The DDI 4 Model:

Binding and Intermediate Models

A Gregory, M Hebing, O Hopt, J Iverson, S Lloyd, F Rizzolo, A Rocha,
D Smith, W Thomas, J Wackerow
Minneapolis DDI Sprint, May 2015

Abstract

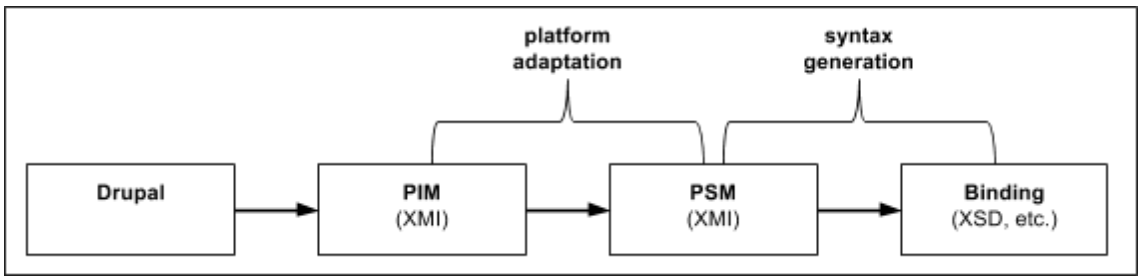
The Production Framework team has identified the possibility of separating the process of binding the model into two stages: (1) In the platform adaptation step, the Platform Independent Model (PIM) gets converted into a Platform Specific Model (PSM). The term "platform" thereby refers to a potential binding like XML, OWL, or SQL. (2) In the syntax generation step, the PSM gets converted into the actual binding. This paper takes a closer look at this workflow and, in particular, implications regarding versioning.

Table of Contents

XML binding details

Workflow

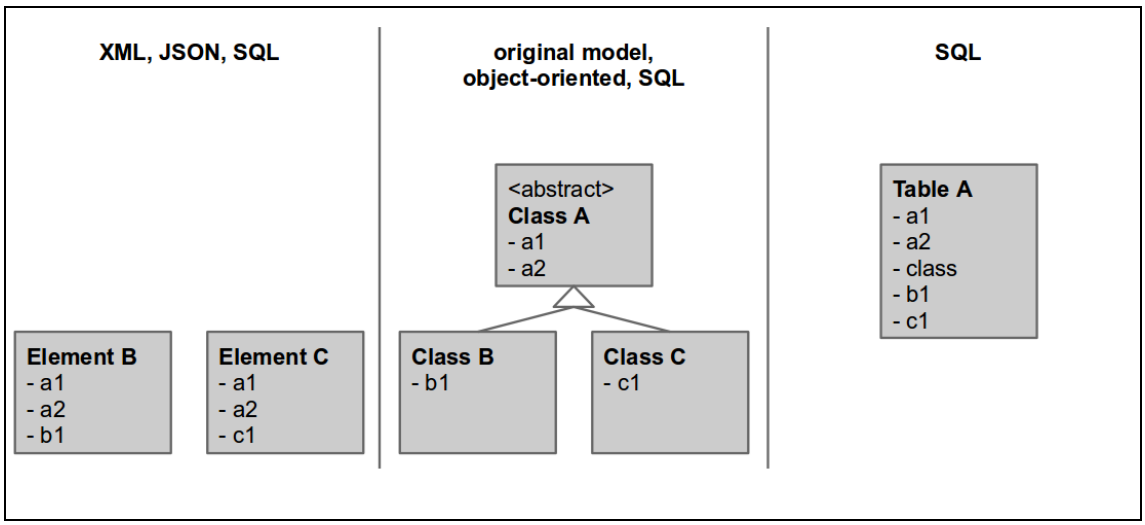
The original idea was to go directly from the model in Drupal to the various bindings. We identified a need to structure this workflow in more detail.



Workflow from Drupal to the final binding.

First, we separate the representation of the Platform Independent Model (PIM), which is represented in XMI, from Drupal as the current tool to edit the model. Based on a set of rules, configuration file(s), and a transformation program (probably XSLT), the PIM will be transformed into a Platform Specific Model (PSM). In the platform adaptation step, the PSM is a reduced version of the PIM which takes into consideration specific needs of a particular binding. In the syntax generation step, the actual bindings are produced based on the PSM. Again, the syntax generation step uses a set of rules, configuration file(s), and a transformation program (probably XSLT).

