Communications Document: DDI Moving Forward Modelling Topics

*Version 2.1*

# I. Overview

This document is intended to help the content modeling teams to consistently model some parts of the views they are producing. DDI 4 now has some abstract models covering specific areas, intended to be extended and specialized in particular views. This includes the “Collection” model, which will be used as the basis for grouping metadata items for various purposes. Another area is in the description of process flows, where a very generic process model has been created.

We will look at examples in both these areas, to show how the views should intersect with the abstract constructions held in the DDI 4 Object Library.

# II. Identification

Within the DDI 4.0 model, there is a base Identifiable object, which is extended by AnnotatedIdentifiable. This pair of objects are found in the inheritance chain of every other object in the model. As a result of this, several different attributes are made available to all of the objects in the DDI 4.0 model.



There are three properties on the Identifiable object, all of which are required. These values are the same set of information used in DDI Lifecycle for identification: agency, id, and version.

All of the properties found on AnnotatedIdentifiable are optional, and one – local ID – is repeatable.

*BasedOnObject* is a structure which references another identifiable object in DDI, indicating that the current object is a copy of another object.

*isUniversallyUnique*, when set to “true”, indicated that the id property of the object contains a universally unique identifier (UUID), and is not scoped to the agency like the normal DDI id.

*localID*  is a structure where you can place any type of identifier, and indicate what system the identifier comes from. There is a reserved word here – “Name” – which indicates that the localID is the name of the object.

*versionDate* is a property which contains the date of the creation or latest update of the object.

*versionRationale* provides a description of why the object was created or updated.

*versionResponsibility* is an indicator of the party responsible for making the change.

These properties are available for all objects in the DDI model, and duplicative properties should not be declared when new objects are modeled.

# III. Names, Labels, Descriptions, and Other Standard Properties

There is a list of properties which should – when needed for an object – be modeled in a consistent fashion. These are listed in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Type** | Definition | **Notes** |
| Name | - |  | Use localID with reserved type value of “Name” |
| Label | Xs:string plus a set of attributes (typeOfLabel, locationVariant, validForStartDate, validForEndDate, and maxLength). | A linguistic sign denoting a general concept (term) or individual concept (appellation). Intended to be used for display purposes. | Example includes labels as found in the stats packages. |
| Description | StructuredString |  |  |
| Definition | InternationalString | A natural language statement of the meaning of the concept. It may be intensional, starting with a previously defined concept and providing differentia, or extensional, providing delineating kinds (i.e. Human teeth are incisors, canines, bicuspids, and molars) |  |
| Xml:lang | Xs:language | The language used by a particular object (for example, a French-language video) | Only for non-textual objects; most textual fields already have the xml:lang property |
| Comments | Xs:string | A memo created by the creator or maintainer of an instance of an object, not to be shared publically. |  |
| Notes | - | - | Use the complex data type Note; this is a public-facing Note, as opposed to internal comments. |

# IV. Complex Data Types

All complex data types (that is, the set of complex structures which are treated within the Drupal modeling platform as primitives, as for the values of properties) are located in the Complex Data Types package. There is a distinct style of modeling these: each complex data type which has a primary content will have a property named “content” of whatever primitive type is needed. Complex data types will not be extensions of the primitive type of their primary content.

# V. Relationships

DDI objects are associated via binary relationships. Relationships have cardinality constraints, e.g. 1..1, 1..n, 0..n, and can be of different types, i.e. *aggregation*, *composition*, and *neither* (simple, untyped). Even though most relationships in the model are undirected, in the Drupal they are always defined within one of the objects participating in the relationship, i.e. the *source*; the object at the other end of the relationship is called the *target*. Similarly with names: the predicate represented in a relationship name does not impose a direction since the implicit converse predicate is also true in all undirected relationships. It’s important to note that this is just a convention used in the Drupal and by no means imposes an actual conceptual direction in the association.

