# DDI Lifecycle (MD) – Structure in Model and Bindings

Joachim Wackerow, 2015-04-06

## Contents

[Concept of Functional Views in the DDI Model 1](#_Toc416077632)

[UML Model 1](#_Toc416077633)

[XSD/XML Binding 5](#_Toc416077634)

[RDFS/OWL Binding 5](#_Toc416077635)

[Namespaces in General 6](#_Toc416077636)

[Production Framework 6](#_Toc416077637)

## Concept of Functional Views in the DDI Model

Functional Views are subsets of the whole model for a variety of purposes. They are realized in related packages in the model and in the XSD/XML and RDFS/OWL bindings. This approach enables the use of DDI for different audiences, hides complexity of the whole model, and supports interoperability.

These Functional Views represented in XSD/XML and RDFS/OWL are the parts of the specification which are used by users (software developers, metadata experts).

The Functional Views are a specific construct in the DDI model. They shouldn’t be understood as UML views which are related to views in software architecture and the related standard ISO/IEC/IEEE 42010[[1]](#footnote-1).

Here are two citations as background:

* The authors of UML[[2]](#footnote-2) state: “A view is a projection into a model. In the Rational Unified Process, the architecture of a system is captured in five interlocking views: the design view, process view, deployment view, implementation view, and use case view.”
* One of the authors of ISO/IEC/IEEE 42010 states in a paper[[3]](#footnote-3): “A view is a description of the system relative to a set of concerns from a certain viewpoint.”

The concept of Functional Views in the DDI model are loosely related to these ideas. Multiple Functional Views can have an overlap in terms of the concerns of multiple viewpoints.

## UML Model

The UML Model comprehends the Library and the Functional Views. The Library consists of all classes and data type definitions. All these items (classes, data types, Functional Views) are organized in packages.

### UML Version

It is intended that the model could be imported in common UML modeling tools. A DDI modeler might use UML modeling tools to validate the model according to multiple criteria. The related XMI serialization of the UML model is the basis for further use of the model like generation of the bindings and import in UML tools.

The broadly accepted UML version 2.4.1[[4]](#footnote-4) and the related XML metadata interchange format XMI version 2.4.1[[5]](#footnote-5) seem to be the appropriate versions. Canonical XMI[[6]](#footnote-6) should be an interoperable way to express the model. Canonical XMI is a subset of XMI which needs to meet multiple criteria in order to assure interoperability across UML tools. A XMI validation tool[[7]](#footnote-7) developed by NIST (National Institute of Standards and Technology) is available. This tool provides also browsing[[8]](#footnote-8) of the types of UML 2.4.1.

### Library

The Library consists of all classes and data type definitions. Both items are organized in packages. The class packages should reflect meaningful sets of information of the thing which is modelled, the data lifecycle.

### Class

A UML class is a classifier which describes a set of objects that share the same features, constraints, and semantics (meaning). A Class has the same unique name in different representations like Drupal software, UML expressed as XMI, DocBook for documentation purposes, XSD/XML and RDFS/OWL bindings, and possibly in program libraries and database schemes.

### Functional View

A Functional View can use classes (by reference only) within one or more packages but not from other Functional Views. Not all classes of a package have to be necessarily used by a Functional View. Functional Views have no class definitions for themselves. Functional Views are realized as packages.

The references to classes can be realized by the UML type “ElementImport”[[9]](#footnote-9). Not all UML tools can consume this type in XMI. A transformation to a custom XMI flavor can be done for these tools if needed.

### Release

Each Release comprehends the latest version of all classes and all Functional Views (both organized in packages). A Release is realized as a package as well. Packages can hold other packages in UML.

Releases should be published in not too frequent cycles like once the year. A regular publishing cycle could help planning of software development.

A new Functional View should only refer to the latest version of a class if possible. This way Functional Views reference only classes in the same Release.

Each Release is archived for an unlimited time in order to provide previous versions of classes for reuse and Functional Views for use.

This approach has the consequence that multiple Releases can hold copies of the same class definition. Classes with same names in different packages are allowed in UML. Classes are uniquely identified, see below.

