reasonably expected.

But we can add a little more. Using another JAXB plugin called Annox[9] we can add annotations to the business objects generated from the XSD. This means that fixed metadata (as opposed to dynamic metadata) can be added to the business objects. The best example is a field label.

Now for the cool bit. The Business Objects need to self-validate, and they need to self-validate based on the whole object graph they are in. So, for example you have a Customer object with attached Invoice objects. The total in the customer should calculate automatically as invoices are added. The DAO program that operates the objects does not need to know anything about this.

Also, if the DAO tries to set a value that is invalid in some way the Business Object will throw an exception. The attempted value will not be retained.

So the collection of related Business Objects, we will call this collection a *case*, is always *valid*, although it may be *incomplete*.

To achieve this we use a bunch of business rules that are run in pluggable rules engines. Depending on the need different rules engines can be plugged into the framework, or none at all. The latter case makes testing simple. The Business Objects behave almost like ordinary Java Objects when no engine is present.

3.2. Metadata

This was touched on in the previous section. Metadata is very important. People write lots of code to manage things that could instead be driven by metadata. There are two basic kinds: static and dynamic. Static is simple enough. It is easily handled by annotations. You need a label for a field? Put it in an annotation. You need some processing instruction for a treewalker that looks at this field? Put it in an annotation. These are static.

Dynamic metadata might cover the following:

- Sometimes a field is available/applicable, sometimes not. It depends on other data.
- Sometimes it is read-only.
- There might be a list of valid values. This might be static, in which case it is just an enum, but sometimes it changes, then it is dynamic metadata.

3.3. Validation Engine

The Validation Engine handles simple validation, which means validating fields in isolation from each other.

It can handle a number of validation requirements based on static metadata:

- Field length (min/max).
- Number of digits (integer/fractional).
- Email: is this a valid email address format?
- Range: min/max inclusive/exclusive.

- Matching a Regex expression

These are loosely based on JSR-303 [8] but not the same. Why not? There are good reasons.

- The JSR-303 definitions, especially as implemented in Hibernate Validation, is designed to be called explicitly to validate some objects you have already loaded with data. Madura Objects works differently. The data is actively and transparently validated behind the setters. So at no time is there ever invalid data in the objects.
- The Madura Objects validators have a more obvious way to specify the error messages.

This is not to say anything against JSR-303. Just that Madura Objects took a different approach.

Like most of the JSR-303 frameworks you can add your own annotations/validators where you need to.

3.4. Plugins

Madura Objects can be injected with plugins which are used to do more than simple field validation. The obvious example is cross-field validation but they might be used to integrate specialised engines, perhaps to derive a price for an order described by the bound objects, or perhaps to assess risk. The plugins must implement the `nz.co.senanque.validationengine.Plugin`.

Do not confuse these plugins with the JAXB plugins which are used to assist in generating the Business Object classes. Those are used at generation time, the MaduraObjects plugin is used at run time. There is a JAXB plugin that is part of Madura Objects, but that is not what is being discussed in this section.

Different plugins may be active at the same time, but they must not intersect. That is: they must not overwrite each others' data.

Like the validation engine the operation of the plugins is completely invisible to the code driving the business objects.

The rules engines are all optional. You can have none if you want.

3.5. Advantages

The advantages of all this should be obvious but let's spell them out:

- There is no API to learn. It is just ordinary Java. Not quite true, as we shall see, but the API is much smaller than Hibernate. Almost all the time you are just operating simple Java objects.
- Serialising to XML and back for web service messages etc is easily handled by standard JAXB. The other main use for this is generating XSL/FO reports.
- Database is handled by Hibernate (thanks to HyperJAXB3).
- Objects are defined outside of Java, in an XSD file. This means they get generated and they cannot be messed about with by people adding code to them when they shouldn't.
- Simple objects means simple DAOs and simple UI code.

3.6. Examples

This is what a small program looks like that uses these objects:

```
// Create a new session using the (probably injected) engine.
ValidationSession validationSession = m_validationEngine.createSession();

// create a customer using the (probably injected DAO)
Customer customer = m_customerDAO.createCustomer();
// This tells the validation session about the object
validationSession.bind(customer);
Invoice invoice = new Invoice();
invoice.setDescription("test invoice");
// Attached objects are automatically added to the session
customer.getInvoices().add(invoice);
boolean exceptionFound = false;
```

```
try
{
 // Setting an invalid value...
 customer.setName("ttt");
}
catch (ValidationException e)
{
 // ...results in an exception
 exceptionFound = true;
}
assertTrue(exceptionFound);
// But valid values are fine
customer.setName("aaaab");
customer.setBusiness(IndustryType.AG);
// save the customer to database
long id = m_customerDAO.save(customer);
```

