# BPMN 2.0 File Formats: With a Transformation to a Transformation

## Applies to:

Business Process Modeling, Business Process Management. For more information, visit the Business Process Modeling homepage

## Summary

BPMN 2.0 defines two official file formats for interchange: One XML format defined by the XML schema and one XMI format defined by the CMOF meta model. Consequently the standard also contains two transformations scripts that can convert files between the two formats. But how are these XSLT scripts created and maintained? How is ensured that they work correctly?

**Author:** Reiner Hille-Doering

**Company:** SAP

**Created on:** 25 October 2010

## Author Bio

Reiner works for SAP since 1997. He has developed several products for Microsoft and Eclipse technology integration and BPM. He is an architect for BPM products, member of OMG task forces and Eclipse committer.

## Table of Contents

## OMG Reality Check

BPMN 2.0 is specified by OMG, the organization that is also famous for other great standards like UML (Unified Modeling Language). OMG is focused on stuff that is related to modeling and model driven development. An interesting "meta" standard from OMG is MOF (Meta Object Facility): It allows defining a new modeling language (like BPMN) by Meta modeling. Defining a meta model in CMOF (Common Meta Object Facility) has several benefits:

- Documentation in form of class diagrams that shows entities with their attributes and relations.
- API for working on (BPMN-) model in memory, e.g. to create or delete entities, modify attributes, validate model and so forth.
- File format to exchange model content.

The last aspect is interesting: By defining a CMOF meta model, we also implicitly define a file format for interchanging instances of this meta model. The corresponding standard XMI (Extensible Model Interchange) defines rules on how to translate a meta model into an XML-based file format. Most rules are straightforward: Model packages are translated into XML namespaces, instances of a class become XML elements and attributes and relation are saved as XML attributes or elements. However the details are complicated and I must admit that I'm not always sure about all details. Anyway, theoretically one could derive an XML schema from any CMOF meta model.

Now, why is there anything else than XMI file formats derived from meta models? Because the whole XMI/MOF story has seldom been proven in reality. Most official standards define their exchange file format in an XML Schema (XSD), because XSD has proven to be well supported by any important technology and XSD is strict enough to avoid ambiguities.

Therefore BPMN task forces decided to go for both, a meta model and an XML Schema. In fact the task force members could be divided into two groups: Meta model affine and XSD affine people.

## Manual and automatic work

The decision had the practical implication that each modification in the standard needed to be manually applied to two places: The CMOF meta model using a UML modeling tool and the XSD using an XML schema editor. Obviously it would be desirable to support the work with tools:

1. A tool should check or enforce consistency between CMOF and XSD.
2. A tool should be shipped to customers that coverts file from XML to XSD and vice versa.

In the second article of my little series I have mentioned that my merge tool is good for 1. For 2, OMG task force decided to provide XSLT (Extensible Stylesheet Language Transformation) templates. XSLT is a well-established standard to transform between XML formats. Implementing two XSLT templates (from XML to XMI and from XMI to XML) thus seems to be a good idea.

The problem is however the creation and maintenance of the XSLT files. Ideally the XSLT files would have been updated together with the CMOF and XSD, but this was considered too much effort for each change.

With BPMN 2.0 Beta 1 (end of submission phase) the two XSLTs were authored by hand under high time pressure. Therefore the XSLTs were not fully functional and didn't cover all elements and attributes of the specification. But at least they were good enough for BPMN 2.0 Beta 1 and proofed that the approach works.

For the final BPMN 2.0 specification however, we needed a complete implementation in the XSLTs. Achieving this goal by hand seemed virtually impossible.

Therefore I tried to generate the XSLT files automatically. Required input is a formal mapping specification between CMOF meta model and XSD file format. And exactly this information is contained in the EMF Meta model implementation that I described in article 1 and article 2. The `ExtendedMetadata` annotations carry exactly the transformation information for each class and each member of the BPMN 2.0 meta model.

## Differences between XMI and XML

As mentioned before, XMI derives the XML format from the meta model. The top-level element is `<xmi>`, containing all “top-level” objects of the model. If there is exactly one top-level element, the `<xmi>` tag can be skipped.