Each Release is available in XMI representations. Multiple Releases can be imported into UML tools as needed.

#### Example Outline of Releases

* Package Release 2015-07-01
	+ Package Library Packages
		- Package DataCapture
			* Class Capture
			* Class InstrumentComponent
		- Package CoreProcess
		- Package Identification
		- Package Representations
		- Package Conceptual
		- Package Collections
	+ Package Functional Views
		- Package SimpleInstrumentView
* Package Release 2016-01-01
	+ Package Library Packages
		- Package DataCapture
		.
		.
	+ Package Functional Views
* Package Development
	+ Package Library Packages
	+ Package Functional Views

### Versioning

The classes within each package and the Functional Views are versioned. The whole model is published in releases. A specific release date acts as part of the identification of a release. The packages are versioned primarily for maintenance purposes.

The version of a package changes if the version of a contained class changes. The version of a Functional View changes if the version of a referenced class changes.

Each class and package has a property for holding the version value of this class. The statement “owl:versionInfo” should be used. It has clear semantics and is documented in OWL[[10]](#footnote-10).

This approach supports identifying the versions of classes and packages in the model and in the bindings. A software program could recognize that used classes (by the program) of an updated Functional View are not changed. The program could use in this case the newer version of the Functional View without any changes in the program.

The documentation for each class and package should indicate the version as well. This could be realized in the used DocBook format by the attribute section/@status. Further exploration is required to use the correct semantics and to make sure that the DocBook stylesheets render this information.

A three-level version number could be used. Major level for incompatible changes (existing functionality is changed), minor level for compatible changes (like an additional property of a class), and sub-minor level (like a change in the documentation without change of the definition of a class).

### Identification

Each class and package (including Functional Views) can be identified by three ways in parallel. Following XMI attributes are used:

* name[[11]](#footnote-11). The value is a unique name (which doesn’t change) in the scope of the DDI model. The version value could be stored in the attribute “label”12. This attribute can be used for any value. It is not clear yet if UML tools make use of this information.
* xmi:id[[12]](#footnote-12). The value is used for identification in a XML/XMI file. The value of this identifier should be a combination of the class name and its version: <name>\_<version>
* xmi:uuid12. The value is a universally unique identifier. It should be a resolvable URI according to OMG rules. This attribute can be used for identification beyond a single XML/XMI file. I.e., this applies for the model editing system in Drupal.

The name is a mandatory attribute. The attributes xmi:id and xmi:uuid are mandatory according to the rules of Canonical XMI.

### Naming Scheme for URIs

A naming scheme for URIs (as universally unique identifiers) of Releases, packages, Functional Views, and classes could be organized in this way:

* http://ddialliance.org/Specification/DDI-Lifecycle/MD/Release/*<date>*
* http://ddialliance.org/Specification/DDI-Lifecycle/MD/Package/*<package\_name>*/*<version>*
* http://ddialliance.org/Specification/DDI-Lifecycle/MD/FunctionalView/*<functionalview\_name>*/*<version>*
* http://ddialliance.org/Specification/DDI-Lifecycle/MD/Class/*<classname>*/*<version>*

### Reference

An xmi:id can be referenced by xmi:idref[[13]](#footnote-13) in the same XML/XMI file. The usage of xmi:idref for referencing inside a XML/XMI file is mandatory according to Canonical XMI. A xmi:uuid can be referenced across XML/XMI files or for other purposes by the XMI attribute href13.

## XSD/XML Binding

Functional Views have a XML wrapper element (the root) in their own namespace. The value of this namespace is the name of the Functional View without the version number. The version can be realized as XML attribute or child element.

All used classes are in a generic namespace “ddi” (without any version number). Classes which are used in multiple views exist as “definition copies” in each of these Functional Views. Therefore it is not possible to have the classes in the namespace of the Functional View. This would result in the existence of multiple definitions of the same class in multiple namespaces.

The described approach can validate XML instances according to a specific Functional View. The approach supports the idea of avoiding version numbers in namespaces.