The code in the DAO is not very complicated, but it is tidier to keep it together. The following leaves out the obvious things like imports and getters and setters.

```
public Customer createCustomer() throws Exception
{
 // The object factory is generated by JAXB and injected into the DAO.
 // You could just do a 'new Customer()' actually but you might want to
 use
 // other JAXB plugins that add code to the object factory so this ensures
 that
 // will always work.
    return getObjectFactory().createCustomer();

}
/**
 * This is ordinary hibernate code. It just saves the customer
 * It uses Spring's Transactional notation, but you can manage the
 transaction
 * any way you like.
 **/
@Transactional
public long save(Customer customer)
{
    Session session = getSessionFactory().getCurrentSession();
    session.saveOrUpdate(customer);
    session.flush();
    long customerId = customer.getId();
    return customerId;
}
/**
 * More ordinary hibernate code. It reads the customer.
 * It uses Spring's Transactional notation, but you can manage the
 transaction
 * any way you like. Getting the size of the invoices collection is just a
 way of forcing an eager
 * fetch. The validation engine cannot validate things that have not been
 fetched.
 * There are other ways of doing eager fetches but I got lazy.
 **/
@Transactional(readOnly=true)
public Customer getCustomer(long id)
{
```

```
    Session session = SessionFactoryUtils.getSession(getSessionFactory(),
 false);
    Customer customer =
 (Customer)session.get("nz.co.senanque.madura.sandbox.Customer", id);
    customer.getInvoices().size();
    return customer;
}
```

4. XSD File

Now let's take a look at the XSD file. First we have to get the header right. Specifically we must define the xjc, annox and md namespaces:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.example.org/sandbox"
 xmlns:tns="http://www.example.org/sandbox"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:jaxb="http://java.sun.com/xml/ns/jaxb" jaxb:version="2.0"
 xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
 jaxb:extensionBindingPrefixes="xjc hj annox"
 xmlns:hj="http://hyperjaxb3.jvnet.org/ejb/schemas/customizations"
 xmlns:annox="http://annox.dev.java.net"
 xmlns:md="http://annox.dev.java.net/
nz.co.senanque.validationengine.annotations"
 elementFormDefault="qualified">
```

The hj namespace is for HyperJAXB3, annox allows us to define miscellaneous annotations in the generated Java and md allows us to add the Madura-specific validations.

Just after that you probably need to add this:

```
  <xsd:annotation>
    <xsd:appinfo>
      <jaxb:globalBindings generateIsSetMethod="false"
 localScoping="toplevel">
        <jaxb:javaType name="java.sql.Timestamp"
         xmlType="xsd:dateTime"

 parseMethod="nz.co.senanque.validationengine.ConvertUtils.parseDateTime"

 printMethod="nz.co.senanque.validationengine.ConvertUtils.printDateTime" /
>
        <jaxb:javaType name="java.util.Date"
         xmlType="xsd:date"

 parseMethod="nz.co.senanque.validationengine.ConvertUtils.parseDate"

 printMethod="nz.co.senanque.validationengine.ConvertUtils.printDate" />
       <xjc:serializable/>
      </jaxb:globalBindings>
      <jaxb:schemaBindings>
        <jaxb:package name="nz.co.senanque.madura.sandbox"/>
      </jaxb:schemaBindings>
    </xsd:appinfo>
  </xsd:annotation>
```

This ensures that dates and dateTime fields are mapped to java.util.Date and java.sql.Timestamp objects. By default JAXB will use a Gregorian date object which you might want, but I prefer this.

An object in the XSD is defined like this:

```
    <complexType name="Customer">
     <sequence>
      <element name="id" type="long">
       <xsd:annotation>
        <xsd:appinfo>
```

```
      <hj:id>
       <hj:generated-value strategy="AUTO"/>
      </hj:id>
        </xsd:appinfo>
       </xsd:annotation>
      </element>
   <element name="version" type="long">
    <annotation>
     <appinfo>
      <hj:version/>
     </appinfo>
    </annotation>
   </element>
     ... other field definitions...
     </sequence>
    </complexType>
```

This is typical for HyperJAXB3 use. The two fields provide an id and a version and they tell Hibernate to auto-generate the id field and manage the object through the version field.

Now we can look at individual fields.

```
<element name="key">
 <xsd:annotation>
  <xsd:appinfo>
   <annox:annotate>
    <md:Label labelName="key"/>
    <md:Range maxInclusive="100"/>
   </annox:annotate>
  </xsd:appinfo>
 </xsd:annotation>
 <simpleType>
  <restriction base="string">
   <maxLength value="5"></maxLength>
  </restriction>
 </simpleType>
</element>
```

This defines a field called 'key'. We use the native xsd definitions to specify a maxLength. These are propagates through to Madura Objects as annotations. But the xsd definitions contain only a few validation definitions. So we supplement them using annox and the md tags Label and Range. Annox just propagates the annotation through to the target classes, which means you can write your own annotations and use this technique in the same way Madura Objects does.

In this case we specified the label name and the maximum (inclusive) value we accept for this field.