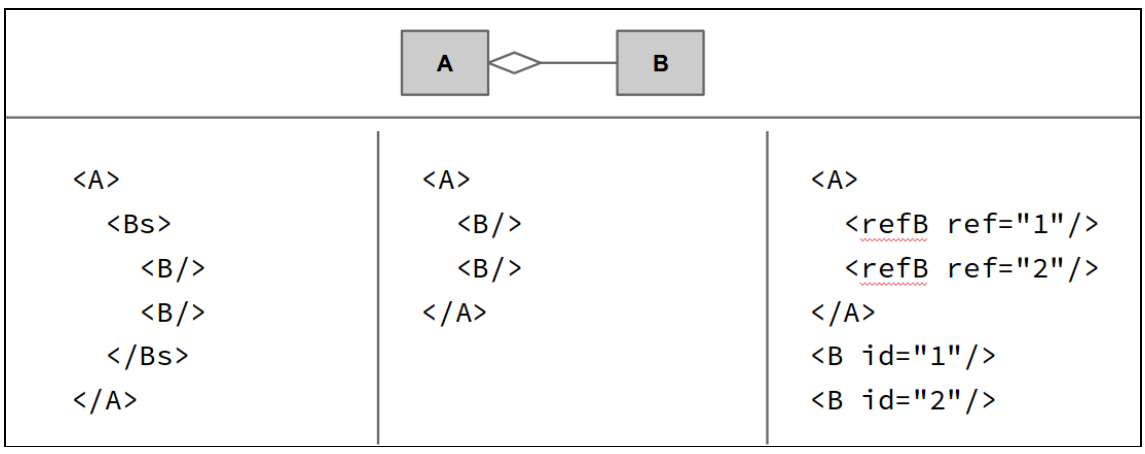
1. Platform Adaptation: PIM → PSM



The figure illustrates various Platform Specific Models (PSM) of one Platform Independent Model (PIM) in the middle.

In the first step, we transform the Platform Independent Model (PIM) to a Platform Specific Model (PSM), see the figure. We agreed on using XMI for both the PIM and the PSM—representing the PSM as an XMI is important to document the specific characteristics of a binding. The transformation from the PIM to the PSM is automated and based on a set of rules, configuration files, and a transformation program. The transformation program is expected to be written in XSLT, the format of the rules and the configuration has not yet been defined. The platform adaptation step results in a loss of information. The rules and configuration files have to cover both generic rules for a specific platform and even more specific rules for individual objects or views.

2. Syntax Generation: PSM → Syntax



Three examples, how a simple aggregation (represented in UML) can be represented in XML.

The syntax generation step generates the syntax binding from the PSM. The transformation is expected to be realized using XSLT and is again based on rules and configuration files.
The documentation of rules for the binding to XML schema on both steps is to be found here:
https://github.com/ddialliance/xmi-transformation/blob/master/xsd/doc
The same documentation for OWL can be found here:
https://github.com/ddialliance/xmi-transformation/tree/master/owl/doc
For each binding type, a specific document will be created. These documents should be structured in a consistent way. There should be sections on:

- the used subset of the binding syntax and the reasoning of the definition of this
- the mapping rules on a detailed level and the reasoning for each rule
    - the mapping rules should be in a table which can be the basis for creation of the formal configuration files containing the rules
- the rules for exceptions from the generic rules for special cases
- related work on which the mapping is based on

