XML Binding from the DDI Model

Version 3.3 – 2016-04-15

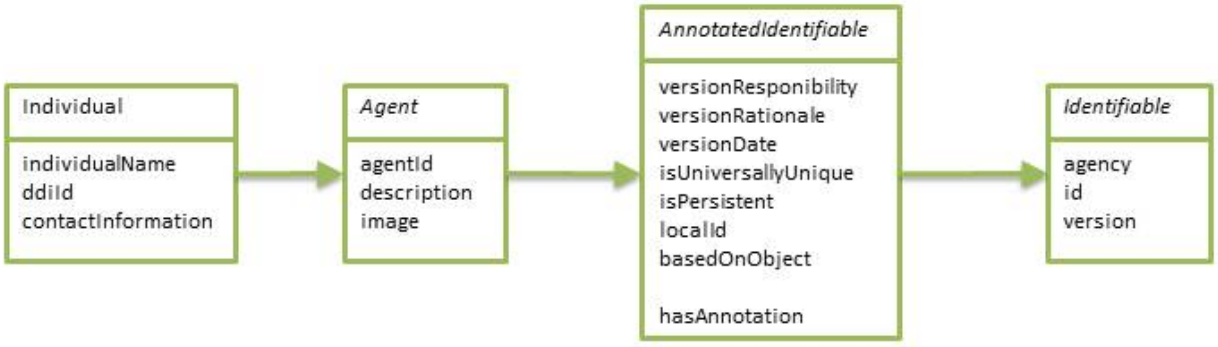
Introduction

This document presents the mapping rules from the XML export from Drupal to the XML schema deliverables. This process is split into two steps. First an intermediate reduction is produced as a second XMI file that focuses on what is the best implementation of the platform independent model in XSD. Second the actual XML schemas are created.

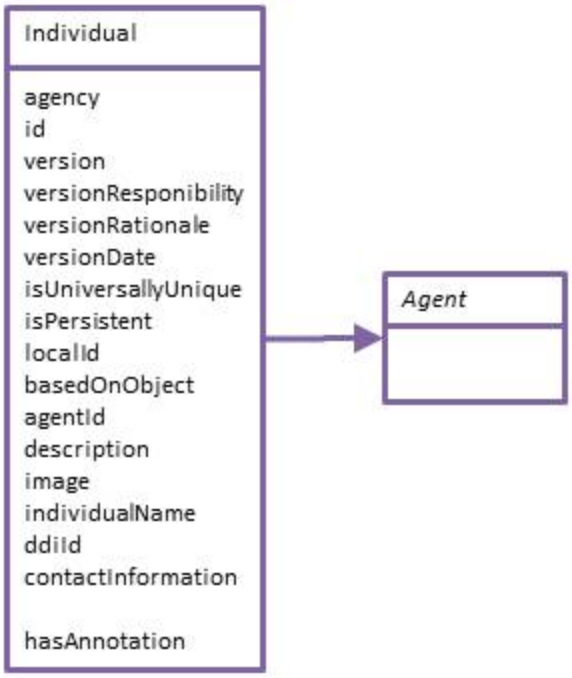
Rules for the Platform Specific Model – PSM

For the reason of getting a simple XML schema definition, with the least possible amount of different XSD expressions without losing validation capability, some reductions are made to the exported model from Drupal. This includes at first that all inheritance relationship between classes will be erased, so that all classes become completely independent entities. Therefore, abstract classes will not be transferred into the PSM for schema, except, when there is an association pointing to this abstract class. They are used to fill the list of possible classes to be target of the references resulting out of the corresponding association. For the same reason, the relationship to the parent class is kept inside the PSM.

As an example, the class Individual, derived from the abstract classes Agent, AnnotatedIdentifiable and Identifieable



will result in the following representation in the PSM.



The XMI export is therefore transformed as follows:

(1) The overall structure with its steering data for XMI is copied to the new XMI file. This includes also the two model layers for library packages and views. The library model is created, all its properties and documentation are copied. The views model is copied entirely as it is.

(2) All library packages will be created, all their attributes and documentation are copied.

(3) All <packagedElements xmi:type=”uml:Enumeration”> and <packagedElements xmi:type=”uml:DataType”> are copied. They usually just appear within the package “ComplexDataTypes”.

(4) All <packagedElements xmi:type=”uml:Class” isAbstract=”true”> are created to ensure the information about the inheritance line. All their attributes and documentation and the generalization child node are copied, no other children will be transferred here.

(5) All <packagedElements xmi:type=”uml:Class” isAbstract!=”true”> are created, all their attributes and documentation are copied.