A more elegant way than “definition copies” would be a reference to elements in the DDI namespace. An issue could be that a Functional View needs the whole Library as XML Schemas which can be an unwanted overhead for XML parsers. This is not intended by the idea of Functional Views as subset of the whole specification.

## RDFS/OWL Binding

A Functional View would be the name of the ontology (in the namespace of the Functional View like in the XSD/XML binding). All used classes would be in the generic namespace “ddi”. An ontology defines a vocabulary and has documentation purposes. An instance cannot be validated according to an ontology (Functional View). This is the case with all ontologies.

Work on validation of ontologies is in progress in the Semantic Web community. It is conceivable that a tool-based approach could validate an instance regarding the existence of required or allowed classes and their relationships.

## Namespaces in General

Having all DDI classes in one namespace can result in a single namespace with a large number of classes. There are examples of large schemas like the ontology schema.org[[14]](#footnote-14) with over 600 classes. Another option is using the namespaces of the UML packages where the classes are living in. This would reflect the organization in the UML model.

The advantage of this approach would be easier maintenance and easier orientation in the model. As Functional Views are using classes probably from only a few Library packages, it wouldn’t be a large load for XML parsers in the case of the XSD/XML binding.

## Production Framework

The production framework consists of the model editing system in Drupal and a set of tools (mostly XSLTs) to generate documentation and bindings. The production framework is built for the developers of the specification.

An advanced user might use the production framework in order to create custom products like a custom Functional View or program libraries.

### Maintenance Table

A maintenance table could hold information about each item in the model like Class, Library package, and Functional View. The above described properties of each element should be in this table: name, identifiers (xmi:id, xmi:uuid), version and previous version (owl:versionInfo), and status (release date or development). A related web service could be provided to support maintenance of software programs.

It could make sense to use parts of XMI or of the UML Human-Usable Textual Notation (HUTN)[[15]](#footnote-15) for this purpose.

1. ISO/IEC/IEEE 42010 Systems and software engineering - Architecture description. <http://www.iso-architecture.org/ieee-1471/>. [↑](#footnote-ref-1)
2. P. Grady Booch, James Rumbaugh, Ivar Jacobson: The unified modeling language user guide. Reading, Massachusetts, 1998. <http://www.icst.pku.edu.cn/course/uml/reference/the-unified-modeling-language-user-guide.9780201571684.997.pdf>. [↑](#footnote-ref-2)
3. P. 4-7. Rich Hilliard: Views and viewpoints in software systems architecture. First Working IFIP Conference on Software Architecture, San Antonio, February 1999. <http://web.mit.edu/richh/www/writings/hilliard99-ifip.pdf>. [↑](#footnote-ref-3)
4. <http://www.omg.org/spec/UML/2.4.1/> [↑](#footnote-ref-4)
5. <http://www.omg.org/spec/XMI/2.4.1/> [↑](#footnote-ref-5)
6. <http://www.omg.org/spec/XMI/CanonicalXMI/Beta2/PDF/> [↑](#footnote-ref-6)
7. <http://validator.omg.org/se-interop/tools/validator> [↑](#footnote-ref-7)
8. <http://validator.omg.org/se-interop/tools/browse-models> [↑](#footnote-ref-8)
9. P. 64-66. OMG Unified Modeling Language, Superstructure. Version 2.4.1. <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF>. [↑](#footnote-ref-9)
10. <http://www.w3.org/TR/owl-ref/#versionInfo-def> [↑](#footnote-ref-10)
11. P. 93. OMG Unified Modeling Language, Infrastructure. Version 2.4.1. <http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF>. [↑](#footnote-ref-11)
12. P. 13-14, OMG MOF 2 XMI Mapping Specification. Version 2.4.1. <http://www.omg.org/spec/XMI/2.4.1/PDF>. [↑](#footnote-ref-12)
13. P. 15, OMG MOF 2 XMI Mapping Specification. Version 2.4.1. <http://www.omg.org/spec/XMI/2.4.1/PDF>. [↑](#footnote-ref-13)
14. <http://schema.org/> [↑](#footnote-ref-14)
15. <http://www.omg.org/spec/HUTN/1.0/PDF> [↑](#footnote-ref-15)