Conclusion and open questions

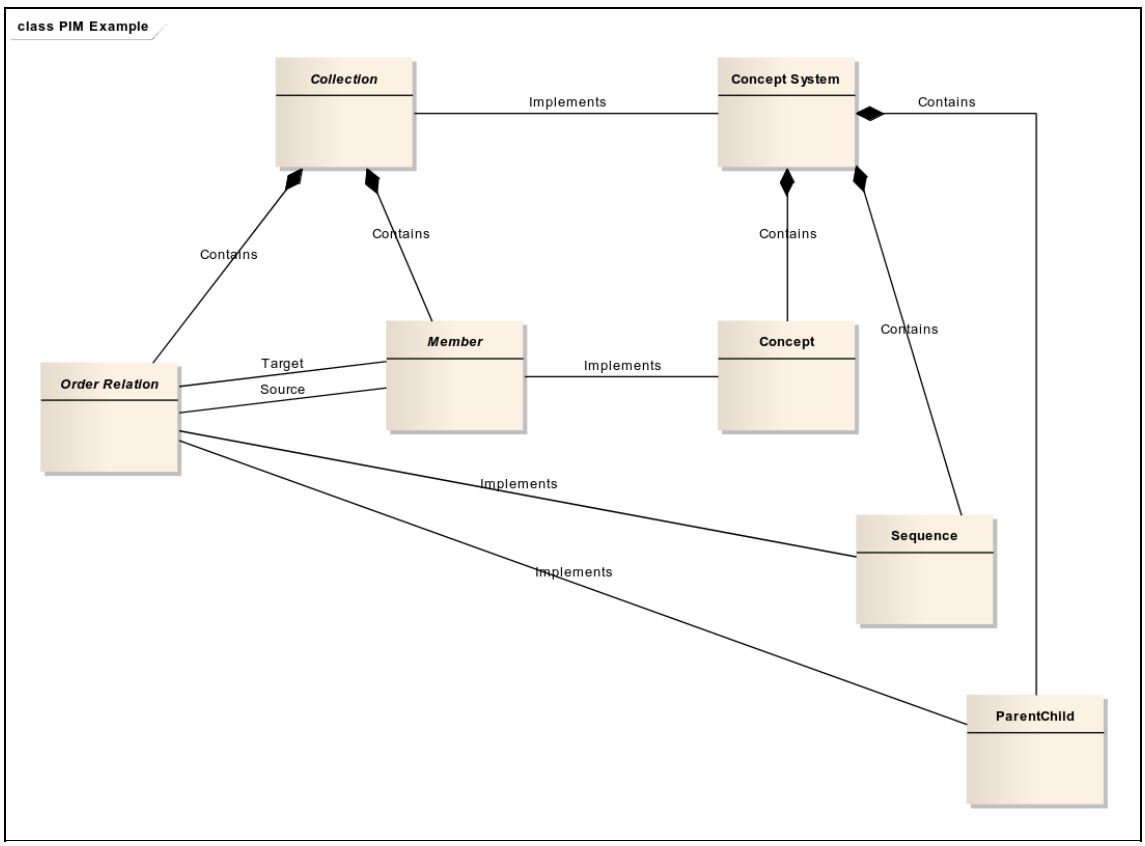
- We decided to separate the transformations.
    - Step 1: platform adaptation
    - Step 2: syntax generation
- We need a comparable documentation for all bindings (for developers). This can be realized through the XMI representation of the PSM.
- How and in which format should we define rules and configuration files for both the platform adaptation and the syntax generation.
- Documentation of both steps: human readable! machine actionable?
- How to handle element-specific rules for both platform adaptation and syntax generation?
- Both bindings and their documentations have to be adopted to the two step approach with the PIM/PSM. For XSD, this is already in progress.
- The PIM is conceptual and notions such as interfaces and implementation of behaviour are closer to the physical (language specific). Do those notions really belong in the PIM now that we have the PSM.
- What's exactly the separation of concerns between the PIM and the PSM? Consider the example below. The reason for saying that Concept "implements" Member instead of "isA" Member is that some platforms cannot deal with multiple inheritance. Shouldn't we model

that association in PIM as “isA” and then translate it into “implements” in the PSM that requires it?

Extended example

Platform Independent Model (PIM)

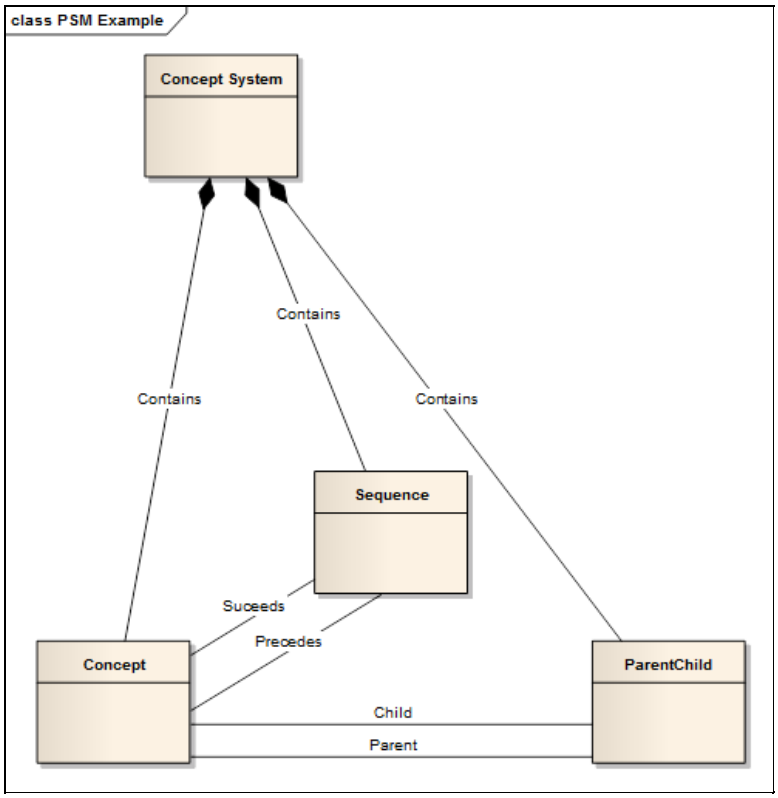
The diagram below shows how an “interface” (the Collections pattern) would be expressed in the PIM in the context of a Concept System.