The name for such an object tag is the name of the meta model class, qualified with a prefix that points to the namespace of the class. Thus an empty XMI file for a BPMN model could look like:

```
<?xml version="1.0" encoding="UTF8"?>
<xmi:XMI xmi:version="2.0"
      xmlns:xmi="http://www.omg.org/XMI"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:bpmn2="http://www.omg.org/spec/BPMN/20100524/MODEL-XMI">
  <bpmn2:Definitions id="id1">
  </bpmn2:Definitions>
</xmi:XMI>
```

This is not so different from the corresponding XML:

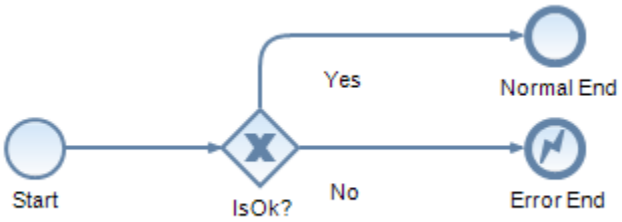
```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn2:definitions
      xmlns:bpmn2="http://www.omg.org/spec/BPMN/20100524/MODEL"
      id="id1">
</bpmn2:definitions>
```

If we skip the (optional) XMI element, the only differences are upper versus lowercase and the difference in the BPMN2 namespace, which either ends with “MODEL” or with “MODEL-XMI”.

In both formats everything else is directly or indirectly contained in the “definitions” element. The XMI rules state that for each class member either an XML element or an XML attribute is created. Simple members, e.g. of type “string” or “integer”, are rendered as XML attributes. Complex ones become XML elements. Thus all BPMN classes will be stored as XML elements. But the name of the element is not derived from the member’s type, but from the member name. The type is stored in the `xsi:type` attribute. In the example below you easily see the XMI characteristics: The process appears quite meaningless as `<rootElements>` tag, because the polymorph reference in the meta model has the name “rootElements”. And all start event, end event, and sequence flow appear as tag `<flowElements>`, again because this is the name of the polymorph reference in the meta model.

All elements carry the `xsi:type` attribute with the qualified name of the CMOF class. All non-contained members in the example are written as XML attributes, where references use simple ID matching.

The following picture shows a simple process that I use to illustrate the file formats and their transformation.



In XMI, the model looks like this:

```
<?xml version="1.0" encoding="UTF8"?>
<bpmn2:Definitions xmi:version="2.0"
     xmlns:xmi="http://www.omg.org/XMI"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:bpmn2="http://www.omg.org/spec/BPMN/20100524/MODEL-XMI"
     id="definitionsId" name="My BPMN Model">
 <rootElements xsi:type="bpmn2:Process" id="processId" name="Simple Process">
  <flowElements xsi:type="bpmn2:StartEvent" id="startId" name="Start"
     outgoing="startToGatewayId"/>
  <flowElements xsi:type="bpmn2:ExclusiveGateway" id="gatewayId" name="IsOk?"
     incoming="startToGatewayId"
     outgoing="gatewayToNormalEndId gatewayToErrorEndId"/>
  <flowElements xsi:type="bpmn2:EndEvent" id="normalEndId" name="Normal End"
     incoming="gatewayToNormalEndId"/>
  <flowElements xsi:type="bpmn2:EndEvent" id="errorEndId" name="Error"
     incoming="gatewayToErrorEndId">
   <eventDefinitions xsi:type="bpmn2:ErrorEventDefinition" id="errorTriggerId"/>
  </flowElements>
  <flowElements xsi:type="bpmn2:SequenceFlow" id="startToGatewayId" name=""
     sourceRef="startId" targetRef="gatewayId"/>
  <flowElements xsi:type="bpmn2:SequenceFlow" id="gatewayToNormalEndId" name="Yes"
     sourceRef="gatewayId" targetRef="normalEndId"/>
  <flowElements xsi:type="bpmn2:SequenceFlow" id="gatewayToErrorEndId" name="No"
     sourceRef="gatewayId" targetRef="errorEndId"/>
 </rootElements>
</bpmn2:Definitions>
```

The same model in the BPMN 2.0 XML format looks a bit nicer:

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn2:definitions
     xmlns:bpmn2="http://www.omg.org/spec/BPMN/20100524/MODEL"
     id="definitionsId" name="My BPMN Model">
 <bpmn2:process id="processId" name="Simple Process">
  <bpmn2:startEvent id="startId" name="Start">
   <bpmn2:outgoing>startToGatewayId</bpmn2:outgoing>
  </bpmn2:startEvent>
  <bpmn2:exclusiveGateway id="gatewayId" name="IsOk?">
   <bpmn2:incoming>startToGatewayId</bpmn2:incoming>
   <bpmn2:outgoing>gatewayToNormalEndId</bpmn2:outgoing>
   <bpmn2:outgoing>gatewayToErrorEndId</bpmn2:outgoing>
  </bpmn2:exclusiveGateway>
  <bpmn2:endEvent id="normalEndId" name="Normal End">
   <bpmn2:incoming>gatewayToNormalEndId</bpmn2:incoming>
  </bpmn2:endEvent>
  <bpmn2:endEvent id="errorEndId" name="Error">
   <bpmn2:incoming>gatewayToErrorEndId</bpmn2:incoming>
   <bpmn2:errorEventDefinition id="errorTriggerId"/>
  </bpmn2:endEvent>
  <bpmn2:sequenceFlow id="startToGatewayId" name="" sourceRef="startId"
    targetRef="gatewayId"/>
  <bpmn2:sequenceFlow id="gatewayToNormalEndId" name="Yes"
    sourceRef="gatewayId" targetRef="normalEndId"/>
  <bpmn2:sequenceFlow id="gatewayToErrorEndId" name="No"
    sourceRef="gatewayId" targetRef="errorEndId"/>
```

```
 </bpmn2:process>
</bpmn2:definitions>
```

The OMG team used XSD tricks like substitution groups to ensure that all objects appear with their type name. Another difference appears in the use of XML elements for the references “incoming” and “outgoing”. The BPMN 2.0 authors have chosen to use elements, as there could be multiple incoming or outgoing sequence flows. In the example the gateway has two outgoing sequence flows. In such case XMI creates a space-separated id list. Finally there is an important difference that is not obvious in the example: XMI does not care about order in the file: class members can be written in an arbitrary order. The XML format however defines a strict order of all sub elements.

## Planning the Transformation with a Sample

Let's now create an XSLT that translates the former example from XMI format to XML format. Later we generalize the sample transformation to support full BPMN and also transform in the opposite direction.

Let's start with the root template (`match="/"`) and the `DefinitionsTemplate`, that matches on `Definitions` elements, regardless if they appear on root or nested inside of an "`XMI`" tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:bpmn2="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:bpmn2xmi="http://www.omg.org/spec/BPMN/20100524/MODEL-XMI">

  <xsl:template match="/">
    <xsl:apply-templates />
  </xsl:template>

  <xsl:template name="DefinitionsTemplate" match="//bpmn2xmi:Definitions">
    <bpmn2:definitions>
      <xsl:if test="@name">
        <xsl:attribute name="name"><xsl:value-of
          select="@name" /> </xsl:attribute>
      </xsl:if>
      <xsl:if test="@id">
        <xsl:attribute name="id"><xsl:value-of
          select="@id" /> </xsl:attribute>
      </xsl:if>
      <xsl:for-each select="rootElements">
        <xsl:choose>
          <xsl:when test="substring-after(@xsi:type, ':') = 'Process'">
            <bpmn2:process>
              <xsl:call-template name="ProcessTemplate" />
            </bpmn2:process>
          </xsl:when>
          …
        </xsl:choose>
      </xsl:for-each>
    </bpmn2:definitions>
  </xsl:template>
```

The `DefinitionsTemplate` will create a `bpmn2:definitions` element, copy id and name attributes and then search for all `rootElements`. If it finds one, it will check if it is of type "`Process`". In this case, it creates a "`bpmn2:process`" element and calls the `ProcessTemplate`.