```
<element name="name">
 <xsd:annotation>
  <xsd:appinfo>
   <annox:annotate>
    <md:Label labelName="xxx"/>
    <md:Description name="this is a description"/>
    <md:MapField name="whatever"/>
    <!--
    <md:Regex pattern="a*b" message="hello world"/>
    <md:length maxLength="30" message="value=30"/>
    -->
   </annox:annotate>
  </xsd:appinfo>
 </xsd:annotation>
```

```
 <simpleType>
  <restriction base="string">
   <maxLength value="30"></maxLength>
   <pattern value="a*b"></pattern>
  </restriction>
 </simpleType>
</element>
```

This example expands on the previous one. As before you can see a label defined as well as two new annotations. A complete list of annotations is in *5.4*. But the reason for this example is the commented out section showing Regex and Length. These do the same thing as the restrictions under the simpleType tag, except they allow you to supply a message to deliver if the validation fails.

```
<element name="amount">
 <xsd:annotation>
  <xsd:appinfo>
   <annox:annotate>
    <md:Label labelName="Amount"/>
    <md:Digits integerDigits="8" fractionalDigits="2"/>
    <md:Range minInclusive="100" maxInclusive="1000"/>
   </annox:annotate>
  </xsd:appinfo>
 </xsd:annotation>
 <simpleType>
  <restriction base="double"/>
 </simpleType>
</element>
```

Numeric fields often need the number of digits validated. This allows the field to be in the format ######.##.

```
<element name="customerType">
 <xsd:annotation>
  <xsd:appinfo>
  <annox:annotate>
   <md:Label labelName="Customer Type"/>
   <md:Inactive/>
   <md:ChoiceList name="customerType"/>
  </annox:annotate>
  </xsd:appinfo>
 </xsd:annotation>
 <simpleType>
  <restriction base="string">
   <maxLength value="30"></maxLength>
  </restriction>
 </simpleType>
</element>
```

In this example you can see the Label being used again. In practice most fields need labels. We have also used the Inactive annotation. Inactive is part of dynamic metadata. What we are saying here is that we expect the field to be initially inactive, and a plugin can change it to active where required.

There is also a ChoiceList annotation with a name. ChoiceLists are described in *5.5*

One more example:

```
 <element name="business" type="tns:IndustryType"/>
 ...
    <xsd:simpleType name="IndustryType">
  <xsd:restriction base="xsd:string">
```

```
   <xsd:enumeration value="agriculture"/>
   <xsd:enumeration value="fish"/>
   <xsd:enumeration value="finance"/>
  </xsd:restriction>
 </xsd:simpleType>
```

There is actually nothing particularly special about this example from a Madura Objects point of view. JAXB will turn this into an Enumerated field. The reason for mentioning it is that Madura Objects is quite happy with this, including treating the list of values as dynamic metadata where required, just the same as ChoiceLists.

Note that if you do not add any validation or metadata information to the field then it will be ignored by Madura Objects. This is allows you to optimise the validation engine a little, ie ignored fields have no overhead.

4.1. Building with ANT

To invoke XJC from Ant use a command like this:

```
<taskdef name="xjc" classname="com.sun.tools.xjc.XJCTask">
  <classpath>
   <fileset dir="${basedir}/temp/lib" includes="*.jar" />
  </classpath>
</taskdef>

<target name="generateObjectsFromXSD">
 <delete dir="${basedir}/generated" failonerror="false"/>
 <mkdir dir="${basedir}/generated"/>
 <xjc extension="true" destdir="${basedir}/generated">
  <arg line="-verbose -Xequals -Xmadura-objects -XtoString -XhashCode -
Xannotate -Xvalidator -Xhyperjaxb3-ejb"/>
  <schema dir=".">
    <include name="${xsdfile}"/>
  </schema>
 </xjc>
</target>
```

The taskdef needs to refer to the jar files that the CJC task depends on. These can be fetched using ivy, specifying this dependency:

```
<dependency org="nz.co.senanque" name="madura-objects" rev="1.0"
 conf="mygenerate->generate"/>
```

Adjust the rev and local config (mygenerate) as necessary.

The actual xjc command shown specifies the following plugins:

equals	Adds the equals() method to the generated objects.
toString	Adds the toString() method to the generated objects.
hashCode	Adds the hashCode() method to the generated objects.
annotate	This is the annox plugin that copies annotations from the XSD file to the generated objects.
validator	One of the Madura Objects plugins. This one identifies information in the 'restriction' tag in the XSD and turns it into annotations.
madura-objects	The main Madura Objects plugin. This is the one responsible for modifying the getters and setters adding the metadata interface.

4.2. Building with Maven

To invoke XJC from Maven you use the JAXB plugin, and to use Madura Objects you define it like this in your pom file:

```
<plugin>
    <!-- jaxb plugin -->
    <groupId>org.jvnet.jaxb2.maven2</groupId>
    <artifactId>maven-jaxb2-plugin</artifactId>
    <version>0.8.3</version>
    <dependencies>
        <dependency>
            <groupId>nz.co.senanque</groupId>
            <artifactId>madura-objects</artifactId>
            <version>2.2</version>
        </dependency>
        <!-- You may want to tweak the jaxb version -->
        <dependency>
            <groupId>com.sun.xml.bind</groupId>
            <artifactId>jaxb-xjc</artifactId>
            <version>2.2</version>
        </dependency>
        <!-- Use the logging implementation you prefer -->
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <type>jar</type>
            <version>0.9.24</version>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
            <configuration>
                <extension>true</extension>
                <!-- the package for the generated java classes
 <generatePackage>nz.co.senanque.pizza</generatePackage> -->
                <!-- If the following not specified all xsd in resources
 are included -->
                <schemaIncludes>
                    <include>PizzaOrder.xsd</include>
                </schemaIncludes>
                <!-- if you don't want old output -->
                <removeOldOutput>true</removeOldOutput>
                <!-- if you want verbosity -->
                <verbose>true</verbose>
                <args>
                    <arg>-extension</arg>
                    <arg>-Xequals</arg>
                    <arg>-XtoString</arg>
                    <arg>-Xannotate</arg>
                    <arg>-XhashCode</arg>
                    <arg>-Xhyperjaxb3-ejb</arg>
                    <arg>-Xmadura-objects</arg>
                    <arg>-Xvalidator</arg>
                </args>
            </configuration>
```

```
        </execution>
    </executions>
</plugin>
```

Most of this is the normal use of the JAXB plugin, but it is worth noting that we are using two plugins to JAXB: madura-objects and validator, these are both contained in the madura-objects dependency.

The JAXB plugin expects the schema file to be in the directory `src/main/resources`

5. API

How much API did we see in the examples in *3.6*? The validation engine object has a createSession method. We also have a bind method on the validation session.

Yes, that is about it.

Objects that have just been fetched from Hibernate or unpacked from XML by JAXB need to be bound. This effectively locks the session onto those objects and starts it monitoring it.

The session also has a getMetadata method which is used to get information about the object. The details of this will come later but the field label, currently valid options, and various flags can be fetched using this. The metadata will change as the rules fire, so some fields will become active or inactive etc.

5.1. Object Interface

Each object does have to implement a small interface. This is generated by the XJC (JAXB) plugin so is no big deal. The following methods are generated:

getMetadata Fetches metadata about the fields in this object.

setValidationSession This is used internally.

Metadata is a list of objects describing the fields on the object. It is a map keyed by the field name. Each entry holds an object that contains:

- Active (boolean). Dynamic. The rules may make this field inactive. That means that setters cannot write to it and getters cannot fetch it. UIs should not display it, they would get an error, of course.
- ReadOnly (boolean). Dynamic. The setters will throw an error if this is true.
- Required (boolean). Dynamic.
- Label. Defined in the XSD but it may be translated using I18n.

The items marked dynamic may be set in the XSD and/or they may be modified by validation plugins dynamically.

5.2. Session

Like Hibernate there is a session, a Madura Objects session is called a ValidationSession. It has the following methods:

bind Binds a ValidationObject to the session, including all its attached objects.

close Closes a the session.

isEnabled/setEnabled Once the session is disabled the Validation Objects behave like ordinary POJOs. This is useful if you have some operations to do that don't need the overhead of validation.

5.3. SessionFactory

This gets us a ValidationSession.

createSession We pass a userDTO containing this user's roles. The rules may make use of these to determine readonly behaviour etc. Returns a session. Optionally pass a UUID. The session is stored on a threadlocal for getSession to use.

getSession Gets the current session from ThreadLocal. Static method.

As well as these there are various setters that define the package(s) containing the Business Objects, the rules engines to attach to it and the locking behaviour of the persistence. These are intended to be wired in rather than driven directly from the application.

The persistence for the Session is the calling application's responsibility. You probably want to save them to Hibernate, or you can serialise to XML and save them in that form.

When you use MaduraBundles[11] the session factory is bundle aware. A new session is created with the default bundle. An existing session is run against the bundle it was created with.

5.4. Annotations

This is the complete list of annotations supported out-of-the box by Madura Objects. You can add your own to this list and how to do that is described *6*. This is a brief summary, for more details see the Javadocs.

Static Metadata	Arguments	
@Label	labelName	Adds a label to the field.
@ChoiceList	name	Name of a list defined in choicesDocument. See *5.5*
@Secret		Not used directly but UI frameworks use this for rendering
@Ignore		Put this on fields you want Madura Objects to ignore. This is usually a relationship to another object which might become circular or perhaps because there is a lazy database fetch. Naturally any rules across that relationship cannot be honoured.
@Description	name	Misc description field, probably use for comments
Dynamic Metadata	**Arguments**	**Description**
@Inactive		Turns a field to inactive by default
@ReadOnly		Turns a field to read only by default
@Required		Turns a field to required by default
Validation	**Arguments**	**Description**
@Digits	fractionalDigits, integerDigits, message	The integerDigits argument is optional and defaults to zero.
@Email	message	Checks for a valid email format.
@Length	maxLength, minLength, message	Checks the length of the field. minLength is optional and defaults to zero. Same as xsd maxLength.
@Regex	pattern, message	Validates against a regex pattern. Same as xsd pattern
@WritePermission	name	Not used directly but UI frameworks use this for rendering
@ReadPermission	name	Not used directly but UI frameworks use this for rendering
@XmlElement	defaultValue	Supplies a default value for this field

The Metadata entries in the table are all available using the Metadata API.

The Validation entries all accept a message argument. In every case the validators look up the message in the `org.springframework.context.MessageSource` passing the current locale as well as arguments for the specific message. See the Javadocs for the annotations for details of which arguments are needed where and what the default messages are. You will want to customise the generic messages to your own needs.

Also note that the labelName for the field is normally one of the message arguments. When the validation fetches a label it tries to interpret it as another lookup in the MessageSource. If it does not find it then it uses the original string, if it does find it then it uses the translated string.

The default value is specified using `@XmlElement` because the various XML systems, such as JAXB, make use of this already. To specify a default in the XSD file use