    1. All children <ownedAttribute> from all superclasses (referenced by <generalization general=”xyz”/>) are transferred fi there was no other property on the inheritance path so far with the same name (overriding properties). This means, that properties and associations from superclasses will appear before the directly owned ones. The ownedAttributes will also be transferred, if they come from abstract classes. Properties will be copied. Associations are created, all attributes except “xmi:id” and “association” are copied. “xmi:id” and “association” are created with treir content changed to replace the superclass name with the processed class name. The child <type> is copied,

the children <lowerValue> and <upperValue> are created and their attributes are copied, except “xmi:id”, which is handled the same way as for the parent node.
The only exception of this will be <ownedAttribute name=”realizes”>.

2. All children <ownedAttribute name=”realizes”> are suppressed as they are indicators for the realization of a pattern.

(6) After each <packagedElements xmi:type=”uml:Class” isAbstract!=”true”> handled, all needed <packagedElements xmi:type=”uml:Association”> are created:

1. All associations from superclasses are created. All attributes except “xmi:id” are copied. “xmi:id” is created with its content changed to replace the superclass name with the processed class name. All children are created and treated to replace the superclass name also. This will also include associations from abstract classes.

2. All associations directly related to the processed class are copied.

(7) The <xmi:Extension>, containing the documentation content on property level, is created, all its attributes are copied.

(8) The documentation content for all packages and views is copied.

(9) With an iteration over all classes, processed within the packages, the field level documentation will be created for each of these classes:

1. <element> is created, all according attributes are copied. All according children are copied.

2. All <attributes><attribute> children of extension nodes according to the processed classes superclasses are transferred including the renaming.

General structure of XML schema files for DDI

The general principle for creating XML schema files from the DDI model is to have a general relation between the type of an entry and its position in the resulting DOM tree. This results in a very flat structure in XML instances together with universally valid paths to each entry. Practically this means that there should be no path that is deeper than three steps from the root element. The resulting four levels of the DOM will be:

(1) The root element is always named “DDI” and can carry an attribute to identify the view it relates to.

(2) This level will be the only one to contain identifiable objects and nothing else. The order of appearance is not relevant for the order of use.

(3) All properties of identifiable objects will be contained inline. There could be two different ways to achieve this:

a. Simple or complex data types are used for direct definition of possible child elements. Some will be handled as child nodes, some as attributes.

b. Associations to other identifiable objects are handled as references. All reference types are derived from a general reference type, which is just completed by an attribute that could contain the type of the referenced object, implemented as a controlled list. The general reference type already defines the referencing mechanism.

(4) Complex data types might have a sub-structure to carry properties. In some cases this will include references to identifiable objects, which are implemented in the same way as on level (3).

```
<DDI>
        ...
        <Individual>
                <Id>some URN</Id>
                <IndividualName>John Doe</IndividualName>
                ...
        </Individual>
        ...
</DDI>
```

There is always one library schema file, containing the entire set of all classes from the model. Besides that, each view defined in the model will also be created as an independent schema file to contain all class definitions used by that view. References to objects outside the view should be marked as isExternal=”true”.

Every schema file will contain a definition (complexType and element) of a root element that is always named DDI. This element will contain:

(1) An attribute to identify its view named “type” that is set to a fixed value for a view schema. This value will be the name of the view.

(2) A sequence of child elements containing one appearance of DocumentInformation and then an unbound choice of instances of all the classes that are contained the according view or the library.

```
<xs:complexType name="DDIType">
   <xs:sequence minOccurs="1" maxOccurs="1">
      <xs:element maxOccurs="1" minOccurs="1" ref="DocumentInformation"/>
      <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:element maxOccurs="unbounded" minOccurs="0" ref="Address"/>
        <xs:element maxOccurs="unbounded" minOccurs="0" ref="ContactInformation"/>
        ...
      </xs:choice>
   </xs:sequence>
   <xs:attribute name="type" use="required">
      <xs:simpleType>
         <xs:restriction base="xs:NMTOKEN">
            <xs:enumeration value="AgentsView"/>
         </xs:restriction>
```

```
      </xs:simpleType>
   </xs:attribute>
</xs:complexType>
```

High-Level XML Constructs – XML schema files

All DDI classes will run under the same namespace. This namespace will also be used for every view schema that is created. The Output of the transformation of XMI into XML schema will therefore be one file to contain the entire library of classes in the published model and one file for each view to contain all classes needed in this view.

(1) Packages are identified by <packagedElement xmi:type="uml:Package"> that is a child node under <packagedElement xmi:id="ddi4_model">. They are just used to distinguish between the complex data type and primitives on one side and the content driven classes on the other side.

(2) In every schema file containing a view, all classes are inserted, that are needed for this view or library. This includes everything from the packages "ComplexDataTypes" and "Primitives" as well as all classes being referenced in the view directly or all classes in case of the library. The only class from outside "ComplexDataTypes" and "Primitives" that is automatically inserted in every view schema file is "DocumentInformation".