The `ProcessTemplate` looks similar – which is intended, as we want to generate the templates from the meta model classes.

```
<xsl:template name="ProcessTemplate">
  <xsl:call-template name="FlowElementTemplate" />
  <xsl:for-each select="flowElements">
   <xsl:choose>
     <xsl:when test="substring-after(@xsi:type, ':') = 'EndEvent'">
       <bpmn2:endEvent>
         <xsl:call-template name="EndEventTemplate" />
       </bpmn2:endEvent>
```

```
      </xsl:when>
      <xsl:when
        test="substring-after(@xsi:type, ':') = 'ExclusiveGateway'">
        <bpmn2:exclusiveGateway>
          <xsl:call-template name="ExclusiveGatewayTemplate" />
        </bpmn2:exclusiveGateway>
      </xsl:when>
      <xsl:when test="substring-after(@xsi:type, ':') = 'StartEvent'">
        <bpmn2:startEvent>
          <xsl:call-template name="StartEventTemplate" />
        </bpmn2:startEvent>
      </xsl:when>
    </xsl:choose>
  </xsl:for-each>
</xsl:template>
```

Again the template must copy the attributes first. But instead of repeating the code here, we call a “super class template”: We assume that we have one template for each meta model class. So we can call the templates of the super classes. This approach minimizes the redundancy in the template, and works very well for generated XSLTs.

After the super class template `FlowElementTemplate` is called, again we search for contained elements. In our sample they are all “`flowElements`”. Again we check the type of the flow element. Depending on the type, we create the appropriate BPMN element and call the fitting template to fill the new element with data.

The approach to loop over generic elements and then use long choose clauses to create the individual element might look odd to you. However I didn’t find a better approach. Note that the value of the XSI type attribute is equivalent to the CMOF class name. But it does not need to match to the BPMN2 element name. This class-to-element mapping must be done somehow. Also note the order inside of the `flowElements` group must stay as it is. The order between different elements however must be corrected according to the XSD.

Let’s now have a look at `EndEventTemplate`, `ExclusiveGatewayTemplate`, and so forth. They all mainly call the super class template `FlowNodeTemplate`.

```
<xsl:template name="FlowNodeTemplate">
  <xsl:call-template name="FlowElementTemplate" />
  <xsl:call-template name="SplitString">
    <xsl:with-param name="list" select="@incoming" />
    <xsl:with-param name="elementName" select="'bpmn2:incoming'" />
    <xsl:with-param name="elementNamespace"
      select="'http://www.omg.org/spec/BPMN/20100524/MODEL'" />
  </xsl:call-template>
  <xsl:call-template name="SplitString">
    <xsl:with-param name="list" select="@outgoing" />
    <xsl:with-param name="elementName" select="'bpmn2:outgoing'" />
    <xsl:with-param name="elementNamespace"
      select="'http://www.omg.org/spec/BPMN/20100524/MODEL'" />
  </xsl:call-template>
</xsl:template>

<xsl:template name="FlowElementTemplate">
  <xsl:if test="@id">
    <xsl:attribute name="id"> <xsl:value-of select="@id" /> </xsl:attribute>
  </xsl:if>
  <xsl:if test="@name">
    <xsl:attribute name="name"> <xsl:value-of select="@name" /> </xsl:attribute>
```

```
    </xsl:if>
  </xsl:template>

  <xsl:template name="SplitStringCore">
    <xsl:param name="list" />
    <xsl:param name="elementName" />
    <xsl:param name="elementNamespace" />
    <xsl:if test="string-length($list) > 1">
      <xsl:element name="{$elementName}" namespace="{$elementNamespace}">
        <xsl:value-of select="substring-before($list, ' ')" />
      </xsl:element>
      <xsl:call-template name="SplitStringCore">
        <xsl:with-param name="list" select="substring-after($list, ' ')" />
        <xsl:with-param name="elementName" select="$elementName" />
        <xsl:with-param name="elementNamespace" select="$elementNamespace" />
      </xsl:call-template>
    </xsl:if>
  </xsl:template>

  <xsl:template name="SplitString">
    <xsl:param name="list" />
    <xsl:param name="elementName" />
    <xsl:param name="elementNamespace" />
    <xsl:call-template name="SplitStringCore">
      <!-- Remove all unnecessary whitespace, but add a marker space at the
        end to simplify recursion. -->
      <xsl:with-param name="list"
        select="concat(normalize-space($list), ' ')" />
      <xsl:with-param name="elementName" select="$elementName" />
      <xsl:with-param name="elementNamespace" select="$elementNamespace" />
    </xsl:call-template>
  </xsl:template>
```

`FlowNodeTemplate` again calls `FlowElementTemplate` to copy `id` and `name` attributes. Then it creates the element `incoming` and `outgoing`. Remember that these have to be filled from a space-separated list in XMI. The recursive helper template `SplitString` does the job. It takes a space-separated string-list as input, together with a name and namespace of an element that should surround each id in the result XML.