```
<element name="myfield" default="400">
```

```
...
</element>
```

The generated class has code injected into the constructor to set all the default values automatically, conversion from the string representation in the file to the datatype of the field is also handled there.

5.5. Choice Lists

When a field has a discrete list of values then it can be defined as a choice list. UIs usually represent choice lists as a drop down list or radio buttons.

There are two ways to define choice lists in Madura Objects. The simplest way is to have them defined as Enum fields. See *3.6* for an example. This approach is fully supported by Madura Objects (which was easy because it is fully supported by JAXB and Hibernate and HyperJAXB3). You use this approach when you have no need to change the values in the list without recompiling your application, which is the usual case.

Sometimes the values may be more dynamic and need to be defined outside the application. In this case you need to use this notation in your XSD file.

```
<md:ChoiceList name="customerType"/>
```

Then you supply an XML file containing the choices. Here is a sample:

```
<MaduraValidator>
 <ChoiceList name="customerType">
  <Choice name="a">A</Choice>
  <Choice name="b">B</Choice>
  <Choice name="c">C</Choice>
  <Choice name="d">D</Choice>
  <Choice name="e">E</Choice>
  <Choice name="f">F</Choice>
 </ChoiceList>
</MaduraValidator>
```

Note the name 'customerType' matches the name in the XSD file. Now you can inject the XML file into the validation engine and it can reside outside your application.

By default it will retain these values until you restart your application. There are ways to manage this more aggressively using Spring that are out of scope for this document.

The names on each entry in the choice list are actually treated as names that can be looked up by locale and translated into messages. The underlying value is the value that is ultimately set in the field.

By implementing the appropriate plugin you can adjust the available values in any choice list dynamically. This applies to Enum choice lists as well.

The choice list contents can be delivered in other ways. See *7*

6. Validation Plugins

The term *plugins* gets used a lot. MaduraObjects itself is a plugin to JAXB and JAXB itself has a plugin to maven. But MaduraObjects has plugins as well, which is what we are talking about here, but the nesting here is getting quite deep.

The plugins at this level are called Validation Plugins because they assist MaduraObjects with validation, among other things.

A core concept here is that these plugins are transparent. The Java code that uses the business objects is unaware of them, though it does get exceptions delivered when something goes wrong. And it is possible to have multiple different plugins monitoring the objects. There are several choices that can be made here.

The simplest case is nothing at all. No validation session is bound to the objects and they really are POJOs, with the exception that they are available to both Hibernate and JAXB for serialisation.

The next simplest case is to use just the validation engine without plugins. With this in place single fields are validated transparently, but no cross-field validation takes place. For example you can add a list of valid values (a fixed list, which is not different to an enum), a regex expression, length checks on strings and min and max checks on values. Violation of these rules will reject the proposed value and deliver an exception. On its own this is worth having.

Then you can add a plugin. The facilities available will depend on the plugin chosen, but in general the plugins either refuse updates that violate constraints, derive other values from the ones already given, or both.

Finally you might decide to use multiple plugins on the same data. You would do this if one plugin was better at expressing some part of the problem than another, or had some facility required that the first one was missing. There is a down side to doing this. You will have to maintain two different plugin configurations and there must be no intersection between the set of fields mapped to one rule engine and the set of fields mapped to another.

When considering constraint engine plugins we need to understand several concepts. Most rules/ constraint engines come out of academic needs targeted at solving complex mathematical problems such as scheduling. They do a lot of interesting things that are not a much use to what is essentially a complex validation exercise. For example they might emphasise developing specific constraints to solve a timetabling problem which is run once and delivers several solutions. But for validation we need to develop one set of rules that will remain stable over many similar problems and we only need one solution. In fact one of our primary goals is to reject new data that is inconsistent with the existing data. Classic constraint engines will often simply deliver a 'no solution' result in that case.

So we are not quite looking for a classic constraint engine.

Forward chaining engines are useful for deriving further values from the given data, for example knowing that we have a specific product in our solution might fire a rule that calculates a price for it.

Backward chaining engines are less useful. Classic backward chaining assumes you can prompt for data you do not yet know but which will be needed to supply the answer requested. The backward chainer can direct the questioning to avoid asking questions that are not necessary, and only asking ones that are. But it can only ask one question at a time, making it difficult to design user interface that works well.

Some engines combine the three in a useful way, for example MaduraRules [12]

To add a plugin to Madura Objects you implement the `nz.co.senanque.validationengine.Plugin`.

This interface gives enough access to the validation engine to allow the plugin to alter the dynamic metadata, including restricting the list of available values on choice lists based on other values in the session.

7. Spring

Use SpringFrameworks to pull all this together. This is the basic configuration you need:

```
<bean id="validationEngine"
 class="nz.co.senanque.validationengine.ValidationEngineImpl">
 <property name="metadata" ref="metadata"/>
 <property name="plugins">
  <list>
   <!-- Optional list of rules engines for cross-field validation -->
  </list>
 </property>
</bean>

<bean id="metadata"
 class="nz.co.senanque.validationengine.metadata.AnnotationsMetadataFactory">

 <!--
 package name where business objects are found
 You can specify 'packages' and give a list of packages
 and you can specify 'classes' and give a list of classes.
 If package, packages and classes will merge their unique results so you
 can use all three at once if you must.
 -->
 <property name="package" value="nz.co.senanque.madura.sandbox"/>

 <!--
 List of all field validators
 This is optional and you can add your own and/or replace these ones
 with custom validators. You specify the full class name.
 -->
 <property name="fieldValidators">
  <list>
   <value>
   nz.co.senanque.validationengine.fieldvalidators.RegexValidator
   </value>
   <value>
   nz.co.senanque.validationengine.fieldvalidators.LengthValidator
   </value>
   <value>
   nz.co.senanque.validationengine.fieldvalidators.RangeValidator
   </value>
   <value>
   nz.co.senanque.validationengine.fieldvalidators.EmailValidator
   </value>
   <value>
   nz.co.senanque.validationengine.fieldvalidators.DigitsValidator
   </value>
  </list>
 </property>
 <!-- The valid choices for fields can be defined in this document-->
 <property name="choicesDocument">
  <bean class="nz.co.senanque.madura.spring.XMLSpringFactoryBean">
   <property name="fileLocation" value="/choices.xml"/>
  </bean>
 </property>
</bean>
```

```
<bean id="messageSourceAccessFactory"
 class="nz.co.senanque.localemanagement.MessageSourceAccessorFactory"/>
<bean id="messageSource"
 class="org.springframework.context.support.ResourceBundleMessageSource">
 <property name="basenames">
  <list>
   <value>Messages</value>
  </list>
 </property>
</bean>
```

There is not much to this, although you will want to do more if you are using Hibernate etc, and you probably are. See the source for a complete example and runnable test. The messageSource bean needs to be loaded with some specific messages (in this case held in the file ValidationMessages.properties). These are the requied contents:

```
nz.co.senanque.validationengine.regex=Failed Regex parse: label={0}
 attempted={1}
nz.co.senanque.validationengine.length=Failed length: label={0} minlength
 value={1} maxlength value={2} attempted={3}
nz.co.senanque.validationengine.digits=Failed Digits: label={0}
 integerDigits value={1} fractionalDigits value={2} attempted={3}
nz.co.senanque.validationengine.email=Failed email: label={0} not a valid
 email address, attempted={1}
nz.co.senanque.validationengine.choicelist=Failed choicelist validation:
 label={0} attempted={1}
nz.co.senanque.validationengine.derivedValue=Cannot set derived field.
 label={0} attempted={1}
nz.co.senanque.validationengine.range.maxExclusive=Failed range: label={0}
 must be less than {1} attempted={2}
nz.co.senanque.validationengine.range.maxInclusive=Failed range: label={0}
 must be less than or equal to {1} attempted={2}
nz.co.senanque.validationengine.range.minExclusive=Failed range: label={0}
 must be greater than {1} attempted={2}
nz.co.senanque.validationengine.range.minInclusive=Failed range: label={0}
 must be greater than or equal to {1} attempted={2}
nz.co.senanque.validationengine.conversion.failure=Cannot convert from {0}
 to {1}
```

Where you need alternate languages for these you just need to add your translated properties file to your application in the usual Java-supported manner.