Note that it would be possible to have the OrderRelations be turned into ComplexDataTypes if this was desirable (it could produce a more terse XML).

Platform Specific Model (PSM)

The same model would be collapsed to hide much of the chain of inheritance, by removing all abstract classes from the XML schema. In such a case, the PSM would appear as below.



From this, using current binding rules, the XML format could be derived.

Syntax binding: XML

(1) Simple list of concepts. Consider a list of three concepts: age, gender, and language.

XML Option 1: Verbose ordering. The first example shows what the XML binding could like if it directly mirrors the platform specific model.

```
<ConceptSystem>
    <ConceptReference ref="ageConcept" />
    <ConceptReference ref="genderConcept" />
    <ConceptReference ref="languageConcept" />

    <Sequence>
        <Precedes ref="ageConcept" />
        <Succeeds ref="genderConcept" />
    </Sequence>

    <Sequence>
```

```
        <Precedes ref="genderConcept" />
        <Succeeds ref="languageConcept" />
    </Sequence>
</ConceptSystem>

<Concept id="ageConcept" />
<Concept id="genderConcept" />
<Concept id="languageConcept" />
```

XML Option 2: Simple ordering. In the XML binding, we could use XML's ordering to represent the order of items in the collection.

```
<ConceptSystem>
    <ConceptReference ref="ageConcept" />
    <ConceptReference ref="genderConcept" />
    <ConceptReference ref="languageConcept" />
</ConceptSystem>

<Concept id="ageConcept" />
<Concept id="genderConcept" />
<Concept id="languageConcept" />
```

(2) Hierarchical Concepts. Consider three concepts: color, red, and blue. Red and blue are subtype or children of color.

XML. For a concept hierarchy, we explicitly state the relationship of one concept to another.

```
<ConceptSystem>
    <ConceptReference ref="colorConcept" />
    <ConceptReference ref="redConcept" />
    <ConceptReference ref="blueConcept" />

    <ParentChild>
        <Parent ref="colorConcept" />
        <Child ref="redConcept" />
    </ParentChild>
```

```
    <ParentChild>
        <Parent ref="colorConcept" />
        <Child ref="blueConcept" />
    </ParentChild>

</ConceptSystem>

<Concept id="colorConcept" />
<Concept id="redConcept" />
<Concept id="blueConcept" />
```

Programming languages like Java

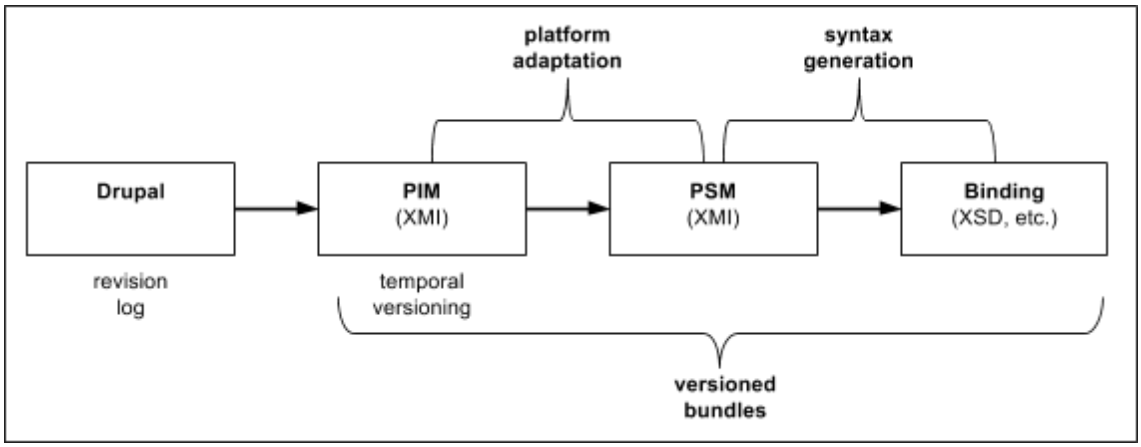
For the reason, that interfaces in several object oriented languages can't include properties, the PSM for those platforms should look as follows. The abstract classes with the “implements” association will become interfaces. The methods from the interfaces usually are generic. The properties from the abstract classes will be injected into the derived classes, as the PSM is derived from the PIM.

