XML Binding from the DDI Model

1. Version 3.1 – 2015-11-27

# 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 XSD expressions without loosing 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 will become an empty element and type definition as entry point for a substitution group later in the schema.

The XMI export is therefore reduces 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. 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.
	2. All children <ownedAttribute> are copied.
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.

# 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. This approach also includes, that the package structure of the model is transparent in the XML binding.

1. Packages are identified by <packagedElement xmi:type=”uml:Package”> that is a child node under <packagedElement xmi:id=”ddi4\_model”>. The name attribute is used to get the value of the xmlns attribute from the XML file mentioned above.
2. In every library XML schema file we import all other library schema files, for mapping to namespace identifiers with xsi:schemaLocation. The namespace identifiers will exist in the model.
3. The Classes from the package with the name “ComplexDataTypes” are handled slightly different. Details within the section “Class-Level Mappings”.
4. For every view, that is part of the Drupal export as a package with references to the contained classes, again an XML schema file is created. A view is identified by <packagedElement xmi:type=”uml:Package”> that is a child node under <packagedElement xmi:id=”ddi4\_model”>. The name attribute is used to get the value of the xmlns attribute from the XML file mentioned above. The XML schema file contains one complex type definition for a root element named after the view, which can contain all the classes referenced. Therefor, all library schema definitions are imported. A distinction, whether an import is needed or not, is not made. A corresponding element definition is also made.

# Class-Level Mappings

1. **Classes:** <packagedElement xmi:type=”uml:class”> corresponds to the generation of several things:
	1. A global element with the same name as the XMI name attribute, and of the complex type declared in the next step. If there is an attribute isAbstract with value true the global element will be defined as abstract.
	2. A complex type using a concatenation of the name attribute and the string “Type”. Within its complexContent this will contain at first an extension declaration with the name of the superclass, derived from <generalization>, via the xmi:id attribute, as base and then a sequence with an element definition for each <ownedAttribute xmi:type=”uml:Property”> following the next step. If no <generalization>is available, the base will be ddic:DDIObjectType
	3. If the class defines one or more properties and/or associations of the same name, as one of its superclasses, a complex type is generated, for which the name is a concatenation of the name attribute and the string “RestrictionType”. This type definition will contain all properties and associations from all superclasses, except those overridden in the same manner, as mentioned above but within a restriction to the superclass and not in extension. In this case, the extension base of the step above will be this restriction type and not the superclass.
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. Concatenate the value of the name with the string “Reference”, and place an element declaration of type “Reference” there. 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”.
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 the type xml:lang.
4. **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:
	1. It and its super classes include no associations.
	2. It or its superclasses have a property named “content”, that is not of type “anyURI” or “xhtml:BlkNoForm.mix”.

# 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 name 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.