(3) The Classes from the package with the name "ComplexDataTypes" are handled slightly different. Details within the section "Class-Level Mappings". Also, while processing this package, a complexType is defined, named ReferenceType, to serve as the prototype of all further references from one class to another.

Class-Level Mappings

(1) **Classes:** <packagedElement xmi:type="uml:class"> with **no** attribute isAbstract="true" corresponds to the generation of several things:

   a. A global element with the same name as the XMI name attribute, and of the complex type declared in the next step.

   b. A complex type using a concatenation of the name attribute and the string "Type". Within its complexContent this will contain an unbounded choice with an element definition for each <ownedAttribute xmi:type="uml:Property"> following the next step. No extention base is set.

   c. Both the element and the complexType will also contain an xs:annotation/xs:documentation with the content of xmi:Extension/elements/element this the name of the class as content of the xmi:idref attribute. The class level documentation is there contained in properties/@documentation

(2) **Associations to other DDI Classes:** If a child <ownedAttribute xmi:type="uml:Property"> has an aggregation or association attribute, then resolve the ID in the association attribute, and take the value of the name attribute from the resolved XMI element. The name will now be the name of a new element. The type of this element is defined inline, extending ReferenceType by an attribute name typoOfClass, that enumerates all possible target classes. The list of target classes is created by iterating over the generalization references starting with the reference from the ownedAttributes type/@xmi:idref. The DDI referencing elements will always take their minOccurs and maxOccurs values from the value attribute of the <lowerValue> and <upperValue> XML elements (a value of "-1" means the maxOccurs has a value "unbounded".

Example:

```
<xs:element name="AgentAssociation" minOccurs="0" maxOccurs="1">
   <xs:complexType>
      <xs:complexContent>
         <xs:extension base="ReferenceType">
            <xs:attribute name="typeOfClass" use="required">
               <xs:simpleType>
                  <xs:restriction base="xs:NMTOKEN">
                     <xs:enumeration value="Individual"/>
                     <xs:enumeration value="Machine"/>
                     <xs:enumeration value="Organization"/>
                  </xs:restriction>
               </xs:simpleType>
            </xs:attribute>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
</xs:element>
```

(3) **Properties:** If a child <ownedAttribute xmi:type="uml:Property"> has no aggregation and association attribute the type for the generated element has to be determined whether it is part of extended primitive definitions in DDI Core or Utility by the name attribute or one of the primitives already defined in XML schema, using the <type xmi:type="???"/> child of ownedAttribute. Cardinalities are handled the same way as for associations. The only exceptions from this are properties to be identified as language definitions (<type xmi:type="xs:language"/>). Those properties will become attributes to the defining complex type of ref xml:lang or of the type xs:language and then keeping their name.

(4) **Documentation:** Both properties and references will contain field level documentation read from xmi:Extension/elements/element/attributes/attribute with the name equal to the ownedAttribute name

(5) **Classes in the package "ComplexDataTypes":** All properties (except a property named "content") will become attributes to a complexType with simpleContent of the same type as the property "content", if this class fulfills the following characteristics:

   a. It and its super classes include no associations.

b. It or its superclasses have a property named “content”, that is not of type “anyURI” or “xhtml:BlkNoForm.mix”.

(6) **Classes from the package “Primitives”** will not appear as classes as they are part of the XMI export to handle linkage of properties to primitive data types like string, number formats or dates.

(7) **Enumerations:** <packagedElement[@xmi:type=”uml:Enumeration”]> will always be handled within the package “ComplexDataTypes”. They will result in “simpleType” definitions based on “xs:NMTOKEN” being restricted to an enumeration of allowed entries.
Example:

```
<xs:simpleType name="ShapeCodedType">
   <xs:restriction base="xs:NMTOKEN">
      <xs:enumeration value="Rectangle"/>
      <xs:enumeration value="Circle"/>
      <xs:enumeration value="Polygon"/>
   </xs:restriction>
</xs:simpleType>
```

Naming conventions and name transformations

- In Drupal, all class name are written in upper camel case. All properties and associations are written in lower camel case.
- Within the transformation to XSD, all names of properties and associations are transformed to upper camel case, to fulfill the conventions of XML. In some cases, underscores and colons are erased, if they appear.

Versioning the model and schema

To make a lightweight versioning possible that is consistent from Drupal to the schema files, there is an enumeration in Drupal that only contains one value which is the DDI 4 version number. This enumeration is just used within the DocumentInformation class inside the model as data type for the property ofType. Therefore this version number will always be fixed for each instance that is bound to one of the view schema files or the library schema file for any given version.

Besides that, the version number is read out explicitly within the transformation for the schema file creation. This makes it possible to use the version number for a version attribute in each schema file’s root node and in all of the schema file names as an postfix.