## A Transformation for a Transformation

From the sample we can now derive a solution for whole BPMN. The basic idea is straightforward: For each CMOF class we create an XSLT template with the name of the class, appended by “Template”. First the template calls all super class templates (CMOF supports multi-inheritance). This is simple, as the EMF meta model contains the necessary information.

Then the template creates all XML attributes according to the target format. Then all elements are created, following the patterns from the sample before. E.g. contained objects are handled with the for-each-choose pattern. Simple elements are copied in straight-forward manner and list references use the `SplitString` trick.

The creation of a XSLT file is in fact a model-to-text transformation from an annotated EMF meta model to the XSLT text file.

EMF provides several technologies for model-to-text transformation (M2T), e.g. JET, JET2, Acceleo, and Xpand. After some experiments I decided to go for JET (Java Emitter Templates), as it is simple to use and sufficiently powerful for my job.

As with most model-to-text systems, the heart of the transformation is a template. The result of the transformation is again a (XSLT) template, so don’t confuse template and meta template. But as a modeling expert, you are used to work with Meta levels, right?

I used an example like the XSLT from above and used it as meta template and then generalized it with some code. In JET we use Java code for controlling that template expansion and to retrieve values that are merged with the template. If for example some XSLT snippet needs to be repeated for each class from the BPMN 2.0 meta model, we need to encapsulate the snipped into a Java “`for`” loop. Such Java code is surrounded by “`<% %>`” signs. Of course each iteration of the snipped would look a little different. This differentiation is achieved with Java expressions, surrounded by “`<%= %>`”.

Templates can easily get messed up with too much Java code. Thus I create the helper Java class “`JetInput`”, which offers compact functions that can be called easily from the template.

Here is a snippet from the XMI-to-XML meta template. You clearly see the mixture of target XSLT code and JAVA snippets that control the generation process.

```
<%@ jet package="org.eclipse.bpmn2.tools.xsltFromEcore.jet" class="XMIToXML"
imports="org.eclipse.emf.ecore.*" skeleton="generator.skeleton"%>
<% JetInput input = (JetInput) argument; %>
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0"/>
<% for(EClass c: input.getClasses()) { %>
  <xsl:template name="<%=c.getName()%>Template"/>

<%for(EClass superClass: c.getESuperTypes()) {%>
     <xsl:call-template name="<%=superClass.getName()%>Template"/>
<%}
 for(EStructuralFeature feature: input.getAllFeatureThatAreAttributesInXml(c)) { %>
     <xsl:if test="<%=input.getXPathForXMI(feature)%>"><xsl:attribute name="<%=
input.getExtendedMetadata().getName(feature) %>"> <xsl:value-of
select="<%=input.getXPathForXMI(feature)%>" /> </xsl:attribute></xsl:if>
<% }
  for(EStructuralFeature feature: input.getAllElementsInXml(c)) {
  if(input.isReference(feature)) { %>
     <xsl:call-template name="SplitString">
       <xsl:with-param name="list" select="<%=input.getXPathForXMI(feature)%>" />
       <xsl:with-param name="elementName" select="'<%=input.getPrefix(feature)%>:<%=
input.getExtendedMetadata().getName(feature)%>'" />
       <xsl:with-param name="elementNamespace"
select="'<%=input.getNamespace(feature)%>'" />
```

```
        </xsl:call-template>
    <% }
```

The snipped shows how Java code and XSLT target content is mixed. The "`for(EClass c: input.getClasses())` loops over the BPMN meta model and create the contained XSLT template for each EClass. The expression "`<%=c.getName()%>Template"` gives each template a unique name. The other loops find all super classes and all members ("features") of the current class. For cases where such a feature should be written as an XML attribute, you see the mapping from meta model to schema using the `ExtendedMetadata` annotations. This ensures that the generated XML exactly fits to the XML schema:

```
<xsl:attribute name="<%= input.getExtendedMetadata().getName(feature) %>">
```

The full source code of the XSLT generator is available on GIT. Please see my first article for instruction on how to use GIT to download projects into Eclipse. You might be especially interested in the meta templates for XMI-to-XML and for XML-to-XMI. Also the final generated XSLTs are submitted: XMI2XML.xslt and XML2XMI.xslt. Note that these versions are newer than the ones which come with the official OMG BPMN 2.0 Beta 2 package and some bugs have been fixed in the meantime.