For instance, *RepresentedVariable* has a relationship with *ValueDomain*. The relationship is defined in *ValueDomain*, which is the source, and it is named by a predicate, i.e. *takesValuesFrom*. This seems to imply a direction from *RepresentedVariable* to *ValueDomain*. However, the converse predicate, i.e. *providesValuesFor*, although not represented in the model, is also valid since such a relationship is conceptually undirected. In general, and unless otherwise indicated, all relationships in the Drupal are undirected.

# VI. Processes

The process model in DDI 4 provides a number of generically useful objects: the Process Step object can be used to describe processes at any level, and is subclassed into a set of objects which can be used to describe logical flows. The Control Construct object is a specialization of the Process Step for this purpose, and it has in turn a number of specific objects which extend Control Construct: Sequence, Repeat While, Repeat Until, Loop, If Then Else, and Act. Of these, Act is the use of a metadata item within the flow, but not as an input or output (Act covers questions, statements, instructions, etc.)

Each of the Control Constructs can also act as aggregations of other Control Constructs, although that is not shown in the diagram. This provides a wealth of extension points for use in other models within DDI 4.



Process steps have Inputs and Outputs, and are themselves composed of other Process Steps. There is a specialized type of Process Step which is the Service which performs a Process Step. This specialized type has an Agent, which is the individual, organization, or machine which is performing the Service.

The DDI 4 process model is a generic, abstract one, which can be used for many purposes: to describe how a process *should* be performed, to provide an historical description of a process as it *was* performed, etc.

The additional semantics around what the process model is to be used for are added when the Process Step or one of its sub-types is extended for use in a view. For example, when describing a Simple Instrument, the Control Construct object is extended for explaining the flow logic within a data collection process. This is not an historical description, but instead is providing the pattern in which the data collection is to be conducted.

In order to incorporate the Process Step into their description of the flow of data collection, the Simple Instrument uses an object specific to data collection as an extension of Control Construct – Instrument Component. This extended object provides the semantics needed to explain how the abstract process model will be used.



Here, we see the extension of Control Construct into Instrument Component, which is itself extended by Capture, an abstract class which is extended by the subclasses Measurement (for data coming from a device) and Question (for data coming from a questionnaire).

# VII. Collections

DDI 4 introduces a generic collection structure that can be used to model different types of groupings, from simple unordered sets to all sorts of hierarchies, nesting and ordered sets/bags. In addition, they can be extended with richer semantics (e.g. generic, partitive, and instance, among others) to support a variety of DDI 3.2 and GSIM structures, such as Node Sets, Schemes, Groups, sequences of Process Steps, etc.

A collection consists of a *container*, which could be either a set (i.e. unique elements) or a bag (i.e. repeated elements) together with an *order relation* that establishes the order of precedence between members in the container. The order relation can be either *total* or *partial*. When every pair of members from the container is in the order relation we say it is total, otherwise it is partial. A total order always defines a sequence where partial order can be used to model hierarchies. The same collection of members can be structured in different ways by different order relations. For instance, a collection with members **a**, **b**, **c**, **d**, **e** can be organized into a sequence by the total order {(**a**, **b**), (**b**,**c**), (**c**, **d**), (**d**, **e**)} (diagram on the left) or into a hierarchy by the partial order {(**a**,**b**), (**a**,**c**), (**c**,**d**), (**c**,**e**)} (diagram on the right). Please note that we don’t include in the relation pairs implied by transitivity, e.g. (**a**,**c**) in the sequence and (**a**, **e**) in the hierarchy.

|  |  |
| --- | --- |
|  |  |
| *Sequence* | *Hierarchy* |

The next diagram shows the UML representation of the collection structure. The **Collection** class represents the container and the **Member** class its elements. An optional **orderRelation**, reified as an association class to allow for subtyping, provides the predecessor-successor pairs in the order relation, which is optional. The **type** attribute in **Collection** indicates whether the container is a *bag* or a *set*, and in **Member** whether the order relation is *total* or *partial*. The left-hand side of the figure shows an example of a **NodeSet** modelled as a specialization of the generic collection structure on the right-hand side. Note that multiple order relations are possible for the same container, e.g. the **parent-child** and **part-whole** shown.