For programming languages, that don’t support interfaces, a different PSM has to be defined.

Versioning

The version discussion is based on Flavio's proposal for an evolution model for versioning.

Process model



Extended version of the original workflow, including three stages of versioning.

In the process model, we can identify three stages of versioning. First, we have a revision log in Drupal that documents every whatsoever small change. Second, we would like to implement a temporal versioning system for the PIM, based on Flavio's proposal. Third, we need a business versioning of the whole production package. The production package bundles everything from the PIM to the syntax bindings, including the PSM and all configuration files, rules, transformation programs etc. that are part of the build process.

Versioned things

We identified three groups of material that result in distinct versioning requirements.

1	**Process Independent Model (PIM)** ● PIM XMI ● PIM documentation (class level) ● Temporal evolution model
2	**Platform Specific Model (PSM)** ● Platform adaptation (incl. configuration, rules, transformation program) ● PSM XMI ● PSM documentation (class level)
3	**Syntax binding** ● Syntax generation (incl. configuration, rules, transformation program) ● Bindings (XSD, OWL, etc.)

Changes on the earlier stages influence the later stages but not the other way round. A change in the PIM, for example, usually results in a change of the actual binding. A change of the binding rules, however, does not influence the PIM.

Versioning rules

MISSING

First conclusion and open questions

Current agreement:

1. Stable monolithic releases (e.g., "DDI 4.0", "DDI 4.1") -- something, somewhere has changed. The monolithic includes everything (models, bindings, config files, rules, etc.).
2. Very granular versioning of all components according to change (views, objects, etc.)

⇒ This implies that multiple monolithic releases could include the same version of an object or view.

Open questions:

- Time intervals for monolithic releases?
- Bindings can change without changes in the PIM -- how to version these changes?
- One version of DDI can have multiple versions of one object—or only exactly one object?
- Do we use the temporal or evolutionary model? (Using the "becomes" or not?).

XML binding details

The discussion focused on the use of a common namespace and root element (<ddi/>) across all functional views.

DDI requirements for sub setting in DDI:

1. Simplicity
    - Easy to approach (subsets)
    - Easy to learn
    - Consistent across subsets
    - Simple
2. Subsets can be user-defined as well as pre-defined
3. Subsets can be validated
4. DDI is interoperable across applications
5. DDI is a well-defined canonical standard

XML binding approach: The DDI xml serialization will start with the DDI root element and provide a place for serializing all identified classes. This is referred to as the "flattened XML" structure. The XML schema for the xml binding will validate documents using this flat format.

All views create XML instances that validates against the flattened XML schema for the model.

In addition to the main XML binding schema, additional XML schema can be created that describes a subset of the main schema. These subsets of the schema will use the same root

namespace and allow validation again either the view's schema, a schema for a view that has a superset of the classes, or the whole model XML schema. These subsets may include serialization for a specific set of classes that are contained within a view. Relationships to classes that are not present in the view can also be removed from the schema for the view. These schema for views can be distributed to users to guide them for specific tasks or use cases. The XML instance created using the schemas for the views would validate against both the view's schema and the overall DDI schema.

Issues addressed:

- When a user creates a view, it can be processed by other software that understands a superset of the classes used in the view. (Interoperability)
- When a user's need expands beyond a simple view, their XML instance does not need to be converted to a new XML instance with a different schema. The current XML instance can expand and use additional classes that are present in the larger view of the model.
- Allows relationships from classes in the view to classes that are not in a view to be hidden from the user, allows the user to focus on the task the view was created for.

Alternative Approach:

One serialization with a single root using the flattened XML structure as in the previous proposal. For each view, instead of creating a subset of the schema, create schematron rules that can both describe and validate a view.