## Testing the XSLTs

Once the XSLTs are generated, they must be tested if they do what they are supposed to do: correctly convert from XMI to XML or vice versa. This also includes that ideally no data is lost and that XML schema and XMI production rules are considered.

Unfortunately this is more complicated than you might expect: Testing requires good test data and validation algorithms. Both lacks on XMI side: There are a couple of BPMN samples available for OMG, but none is available in XMI file format. I'm also not aware of any BPMN 2.0 modeling tool that is able to write XMI files. The only tool that for sure can create BPMN 2.0 XMI files is the MDT BPMN2 Meta model itself. But using this as input is kind of dangerous: The same meta model is used as input to create the XSLT files. If there is a bug in the EMF meta model, it most likely affects the XMI files written directly from EMF and the XMI files created by the XSLT transformation in the same way.

Anyway it makes sense to test the following use cases:

1. Check if all OMG samples can be converted with XML2XMI.xslt to XMI format and look if the result looks ok.
2. Convert the XMI file from step 1 back to XML using XMI2XML.xslt.
3. Check that the file from step 2 is semantically identical to the original BPMN XML file.
    a. Check if the file from step 2 complies to the XML schema
    b. Check if the content is roughly identical to the original file. There will be of course differences, e.g. in namespace prefixes, attribute order and so forth, but the XML infoset should be (almost) identical.
4. Use the BPMN2 EMF Meta model to convert the samples from XML to XMI.
5. Compare the files from step 4 with the files from step 1.
6. Convert the files from step 4 back to XML using XMI2XML.xslt and proceed according to step 3.

I did some of the checks and fixed several issues I found, however it can always happen that new test combinations appear and highlight new issues.

## Limitations and Outlook

The generated XSLT stylesheets work quite nicely, however they have some limitations:

- BPMN 2.0 allows extensions. Unfortunately the extensibility model for XML and XMI is completely different. The XSLTs can't convert the extensions. I tried it, but it seems to be not possible without knowing the extension. The XSLTs have an "`EObjectTemplate`" that could be used as starting point to put conversion logic for your extensions into it. Of course you can also modify the meta templates and regenerate the XSLTs to support your extensions.
- XMI has a lot of ambiguities. E.g. references could appear as XML attributes, containing a space-separated id list. But under some circumstances XMI would use elements with some special `XLink` attributes. The XSLTs cannot consider all possibilities and assume always the easiest variant. Thus the transformation could loose information or create not 100% correct results.
- The performance might not be perfect, as the XSLT contain quite a lot of filters and conditions. Anyway converting all 25 OMG BPMN samples forth and back and doing an XSD validation for all files only took about 1.5 seconds on my laptop, which seems to be no so bad.

The good thing at the generative approach is evident in such cases: A fix or extension in the meta template and regeneration of the XSLTs is quickly done. Imagine how much higher the effort was with manually written XSLTs. And it should be not too complicated to use the approach for other meta models that have both, an XMI and an XML representation. My meta template contain only little BPMN 2.0 specifics.

Anyway using the XSLTs might often be not the best option. If you just need to load XML or XMI file into memory or write them back, simply use the BPMN2 Ecore meta model directly. And if you need to covert between XML and XMI inside Eclipse, also the Ecore meta model is the best solution. The XSLTs are handy, if only an XSLT processor is available.

Overall it was impressive to me what can be achieved with standard and open-source technologies like XSLT and EMF. You just need to spend some time and patience and keep the attitude: there is always a solution for a problem.

If you have questions, feedback, ideas, or even want to contribute something to the MDT/BPMN2 project, contact me, send a mail to the MDTBPMN2 mailing list, join the MDT news group, or create a bugzilla ticket with product set to "MDT" and component to "BPMN2".

## Related content

BPMN 2.0 Meta model Implementation for Eclipse: Get it and Use it

Making of the BPMN 2.0 Meta Model for Eclipse: Merge and Conquer

For more information, visit the Business Process Modeling homepage

## Copyright