```
MessageSourceAccessorFactory.getMessageSourceAccessor().getMessage(code);
```

will return the translated message. You can add message arguments as well, see the Javadocs for details.

8. Implementing new Validators

To implement a new validator you need to first write an annotation. It ought to look something like this:

```
...
@Retention(RetentionPolicy.RUNTIME)

public @interface NewValidation {
 String minLength();
 String maxLength();
    String message() default "org.my.annotations.newvalidation";
}
```

You can then refer to the new annotation in your XSD. Let us assume you want to implement your annotation in a package named org.my.annotations

First add a line to the XSD header like this:

```
xmlns:my="http://annox.dev.java.net/org.my.annotations"
```

To refer to your new annotation you add this to one of the fields in the XSD:

```
...
<annox:annotate>
 <my:Length maxInclusive="100"/>
</annox:annotate>
...
```

You are not done yet. You still have to write the validation code. Create a class like this:

```
...
public class NewValidation implements FieldValidator<NewValidation>
{
    private int m_minLength;
    private int m_maxLength;
    private String m_message;
    private PropertyMetadata m_propertyMetadata;
    /* (non-Javadoc)
     * @see
 nz.co.senanque.validationengine.annotations1.FieldValidator#init(java.lang.annotation
     */
    public void init(NewValidation annotation, PropertyMetadata
 propertyMetadata)
    {
        m_minLength =
 annotation.minLength()==null?-1:Integer.valueOf(annotation.minLength());
        m_maxLength =
 annotation.maxLength()==null?-1:Integer.valueOf(annotation.maxLength());
        m_propertyMetadata = propertyMetadata;
        m_message = annotation.message();
    }

    /* (non-Javadoc)
     * @see
 nz.co.senanque.validationengine.annotations1.FieldValidator#validate(java.lang.Object
     */
    public void validate(Object o, Locale locale)
```

```
    {
        if (o != null && o instanceof String) // ensure you handle the
 null case
        {
            int l = ((String)o).length();
            if (m_minLength != -1 && l < m_minLength)
            {
                String message =
 m_propertyMetadata.getMessageSource().getMessage(m_message,
 new Object[]{ m_propertyMetadata.getLabelName(locale),
 m_minLength,m_maxLength,String.valueOf(o) },locale);
                throw new ValidationException(message);
            }
            if (m_maxLength != -1 && l > m_maxLength)
            {
                String message =
 m_propertyMetadata.getMessageSource().getMessage(m_message, new Object[]
{ m_propertyMetadata.getLabelName(locale),m_minLength,m_maxLength,
 String.valueOf(o) },locale);
                throw new ValidationException(message);
            }
        }
    }

}
```

This is just a copy of the existing length validator and the logic is all fairly obvious.

Finally you need to add it to the list of validator classes wired into the AnnotationsMetadataFactory in your Spring beans definitions.

9. Extending the Choice Lists

9.1. Choice List Factories

If you need to fetch the choice list values from somewhere other than XML you can do that two ways. The first way is to have some process that builds the XML file and makes it available to your application. It might, for example, call a web service or a database to find the valid values for your field(s) and automatically edit them into your XML file.

The next time Madura Objects initialises it will load the new XML and see the new values.

But you can cut out this XML step entirely by wiring one or more classes that implement `nz.co.senanque.validationengine.choicelists.ChoiceListFactory`. This has one method that is passed the name of the choice list and it returns a list of `ChoiceBase` objects that describe each allowed value.

Your implementation code can do whatever it needs to to make that list.

To configure your ChoiceListFactory you must inject it into the `AnnotationsMetadataFactory` in a map, like this:

```
<bean id="metadata"
 class="nz.co.senanque.validationengine.metadata.AnnotationsMetadataFactory">
 ...
 <!-- The valid choices for fields can be defined in this document-->
 <property name="choicesDocument">
  <bean class="nz.co.senanque.madura.spring.XMLSpringFactoryBean">
   <property name="fileLocation" value="/choices.xml"/>
  </bean>
 </property>
 <property name="choiceListFactories">
  <map>
   <entry key="customerType">
    <bean class="myclasspath"/>
   </entry>
  </map>
 </property>
</bean>
```

The key is your choice list name so you can have a different (or the same) factory for each choice list you define.

Because this is completely optional you can still use the choices defined in the XML file for testing. All you need to do is *not* add the factory to the map and the XML choices will be used.

9.2. Choice Lists and I18n

Each item in a choice list has a key and a description. The key is the internal value, this is what your custom code should use and see, and this is the value you want to store on the database. For all intents and purposes this is the actual value selected.

But it may not be the value you want displayed. The key might be a word you want to translate to another language, which is why there is also the description. This value is the display value.

Each factory described in *9.1* has a `MessageSource` it can use to select the right resource to use to look up the base value found in the description field, effectively translating it to something appropriate for the current locale.

The default factory, the one you get if you don't specify any factories and just expect to use the description value as specified in the choices XML file will use the default message source. It will also fall back to supply the description value itself if you don't supply a translation for it. This means that if you do nothing at all about this then for a file like this:

```
<ChoiceList name="customerType">
```

```
 <Choice name="a">A</Choice>
 <Choice name="b">B</Choice>
 <Choice name="c">C</Choice>
 <Choice name="d">D</Choice>
 <Choice name="e">E</Choice>
 <Choice name="f">F</Choice>
</ChoiceList>
```

You will get descriptions A, B, C.... However if you add `A=door` to the TestMessages.properties file in the sample (that is the file the message source loads) then instead of A you will see 'door'. Then if you sent your language to French and create a file TestMessages_fr.properties with the appropriate translations including `A=port`, you will see 'port' instead of A.

This is fine as far as static choices go, but what if your choices are more dynamic? What if you are using a factory to read the choices externally?

What you do then is inject a `MessageSource` into your factory and use that when you call the constructor of each `ChoiceBase`. The `ChoiceBase` will call your `MessageSource` with the current Locale whenever it needs to translate a description. It is up to you to figure out what your `MessageSource` should do, of course.

If you do need to do this you might take a look at `org.springframework.context.support.JdbcMessageSource` which is included in the source. This is an example (by Olivier Jolly) of mapping a Spring message source to JDBC queries.

A. Licence

The code specific to MaduraObjects is licenced under the Apache Licence 2.0 [13].

The dependent products have the following licences:

jaxb	Dual license consisting of the CDDL v1.0 and GPL v2
hyperjaxb3	The BSD Style License
hibernate	GNU Lesser General Public License, Version 2.1
persistance-api	Common Development and Distribution License (CDDL) v1.0
maduraconfiguration	Apache Software License, Version 2.0
slf4j	SLF4J License, V1.0
Spring Framework	Apache Software License, Version 2.0
jdom	Apache Software License, Version 2.0

B. Release Notes

You need Java 1.7 to compile this project.

2.2.1

Migrated the build to maven. No functional changes but some documentation revision.

2.2

Added pom file for maven projects.

Fixed conflict with slf4j dependencies

Fixed conflict with slf4j dependencies

2.1

Now built with Java 1.7

2.0

Added the 'unknown' mechanism, allowing us to register a field as unknown even if we have a value for it. This is not actually used by validation but the Madura Rules plugin uses it. It is potentailly useful for other plugins.

Reworked the mechanism to set default values. It is now done from the constructor.

Various additions to make FieldMetadata more useful.

1.8

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

Change to allow you to set the class definitions differently in AnnotationsMetadataFactory. The previous approach, setting the package name, still works but not always when the classes reside in a Madura Bundle. In that case you can set the classes using the `classes` property. You do have to list each class, though, so use the package name when you can... while this works you actually don't need it because the Madura Bundle classes are now handled using the right classloader.

Made some classes Serializable.

1.7

Now allows forcing a value when assign() is used, which means rules can always force a value when they need to but setting a derived value still (correctly) fails.

Updated the licence information.

Added the 'initialValue' field to the proxyField.

Enhanced the plugin interface to pass information about changing elements in arrays. Also tidied up the way array changes are handled in the listening array and ensured that the clear() method works properly

1.6

Added unbindAll to validationEngine. This clears all the bound objects from the session. UnbindAll is a steamroller solution, a more selective unbind is preferred but there are problems with lazy collections and the generated equals() method that prevent the selective unbind from working

at times. Failure of the selective unbind is solved by the 'open session in view' pattern but the unbindAll is simpler to implement in some cases.

1.5

Simplified validation configuration.

Added @Ignore and @Secret annotations.

1.4

Added factory for messageSource. Previously relied on injection. Also now using Spring's LocaleContectHolder instead of trying to store the Locale for this thread.

1.3

Refactored some classes to use interfaces.

Added permissions annotation. Not used directly but UI and other app code can check it.

fixed an issue with locale not 'sticking' to the session correctly.

Fixed a problem relating to setting primitives (boolean, long, int etc) to null.

A class that extends another was generated with duplicate methods etc: fixed.

Tidied up the I18n issues so that Locale is not being passed around everywhere.

Added support for Default Value. The default is converted to appropriate datatype and the setter for the field is called. This happens at *bind* time, not when the object is instantiated.

If we are setting a value into a non-empty choice list field then we need to set it to null first and let any derived choices reset else our current choice may be rejected by the validation even though the underlying rules engine knows it is okay.

Improved shareability of choiceslist file, ie now rules plugins can see the document.

Resource bundle in choice lists.

Simpler ways to specify the business objects in Spring.

Build improvements

Add optional factory mechanism to choice lists

Support for 'isXXX' type getters.

1.1

Document that fields that are ignored if they have no annotation. Issue #3

Added getProxyFields() method to ObjectMetadata. Needed for MaduraRules to handle lists correctly. Issue #2

Adding attached obects that are not in a list was broken. The target object was not bound properly. Issue #1

1.0

Initial version